UNIVERSITY OF CALIFORNIA, SAN DIEGO

On the Power of the Basic Algorithmic Design Paradigms

A dissertation submitted in partial satisfaction of the requirements for the degree
Doctor of Philosophy

in

Computer Science

by

Sashka Tchameva Davis

Committee in charge:

Russell Impagliazzo, Chair
Samuel Buss
Fan C. Graham
Daniele Micciancio
Ramamohan Paturi

2008

DEDICATION

To my husband, my son, and my parents for their love and support.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

My deepest gratitude goes to my advisor Russell Impagliazzo. The road of my doctoral studies was long and his guidance was invaluable.

The work presented in Chapter 2 appeared in the proceedings of the 2004 SIAM SODA conference, “Models of Greedy Algorithms for Graph Problems”, by Sashka Davis and Russell Impagliazzo, [20]. The full version “Models of Greedy Algorithms for Graph Problems”, by Sashka Davis and Russell Impagliazzo, was published in Algorithmica in 2007, [21].

The results in Chapter 3 are part of manuscript “A General Model for Backtracking and Dynamic Programming Algorithms”, by Josh Buresh-Oppenheim, Sashka Davis and Russell Impagliazzo, [16], to be published in the future.

The results presented in Chapters 4 and 5 together with results which are not included in this dissertation are part of manuscript “A Stronger Model for Dynamic Programming Algorithms”, by Josh Buresh-Oppenheim, Sashka Davis and Russell Impagliazzo, [17], to be published in the future.

VITA

1991 Bachelor of Science, Technical University Sofia, Bulgaria

1999 Master of Science, Rochester Institute of Technology, USA

2008 Doctor of Philosophy, University of California, San Diego, USA

PUBLICATIONS

Josh Buresh-Oppenheim, Sashka Davis, Russell Impagliazzo, "*A Stronger Model for Dynamic Programming Algorithms*", 2008 manuscript in preparation.

Josh Buresh-Oppenheim, Sashka Davis, Russell Impagliazzo, "*A General Formal Model for Backtracking and Dynamic Programming Algorithms*", 2008 manuscript in preparation.

Sashka Davis, Russell Impagliazzo, "*Models of Greedy Algorithms for Graph Optimization Problem*", Algorithmica 2007.

Sashka Davis, "*Evaluating Algorithmic Design Paradigms*", 2006 Grace Hopper Celebration of Women in Computing.

Sashka Davis, Jeff Edmonds, Russell Impagliazzo, "*Online Algorithms To Minimize Resource Reallocations and Network Communication*", Proceedings, Lecture Notes in Computer Science 4110, Springer 2006.

Sashka Davis, Russell Impagliazzo, "*Models of greedy algorithms for graph problems*", Proceedings of the 2004 SIAM SODA.

Sashka Davis, "*Hu-Tucker Algorithm for Building Optimal Alphabetic Binary Search Trees*" Technical Report RIT-99-019, Computer Science Dept., Rochester Institute of Technology, 1999.

FIELDS OF STUDY

Bachelor of Science in Computer Engineering

Master of Science in Computer Science
Professor Stanislaw Radziszowski

Doctor of Philosophy in Computer Science
Professor Russell Impagliazzo

ABSTRACT OF THE DISSERTATION

On the Power of the Basic Algorithmic Design Paradigms

by

Sashka Tchameva Davis
Doctor of Philosophy in Computer Science

University of California, San Diego, 2008

Russell Impagliazzo, Chair

This dissertation formalizes the intuitive notion of the basic algorithmic paradigms. We present three formal models which aim to capture the intrinsic power of greedy, backtracking and dynamic programming algorithms. We develop lower bound techniques for proving negative results for all algorithms in all models, which allow us to make strong statements about the limitations of each paradigm.

[14] designed the *Priority* algorithms, a formal model of greedy algorithms for scheduling problems. We generalized the priority model to arbitrary problem domain and in particular graph problems and develop a lower bound technique for proving negative results for the class of all priority algorithms. We use the lower bound technique to show that finding shortest path in graphs with negative weights cannot be solved by a priority algorithm. We also prove that Dijkstra's algorithm is inherently adaptive and cannot be made non-adaptive. We show inapproximability results within the model for minimum weighted vertex cover, minimum metric Steiner tree, and maximum independent set problems. We develop a new 1.8-approximation scheme for the $Steiner(1,2)$

problem.

[1] presented a model of backtracking and dynamic programming algorithms called *prioritized Branching Trees* (pBT). We generalize their model to allow free branching and call this new model *prioritized Free Branching Tree* (pFBT) algorithms and developed a lower bound technique for proving negative results for randomized priority algorithms, pBT and pFBT algorithms. We use the technique to prove that pBT algorithms require exponential width to solve the 7-SAT problem and that pFBT algorithms require width $2^{\Omega(\sqrt{n})}$ to solve the 7-SAT problem.

Bellman-Ford is a classical dynamic programming algorithm and we show that pBT algorithms require width of $2^{\Omega(n^{1/9})}$ to solve the shortest path problem in graphs with negative weights exactly.

Next we develop a stronger model of dynamic programming algorithms called *prioritized Branching Programs* (pBP). pBP algorithms can simulate pBT algorithms at no additional cost but also capture the notion of memoization which we believe is an essential part of the dynamic programming paradigm. We show that this class of algorithms can solve the shortest paths in graphs with negative weights but no negative cycles efficiently. We also show that two pBP sub-models can be simulated by pBT algorithms.

Chapter 1

Do We Need Formal Models of the Algorithmic Paradigms?

In an algorithm design class, we are taught the basic algorithm paradigms such as divide-and-conquer, greedy algorithms, backtracking and dynamic programming. The paradigm is taught by an intuitive example together with a number of counter examples. Intuitive formulations, while easy to understand, do not allow us to answer the following natural questions. Suppose we have an optimization problem that we want to solve.

1. What algorithmic design paradigm can help? For example: Do we need dynamic programming to solve single source shortest path in graphs with negative weights, or can we solve the problem using some greedy strategy? Do we need flow algorithms to find a maximal matching in bipartite graphs or can we use a simpler and more efficient dynamic programming approach?

For example, greedy algorithms are typically simple and have efficient implementations. Hence, we would like to know whether we can find a greedy algorithm that solves a given optimization problem. Suppose that, after much thought and effort we were unable to find a greedy solution for our problem. Then we would like to be able to make a strong statement that no greedy algorithm exists that solves the problem exactly. To do so, we need a formal model of greedy algorithms. Once we certify that our problem is hard for all greedy algorithms in the model, then we try the next "more powerful" technique, e.g., a backtracking approach, or dynamic programming.

2. Is a given algorithm optimal? If we were lucky and found an algorithm for our problem using some algorithmic paradigm, then the next natural question to ask is whether our solution is the most efficient solution within this class of algorithms. We could give a precise answer to such a question if we are able to prove lower bounds on resources for all algorithms that fit a given class (Note that the space of such algorithms is infinite). After formalizing the algorithm class, if we prove a lower bound matching the resources used by our algorithm then we have a certificate that our algorithm is optimal within the paradigm and cannot be improved without radical change. Suppose we exhaust all known algorithmic techniques and still have not been able to solve our problem, then we face the next question.

3. How good an approximation scheme can we get? If we cannot solve our problem exactly using the formalized approaches then we settle for approximation. The question here is what algorithmic technique delivers the best approximation. And how do we know that a given heuristic is really the best? We can certify a given approximation scheme as optimal, similarly as in the case of exact algorithms, if we prove a matching lower bound on the approximation ratio achievable by any other algorithm within a given class.

Another aspect of this line of work is a general *characterization of the power of the different approaches to optimization.* Intuitively greedy algorithms are efficient but weak, and whatever problems we can solve using greedy algorithms we can solve using say dynamic programming algorithms. But we would like to back this intuition with strong formal reasoning, which is impossible as long as our definitions are intuitive and our lower bounds hold for a single algorithm and not for an infinite class of algorithms.

To answer such questions, we need formal models that capture the intuitive notion of a given paradigm, along with a technique for proving lower bounds on the resources used by all algorithms that fit the given model. Not only can this formal approach be used to answer the questions above but it can also help us first to better understand the intrinsic structure or hardness of our problem, by knowing what algorithmic paradigms can or cannot solve it. We also learn the strength and weaknesses of the known algorithmic design techniques, which we measure by the kinds of problems they can solve exactly or approximately, and the types of problems that are hard for the technique. Third, by understanding the problems and the paradigms better, we improve our skills as algorithm designers.

1.1 History

Four formal models of algorithmic paradigms have been developed so far. [14] and [20] formalized the intuitive notion of greedy algorithms, [1] designed a formal model for backtracking and dynamic programming, [16] generalized the model of [1] and [17] defined a stronger model for dynamic programming algorithms.

1.1.1 Priority Model

Borodin, Nielsen, and Rackoff ([14]) gave a model of greedy-like algorithms for scheduling problems and [7] extended their work to facility location and set cover problems. [14] built a formal model of greedy algorithms for scheduling problems, which they called Priority algorithms. [20] generalized the Priority models to any problem domain and instantiated it for graph optimization problems, e.g., shortest paths, vertex cover, spanning trees, independent set, and minimum Steiner trees. Consider the Vertex Cover problem. Within the framework of Priority algorithms an instance of the problems is given by a set of vertices of the graph. Each vertex is described by its name and a list of the names of its neighbors. At each iteration a Priority algorithm chooses an ordering of the remaining vertices of the graph. It then considers the first vertex in the order and commits to a decision to add the vertex to the cover or reject it. The decision whether to accept or reject a vertex is irrevocable and can only depend on the current and previously considered vertices but not on future unseen vertices. The algorithm halts when decisions for all vertices are made.

[20] generalized the lower bound technique of [14] by abstracting away the domain specific details and defined it as a combinatorial zero-sum game between two players, an Adversary and a Solver. A strategy for the Adversary in the game establishes a lower bound on the resources needed by *any* Priority algorithm, while existence of a Priority algorithm can be used by the Solver player as a strategy in the game. Priority algorithms were shown to be weaker than dynamic programming algorithms ([14, 20]), and their power has been characterized by proving lower bounds for a variety of problems. We would like to point out that Priority algorithms are a very general model of greedy algorithms which captures the majority of the intuitively named greedy algorithms in the literature, such as: Kruskal's and Prim's algorithms for minimum spanning trees, Dijkstra's single source shortest path algorithm, the known greedy 2-approximation of weighted vertex cover problem, the greedy approximations for set cover problem, facility location, job and interval scheduling problems.

The Priority algorithm model resembles that of on-line algorithms. In both models, decisions affecting the output have to be made irreversibly based on partial information about the input (See Sect. 2.1.1 for a discussion about Priority algorithms and on-line algorithms). For this reason, the techniques used to prove bounds for Priority algorithms often are borrowed from the extensive literature on on-line algorithms (See [13] for a good overview). However, where an on-line algorithm sees the parts of its input[1] in an adversarial order or one imposed by some real-world constraint such as availability time, a Priority algorithm can specify the order in which inputs are examined. [14] considered two variants: *FIXED Priority* algorithms where this order is independent of the instance and constant throughout the algorithm, and *ADAPTIVE Pri-*

[1]To clarify what we mean by "parts of the input", consider the on-line vertex cover problem. An on-line algorithm will see each vertex of the graph together with its adjacency list, one at a time, and must decide whether to add the vertex to its solution or not. Hence, for this problem the "parts of the input" refers to the vertices of the graph. For other graph problems the edges of the graph might be more appropriate "part of the input".

ority where the algorithm can change the order of future parts based on the part of the instance that it has seen (See Sect. 2.2 for precise definitions of Priority algorithms). They also defined some sub-classes of "intuitive" Priority algorithms: *GREEDY* Priority algorithms which are restricted to make each decision in a locally optimal way[2], and *MEMORYLESS* adaptive Priority algorithms, which must base decisions only on the set of previously accepted data items.

[14] proved many non-trivial upper and lower bound results for a variety of scheduling problems (interval scheduling with unit, proportional, and arbitrary profits; job scheduling; minimum makespan). They showed a separation between the class of ADAPTIVE Priority algorithms and FIXED Priority algorithms, by proving a lower bound of 3 on the approximation ratio achieved by any FIXED Priority algorithm for interval scheduling on identical machines with proportional profit and observed an ADAPTIVE Priority algorithm with approximation ratio 2 for the same problem. For the interval scheduling problem with proportional profit [14] proved that the Longest Processing Time heuristics is optimal within the class of FIXED Priority algorithms. They also proved a separation between the class of deterministic and randomized Priority algorithms. The problem [14] considered is interval scheduling with arbitrary profits. They showed a lower bound of Δ (the ratio of the maximum to the minimum unit profit among all intervals) on the approximation ratio achieved by any ADAPTIVE Priority algorithm, for multiple and single machine configurations. However, if a FIXED Priority, not necessarily greedy, algorithm is given access to randomness, then it can achieve an approximation ratio of $O(\log \Delta)$.

Angelopoulos and Borodin ([7]) proved that no ADAPTIVE Priority algorithm can achieve an approximation ratio better than $\ln n - \ln \ln n + \Theta(1)$ for the set cover problem. This bound is tight because the greedy set cover heuristic, classified as an ADAPTIVE Priority algorithm, achieves the bound. [7] also considered the unrestricted

[2]In the context of scheduling problems a GREEDY Priority algorithm would schedule a job, if the job does not violate the current schedule. In other words the "greediness" is a property of the decisions made by the algorithm, not the ordering of the jobs.

facility location problem, and proved a tight bound of $\Omega(\log n)$ on the approximation ratio achieved by any ADAPTIVE Priority algorithm, which is matched by the known greedy heuristic for the problem. For the metric facility location problem, they were able to show a tight bound of 3 on the approximation ratio achieved by FIXED Priority algorithms and a lower bound of 1.463 on the approximation ratio achieved by any MEMORYLESS or GREEDY Priority algorithm.

Borodin, Boyar and Larsen ([11]) proved a lower bound of $\frac{4}{3}$ for the vertex cover problem in the general Priority model, where a data item encodes a vertex name along with the names of its neighbors. They also considered the independent set and vertex coloring problems in more restrictive Priority models, for example, where the priority function orders the vertices of the graph based on the degree of the vertex, excluding the names of the vertices adjacent to it. With this restriction on the priority function they proved stronger bounds on the approximation ratio achieved by such Priority algorithms.

Angelopoulos further studies two different extensions of Priority algorithms framework. In [5] he defines randomized Priority algorithms and proves lower bounds on the approximation ratio obtained by such algorithms for facility location and makespan problems. In [6] the author defines a restricted model of Priority algorithms, and shows lower bounds on the approximation ratio achieved by such algorithms for complete facility location and dominating set problems.

Papakonstantinou ([39]) studies the performance of different classes of Priority algorithms (MEMORYLESS, GREEDY, and FIXED Priority algorithms) for Job Scheduling problems and defines a hierarchy of memoryless priority algorithms.

1.1.2 Prioritized Branching Trees

[1] defined a formal model for backtracking and dynamic programming algorithms, called prioritized Branching Trees (pBT). A pBT algorithm like backtracking and dynamic programming algorithms, maintains multiple solutions to subproblems and each computation step extends an already existing solution. [1] defined a hierarchy of

submodels based on the levels of adaptivity in how the algorithm chooses its ordering rule: fixed, adaptive and fully adaptive pBT algorithms.

[1] were able to show a variety of non-trivial lower bounds. First they showed that the ability to branch and maintain multiple solutions to a problem separates makes them more powerful than the priority algorithms. [14] considered interval scheduling with proportional profit on a single machine and proved no adaptive priority algorithm can solve the problem and achieve an approximation ratio of $\frac{1}{3}$ (this lower bound is matched by the LPT, the longest processing time first, algorithm). [1] designed an adaptive-order pBT algorithm of width 2 and showed that it achieves an approximation ratio of $1/2$.

[1] showed an exponential separation between the power of fixed and adaptive pBT algorithms by proving that any fixed pBT algorithm requires exponential width to solve a 2-SAT instance but adaptive pBT algorithms can solve 2-SAT using linear-width. For the 3-SAT problem [1] showed that fully-adaptive pBT algorithms require exponential width and exponential depth first size.

For the knapsack problem the standard approximation algorithm which either accepts or rejected the highest profit item and then sorts items according to their unit profit (profit to weight ratio) and takes them in this order if possible, can be seen as a width 2 pBT algorithm. But [1] showed that priority algorithm cannot achieve an approximation ratio better than $n^{1/4}$. [1] also showed that any adaptive pBT will require width $\binom{n/2}{n/4}\Omega(2^{n/2}/\sqrt{n})$ to solve the Subset-Sum problem exactly.

1.2 Techniques

The main techniques used to establish the results in this dissertation are competitive analysis of online algorithms, the probabilistic method and the principle of deferred decisions.

All formal models which we would discuss in this dissertation resemble on-line algorithms in the sense that the algorithms examine the instance one piece at a time in the same way in which an online algorithm sees the input. Competitive analysis and zero sum games have been applied to the analysis of priority algorithm, the formal model of greedy algorithms. Good references for online algorithms and competitive analysis include [13, 23].

For the analysis of prioritized Branching Tree, prioritized Free Branching Tree algorithms we utilized many basic techniques from probabilistic method union bounds; martingales; concentration inequalities; together with some basic counting techniques from discrete mathematics. Good references regarding the probabilistic methods include [4, 34, 36]. We also used the principle of deferred decisions to construct the input to those algorithms in stages. [32] contains a good description of the method.

Chapter 2

Priority Algorithms

2.1 Why Study the Greedy Paradigm?

The greedy algorithm paradigm is one of the most important in algorithm design, because of its simplicity and efficiency. Greedy algorithms are used in at least three ways: they provide exact algorithms for a variety of problems; they are frequently the best approximation algorithms for hard optimization problems; and, due to their simplicity, they are frequently used as heuristics for hard optimization problems even when their approximation ratios are unknown or known to be poor in the worst-case. To cover all the uses of greedy algorithms, from simple exact algorithms to unanalyzed heuristics, one needs to study a cross-section of problems from the easiest (minimum spanning tree) to the hardest (NP-complete problems with no known approximation algorithms).

While greedy algorithms are simple and intuitive, they are frequently deceptive. It is often possible to generate many greedy algorithms for a problem, and one's first choice is often not the best algorithm. How do we know whether an algorithm is the best possible within a given algorithmic paradigm? Here, "best" can mean "most efficient", or "the best possible approximation ratio", or even "conceptually simplest". To clarify the last criterion, consider the minimum spanning tree problem. Kruskal's algorithm for minimum spanning tree is in some sense simpler than Prim's algorithm, because it just scans through the edges in sorted order, rather than dynamically growing a tree. Hence we could ask the question: can all greedy algorithms be conceptually simplified?

The priority model allows one to formally address all of these uses of greedy algorithms and issues in design of greedy algorithm. One can use this model to:

1. Tell when a known but slightly complicated greedy algorithm cannot be simplified. This can be done by defining a sub-model of "simple" priority algorithms, and showing that this subclass is weaker than the general priority model. (See the definitions of FIXED and ADAPTIVE priority algorithms in Sect. 2.2 and the separation results in Sect. 2.4.1).

2. Show that sometimes greedy approximation algorithms need to be counter-intuitive. This can be done by defining a sub-model of "intuitive" priority algorithms, and showing that this subclass is weaker than the general priority model. (See the definition of MEMORYLESS priority algorithms in Sect. 2.5).

3. Formalize the intuition that greedy algorithms are weaker than some of the other paradigms, by proving lower bounds for priority algorithms for problems with known algorithms of a different paradigm.

4. Prove that the known greedy approximation algorithm for a problem cannot be improved, by showing a matching lower bound for any priority algorithm.

5. Rule out the possibility of proving a reasonable approximation ratio for any greedy algorithm for a hard problem. This is particularly interesting for problems where greedy algorithms are used as heuristics.

6. Use the intuition behind the lower bound proofs to deepen our understanding of the intrinsic power and limitations of the algorithmic technique to design better algorithms.

2.1.1 What Is a Greedy Algorithm?

The term "greedy algorithm" has been applied to a wide variety of optimization algorithms, from Dijkstra's shortest path algorithm to Huffman's coding algorithm. The pseudo-code for the algorithms in question can appear quite dissimilar. There are few high-level features common to most greedy algorithms, unlike say divide-and-conquer algorithms that almost always have a certain recursive structure, or dynamic programming algorithms, which almost always fill in a matrix of solutions to sub-problems. What do these algorithms have in common, that they should all be placed in the same category?

A standard undergraduate textbook by Neapolitan and Naimipour ([38]) describes the greedy approach as follows: a greedy algorithm "grabs data items in sequence, each time taking the one that is deemed 'best' according to some criterion, without regard for the choices it has made before or will make in the future." This seems to us a fairly clear and concise informal working definition, except for the words "without regard for the choices it has made before" which we think does not in fact apply to most of the "canonical" greedy algorithms. (For example, a graph coloring algorithm that assigns each node the first color not used by its neighbors seems to be a typical greedy algorithm, but certainly bases its current choice on previously made decisions.)

This is the sense of a greedy algorithm the priority model is meant to capture. More precisely, a priority algorithm:

Definition 1. *(Informal definition of PRIORITY algorithms)*

A priority algorithm:

1. *Views the instance as a set of "data items".*
2. *Views the output as a set of "choices" (decisions) to be made, one per "data item".*

3. *Defines a "criterion" for "best choices", which orders data items. (Making this formal leads to two models, FIXED vs. ADAPTIVE priority algorithms.)*

4. *In the order defined by this criterion, makes and commits itself to the choices for the data items. Never reverses a choice once made (i.e., decisions are irrevocable).*

5. *In making the choice for the current data item, only considers the current and previous data items, not later data items.*

All but the third point are also true of on-line algorithms. The main difference is that on-line algorithms have the order of choices imposed on them, whereas priority algorithms can define this order in a helpful way. For example, an on-line algorithm for the minimum cost spanning tree problem is given the edges of the graph, one at a time, and must make a decision whether to add the edge to the solution or not. The on-line algorithm has no control of the order in which the edges will appear. In contrast a FIXED priority algorithm will *choose* the edge of the graph, say whose weight is minimum (other priority functions are possible) and then will decide whether to add it or not (continuing until all edges of the graph are considered). Many of the lower bound techniques here, in [14], and in [7] are borrowed from the extensive on-line algorithm literature.

Are the characteristics listed above the defining features of "greedy algorithms"? Many of the known algorithms can be classified as priority algorithms (ADAPTIVE or FIXED). For example, Prim's and Kruskal's algorithms for the minimum cost spanning tree problem are classified as ADAPTIVE and FIXED priority[1] algorithms, respectively. Dijkstra's single source shortest path algorithm also can be seen to fit the ADAPTIVE priority model. The known greedy approximation [30] for the weighted vertex cover problem (WVC) can be classified as an ADAPTIVE priority algorithm. The greedy

[1]For examples of a formalization of specific greedy algorithms in the framework of priority algorithms see Adaptive Contract algorithm described in Sect.2.4.3 and formalized in Appendix A; and also the Adaptive WIS algorithm described in Sect. 2.5.2.

approximation for the independent set problem [25] also fits our model. As noted in [7], the best known greedy approximation algorithm for the set cover problem also fits the framework of priority algorithms, and similarly the greedy algorithms for the facility location in arbitrary and metric spaces have priority models.

Although Matroids and Greedoids provide theoretical foundation about greedy algorithms our current work goes beyond the greedy algorithms on Matroids. For example, the activity selection problem ([19]) and the problem of finding single source shortest paths in graphs with non-negative weights, to name a few, have greedy algorithms which are not covered by the theory of Matroids, yet they fit the framework of ADAPTIVE priority algorithms. Also the greedy algorithm on Matroids seem to fit the framework of FIXED priority algorithms and we show later (see Sect. 2.4.1, Theorem 4) that the class of FIXED priority algorithms is properly contained in the class of ADAPTIVE priority algorithms.

However, the term "greedy algorithm" is used in at least one other sense which the priority model is not meant to capture. A hill-climbing algorithm that uses the "steepest ascent" rule, looking for the local change that leads to the largest improvement in the solution, is frequently called a "greedy hill-climbing" or simply "greedy" algorithm. The Dijkstra heuristic for Ford-Fulkerson, which finds the largest capacity augmenting path during each iteration, is "greedy" in this sense. As far as we can tell, there is no real connection between this sense of greedy algorithm and the one defined above. We make no claims that any of our results apply to steepest ascent algorithms, or any other classes of algorithms that are intuitively "greedy" but do not fit the above description[2].

[2]While there are a few uses of the phrase "greedy algorithm" that do not seem to fit the priority model, this seems more a matter of the inherent ambiguity of natural language than a weakness in the model. A useful scientific taxonomy will not always classify things according to common usage; e.g., a shellfish is not a fish. There will also always be borderline objects that are hard to classify, e.g., is a marsupial a mammal? We should not be overly concerned if a few intuitively greedy algorithms go beyond the restrictions of the priority model and require global information about the instance. However, this will be motivation to try to extend the model in future work.

2.2 Priority Models

To make Definition 1 precise, we need to specify a few components: What is a decision? What choices are available for each decision? What is the "data item" corresponding to a decision? What is a criterion for ordering decisions? The answers to these questions will be problem-specific, and there may be multiple ways to answer them even for the same problem. In this section, we give a format for specifying the answers to these questions, leaving parameters to be specified later to model different problems.

The general type of problem we are discussing is a combinatorial optimization problem. Such a problem is given by an instance format, a solution format, a constraint (a relation between instances and solutions), and an objective function (of instance and solution, giving a real number value). The problem is, given the instance, among the solutions meeting the constraint, find the one that maximizes (or minimizes) the objective function.

We want to view an instance as a set of ***data items***, where a solution makes one decision per data item. Let Γ denote the type of a data item; thus an instance is a set of items of type Γ, $I \subseteq \Gamma$. (We are not necessarily assuming that every subset of data items constitutes a valid instance. For scheduling problems any sequence of jobs is a valid instance. Instances of graph problems have more structure, which prevents some sets of data items from being valid graphs. We also frequently want to restrict to instances with some global structure, e.g., metric spaces or directed graphs with no negative cycles.) The solution format will assign each $\gamma \in I$ a ***decision*** σ from a set of ***options*** Σ, so a *solution* is a set of the form $\{(\gamma_i, \sigma_i) | \gamma_i \in I\}$.

For example, for k-colorings of graphs on up to n nodes, we need to assign colors to nodes. So Γ should correspond to the *information available about a node* when the algorithm has to color it, and $\Sigma = \{1, \ldots, k\}$ are the k colors. Γ is not uniquely defined, but a natural choice, and the one we will consider here, is to let the algorithm see the name and adjacency list for v when considering what to color v. Then the data item corresponding to a node is the name of the node and the adjacency list of a node, i.e., Γ would be the set of pairs, $(NodeName, AdjList)$, where a $NodeName$ is an integer from $\{1, \ldots, n\}$; $AdjList$ is a list of $NodeNames$. We then view G as being given in adjacency list format: G is presented as the set of nodes v, each with its adjacency list $AdjList(v)$[3]. More generally, a **node model** is the case when the instance is a (directed or undirected) graph G, possibly with labels or weights on the nodes, and data items correspond to nodes. Here, Γ is the set of pairs or triples consisting of possible node name, node weight, or label (if appropriate[4]), and a list of neighbors. For example when the instance is a weighted graph then the data item is a triple $(NodeName, NodeWeight, AdjList)$. Σ varies from problem to problem; often, a solution is a subset of the nodes, corresponding to $\Sigma = \{accept, reject\}$.

Alternatively, in an **edge model**, the data items requiring a decision are the edges of a graph. In an edge model, Γ is the set of (up to) 5-tuples with two node names, node labels or weights, and an edge label or weight (as appropriate to the problem). In an edge model, the graph is represented as the set of all of its edges. Again, the options Σ are determined by the problem, with $\Sigma = \{accept, reject\}$ when a solution is a subgraph.

As another example, [14] consider scheduling problems, where jobs are to be scheduled on p identical machines. Here, we have to decide whether to schedule a job, and if so, on which machine and at what starting time. So $\Sigma = \{(m_i, t) | 1 \leq i \leq p, t \in R\} \cup \{Not\ scheduled\}$. They allow the algorithm to see all information about a job when scheduling it. A data item is represented by (a_i, d_i, t_i, w_i), where a_i is the arrival time of job i, d_i its deadline, t_i its processing time, and w_i its weight. Thus, $\Gamma = \{(a, d, t, w) | a < d, t \leq d - a, w \geq 0\}$.

[3]As mentioned before, not all sets of data items will code graphs. To actually code an undirected graph, a set of data items has to have distinct node names, and have the property that, if $x \in AdjList(y)$ then also $y \in AdjList(x)$.

[4]The presence of a label in the data item, depends on the particular graph problem. For the vertex cover and independent set problems labels are not needed. However, labels might be useful in encoding additional information. For example, if instances are k-partite graphs, the label might encode which partite set a vertex is a member of. For some network optimization problems each node of the network might have a different "label", depending on whether the node is an end-host, or AS router, or a Backbone router. For the Steiner tree problem the vertices of the graph are partitioned into two sets, Steiner and required nodes, and labels will be needed to identify whether the node is a Steiner or a required node.

In fact, we can put pretty much any search or optimization problem in the above framework. Any solution S to an instance I can be described as an array of elements from Σ indexed by some set $D(I)$, $S \in \Sigma^{D(I)}$. Assume we give the algorithm access to the information $\mathit{LocalInfo}(d, I)$ when picking the value of coordinate $d \in D(I)$. Then we can set $\Gamma = \{(d, \mathit{LocalInfo}(d, I)) \mid d \in D(I),\ I \text{ is a valid instance}\}$. Since the union of all $\mathit{LocalInfo}$ is all we are given to solve the problem, we can view I as $\{(d, \mathit{LocalInfo}(d, I)) \mid d \in D(I)\}$.

As in [14], we distinguish between algorithms that order their data items at the beginning, and those that reorder them at each iteration. A FIXED priority algorithm orders the data items at the beginning, and proceeds according to that order. The format for a FIXED priority algorithm is as follows:

```
FIXED PRIORITY ALGORITHM
Input: instance $I \subseteq \Gamma$, $I = \{\gamma_1, \ldots, \gamma_n\}$
Output: solution $S = \{(\gamma_i, \sigma_i) | i = 1, \ldots, n\}$

  1. Initialize an empty partial instance, solution and a counter: $PI \leftarrow \emptyset; S \leftarrow \emptyset$;
     $t \leftarrow 1$
  2. Determine a criterion for ordering data items[5]   $\pi : \Gamma \rightarrow \mathbb{R}^+ \cup \{\infty\}$
  3. Order $I$ according to $\pi(\gamma_i)$, from smallest to largest
  4. Repeat
       • Go through the data items $\gamma_i$ in order
       • (a) In step $t$, observe the $t$'th data item according to $\pi$, let that be $\gamma_{i_t}$
         (b) Make an irrevocable decision $\sigma_{i_t} \in \Sigma$, based only on currently observed
             data items (i.e., the $t$ smallest under the priority function $\pi$)
```

[5]We could instead use a more general notion, where π is a total ordering of Γ, or use a function $\pi : \Gamma \rightarrow \mathbb{R}^l$, for some fixed l. Because we use only finite sets of instances in our lower bounds, all of our lower bounds also hold for this more general class. Our upper bounds use orderings based on real-valued priority functions as given here.

```
        (c) Update the partial solution: $S \leftarrow S \cup \{(\gamma_{i_t}, \sigma_{i_t})\}$
      • Increment counter $t \leftarrow t+1$; Go on to the next data item $\gamma_{i_{t+1}}$
    Until (decisions are made for all data items)
5. Output $S = \{(\gamma_i, \sigma_i) | 1 \leq i \leq n\}$.
```

ADAPTIVE priority algorithms, on the other hand, have the power to reorder the remaining decision items during the execution, and clearly can simulate the simpler FIXED priority algorithms.

ADAPTIVE PRIORITY ALGORITHM

```
Input: instance $I \subseteq \Gamma$, $I = \{\gamma_1, \ldots, \gamma_n\}$
Output: solution vector $S = \{(\gamma_i, \sigma_i) | 1 \leq i \leq n\}$
1. Initialize the set of unseen data points $U$ to $I$, an empty partial instance $PI$ and
   solution $S$, and a counter $t$ to 1: $U \leftarrow I, PI \leftarrow \emptyset, S \leftarrow \emptyset, t \leftarrow 1$
2. Repeat
      • Based only on the previously observed data items $PI$, determine an ordering function
        $\pi_t : \Gamma \rightarrow \mathbb{R}^+ \cup \{\infty\}$
      • Order $\gamma \in U$ according to $\pi_t(\gamma)$
      • Observe the first unseen data item $\gamma_t \in U$ according to the order; and add
        it to the partial instance, $PI \leftarrow PI \cup \{\gamma_t\}$
      • Based only on $PI$, $S$, and $\gamma_t$, make an irrevocable decision $\sigma_t$ and add
        $(\gamma_t, \sigma_t)$ to the partial solution $S$, $S \leftarrow S \cup \{(\gamma_t, \sigma_t)\}$
      • Remove the processed data point $\gamma_t$ from $U$, and increment $t$
   Until (decisions are made for all data items, $U = \emptyset$)
3. Output $S$
```

The current decision made depends in an arbitrary way on the data points seen so far. The algorithm also has an implicit knowledge about the unseen data points: no unseen point has a smaller value of the priority function π_t than γ_t.

[14] also define two other restricted models: GREEDY and MEMORYLESS priority algorithms. They define a GREEDY priority algorithm as follows: "*A greedy algorithm makes its irrevocable decision so that the objective function is optimized as if the input currently being considered is the final input.*" MEMORYLESS priority algorithms were also defined for facility location and set cover problems in [7]. In the context of scheduling problems, where any set of jobs constitutes a valid instance, a GREEDY priority algorithm would schedule a job provided the job does not conflict with the current schedule. This concept of GREEDY algorithms does not seem to be well-defined for arbitrary priority models, in particular graph models, where not every set of data items constitutes a valid instance. Note that, when a priority algorithm for a scheduling problem has observed a partial instance, the algorithm learns nothing about the remaining instance, except that the jobs have priorities higher than the ones already observed. This is not the case for priority algorithms for graph problems. In the latter case a priority algorithm would know that some data items are not present in the remaining instance. For example consider priority algorithms for graph coloring in the node model. If the algorithm has observed one node, let that be a with its adjacency list, denote it as $adj(a)$, then the algorithm knows that every valid instance has to be consistent with the current partial instance, and hence must have node a present in the adjacency list of each node in $adj(a)$. [11] and [6] study GREEDY priority algorithms and prove lower bounds on approximation ratio achieved by such algorithms for various optimization problems.

We formalize the notion of MEMORYLESS algorithms in Sect. 2.5 and show a separation between the class of MEMORYLESS algorithms and ADAPTIVE priority algorithms.

2.3 A General Lower Bound Technique

In this section, we give a characterization of the best approximation ratio achievable by a (deterministic) ADAPTIVE priority algorithm, in terms of a combinatorial game. The techniques used in this section are borrowed from competitive analysis of on-line algorithms and can be viewed as a completeness theorem for the style of lower bound results used in [14].

Let Π be a maximization problem, with objective function μ. Let Σ, and Γ be a priority model for Π. Let T be a finite collection of instances of Π. The ADAPTIVE priority game for T and a fixed ratio $\rho \geq 1$ between two players a Solver and an Adversary is as follows:

Adaptive Priority Game (Solver, Adversary, ratio ρ):

1. Initialize an empty partial instance PI and a partial solution PS. The Adversary picks any subset $\Gamma_1 \subseteq \Gamma$
2. Repeat until ($\Gamma_t = \emptyset$)
   begin; (Round t)
   (a) The Solver picks $\gamma_t \in \Gamma_t$, and $\sigma_t \in \Sigma$
   (b) γ_t is added to PI, and deleted from Γ_t. (γ_t, σ_t) is added to PS
   (c) The Adversary replaces Γ_t with a subset $\Gamma_{t+1} \subseteq \Gamma_t$

   end; (Round t)
3. In the endgame, the Adversary presents a solution S for PI
4. The Solver wins if one of the following three events occurs
   - if $PI \notin T$
   - $PI \in T$, but S is not a valid solution

- $PI \in T$, and S and PS are valid solutions to PI, and $\frac{\mu(S)}{\mu(PS)} \leq \rho$ (when Π is a maximization problem[6])

Otherwise the Adversary wins.

Lemma 1. *There is a winning strategy for the Solver in the Adaptive Priority game if and only if there is an ADAPTIVE priority algorithm that achieves an approximation ratio of ρ on every instance of Π in T.*

Note that the Lemma does not trivially hold. There is a difference between the actions of the Solver in the adaptive combinatorial game and the structure of an adaptive priority algorithm. At each round the Solver selects a data item from a finite set, and makes a decision for it. In contrast the priority algorithm has to define an ordering on all data items, without looking at the instance, and then has to make a decision for the data item with highest priority value. Hence the algorithm cannot directly act as the Adversary in the adaptive priority game.

Also note that existence of a priority algorithm for a problem Π does not translate to existence of an efficient algorithm. The reason is that we do not place complexity restrictions on the priority functions used by the priority algorithm to order the data items. The only restriction applied to the priority function π_t used during the t-th iteration of an ADAPTIVE priority algorithm is information theoretic. Observe that $\pi_t : \Gamma \rightarrow \mathbb{R}^+ \cup \{\infty\}$, as defined, can only depend on the previously observed data items PI but not on future items. The priority function does not have to be polynomially time computable function or FNP, in fact it could be any non-computable function, obeying the information-theoretic restrictions imposed by the model. However, in all upper bounds here, the algorithm is also polynomial-time and usually near linear-time. Furthermore the lower bounds results in this paper hold against any algorithm using a priority function with the imposed information-theoretic restrictions.

- *Assume there is an ADAPTIVE priority algorithm achieving approximation ratio*

[6]When Π is a minimization problem the ratio is $\frac{\mu(PS)}{\mu(S)} \leq \rho$.

ρ on all instances of T.

The Solver uses the following strategy. The strategy maintains the invariant that after the first t rounds of the game, on any instance $I \in T$ with $PI \subseteq I$ and $I - PI \subseteq \Gamma_{t+1}$, the first t data items seen and decisions made by the priority algorithm are equal to the moves $\gamma_1, \ldots, \gamma_t$ and $\sigma_1, \ldots, \sigma_t$ for the Solver player in the game, and so that there is at least one such I. In round $t + 1$, the Solver simulates the priority algorithm on the partial instance PI, and obtains an ordering π_{t+1} of Γ_{t+1}. She chooses the first element under π_{t+1}, γ_{t+1}, as her move, and simulates the algorithm on the additional data item γ_{t+1} to get the corresponding decision σ_{t+1}. The Adversary chooses $\Gamma_{t+2} \subseteq \Gamma_{t+1}$, so for any I with $PI \cup \gamma_{t+1} \subseteq I$ and $I - PI - \gamma_{t+1} \subseteq \Gamma_{t+2}$, the first t seen data items will be $\gamma_1, \ldots, \gamma_t$ by the induction hypothesis, and the π_{t+1}-first unseen one at step $t+1$ will be γ_{t+1}, so the algorithm will view γ_{t+1} next, and by definition of the strategy, choose σ_{t+1} as the decision. So the invariant is maintained until the endgame. In the endgame, the Adversary presents a solution S, and the Solver will output a solution PS. Since the algorithm achieves an approximation ratio ρ, then $\rho \cdot \mu(PS) \geq \mu(S)$.

- *For the converse, assume that the Solver has a winning strategy for the the game for an approximation ratio ρ.* We describe an ADAPTIVE priority algorithm that ensures approximation ratio ρ on all instances in T.

  Let $I \in T$ be the input. We maintain the invariant that there is a game position so that the first t items seen by our algorithm on I are from the first t moves of the Solver; and the first t decisions are from the first t moves of the Solver, and $I - PI \subseteq \Gamma_t$. For the $t+1$'st iteration the algorithm now orders the possible data items in Γ_t as follows: It considers the above run of the game. It then sets the next move $\Gamma_{t+1} = \Delta_1$ of the Adversary to be the set Δ_1 of all data items $\gamma \in \Gamma_t - \gamma_t$ so that there is some $I' \in T$ with $PI \cup \{\gamma_t\} \subseteq I'$, $I' - PI \subseteq \Gamma_t$ and $\gamma \in I'$. (Since I is such an I', we know $I - PI - \gamma_t \subseteq \Delta_1$.) Let δ_i be the

Solver's data item response to the Adversary choosing $\Gamma_{t+i} = \Delta_i$. Then let Δ_{i+1} be the subset of Δ_i of all data items $\gamma \in \Delta_i - \delta_i$ so that there is some $I' \in T$ with $PI \cup \{\gamma_t\} \subseteq I'$, $I' - PI - \gamma_t \subseteq \Delta_i - \delta_i$ and $\gamma \in I'$. Repeat until Δ_i is empty. Let j be the maximal j' so that $I - PI - \gamma_t \subseteq \Delta_{j'}$. Since the condition is true for $j' = 1$, such a j must exist.

The algorithm uses the priority function π_{t+1} that gives δ_j priority j, and all other elements of Γ infinite priority. We claim the next data item the algorithm views is δ_j. Since $I - PI - \gamma_t \subseteq \Delta_j$, no smaller priority $\delta_{j'}$ can be in U. On the other hand, if $\delta_j \notin U$, $U \subseteq \Delta_j - \delta_j$ so all elements of U are in Δ_{j+1}, which would contradict maximality. The algorithm then simulates the move in the previous round of the Adversary setting $\Gamma_{t+1} = \Delta_j$. By definition, the Solver responds with $\gamma_{t+1} = \delta_j$, the same data item as the algorithm, and same decision σ_{t+1}. The algorithm uses σ_{t+1} as its next decision, maintaining the invariant.

When U is empty, the algorithm outputs the solution. To see that this is within ρ of optimal, simulate the Adversary moving to the endgame, and choosing I and an optimal solution S' for I as its endgame moves. The Solver must respond with a solution S extending PS; but, since we've seen the entire output, $S = PS$, so the Solver is forced to provide the same output as our algorithm. Since the Solver wins the game for the fixed ratio ρ, then $\frac{\mu(S')}{\mu(S)} \leq \rho$, so our algorithm achieves an approximation ratio of ρ.

Corollary 2. *If there is a strategy for the Adversary, in the game defined above, that guarantees a payoff of at most ρ, then there is no ADAPTIVE priority algorithm that achieves an approximation ratio better than ρ.*

We can similarly characterize the FIXED priority model by replacing steps 1 and 2(a) as follows. The rest of the game is the same.

1′ Initialize an empty partial instance PI and a partial solution PS. The Solver picks a total ordering $<$ on Γ. The Adversary picks any subset $\Gamma_1 \subseteq \Gamma$.

$\mathbf{2(a)'}$ Let $\gamma_t \in \Gamma_t$ be the $<$-first element of Γ_t. The Solver picks $\sigma_t \in \Sigma$.

Lemma 3. *There is a winning strategy for the Solver in the game if and only if there is a FIXED priority algorithm that achieves an approximation ratio ρ on every instance of Π in T.*

2.4 Results for Graph Problems

In this section we present our results for FIXED and ADAPTIVE priority algorithms. The proofs of lower bounds here and in Sect. 2.5 resemble derivation of integrality gaps for given LP formulation, in that easy instances are used to establish bounds on the approximation ratio (See [48] for examples). Our results are used to evaluate the power and weaknesses of the priority algorithms framework and not to establish hardness of approximation results for the particular problem in general.

2.4.1 Shortest Paths

FIXED priority algorithms are simpler and ADAPTIVE priority algorithms can simulate them. We want to show that the two priority models ADAPTIVE and FIXED, are not equivalent in power. We define the following graph optimization problem.

Definition 2 (SHORTEST PATH PROBLEM). *Given a directed graph $G = (V, G)$ and two nodes $s \in V$ and $t \in V$, find a directed tree of edges, rooted at s. The objective function is to minimize the combined weight of the edges on the path from s to t.*

We consider the SHORTEST PATH problem in the edge model. The edge model is natural for graph problems where the solution is a path or a tree, because the solution labels the edges, either in or out. Furthermore, the standard algorithm for this problem (Dijkstra's) is a priority algorithm, in the edge model. A minor point is that, if we consider the problem in the edge model, then we could define the problem for multi-graph instances.

The data items are the edges in the graph, represented as a triple (u, v, w), where the edge goes from u to v and has a weight w. The set of options is $\Sigma = \{accept, reject\}$. A valid instance of the problem is a graph, represented as a set of edges, in which there is at least one path from node s to t. An alternative definition of the SHORTEST PATH problem would insist on the edges selected to form a path, rather than a tree. However, most standard algorithms construct a single source shortest paths tree, rather than a single path. In fact, not only is constructing a tree a more general version of the problem; it is not difficult to show that no priority algorithm can guarantee a path. (A brief justification why no priority algorithm can guarantee a path is given in a paragraph, after the end of the proof of Theorem 4.)

The well-known Dijkstra algorithm, which belongs to the class of ADAPTIVE priority algorithms, solves this problem exactly.

Theorem 4. *No FIXED priority algorithm can solve the* SHORTEST PATH *problem with any constant approximation ratio* ρ.

Proof: We show an Adversary strategy for the FIXED priority game for any ρ. Let $k \geq 2\rho$. Let T be the set of directed graphs on four vertices s, t, a, b with edge weights either k or 1, so that t is reachable from s. The Adversary selects the set Γ_1, as shown on Figure 2.1. For example, u stands for the edge from s to a, with weight k. Note that the parallel edges in the figure are just possible data items; the instance graph will be guaranteed to be a simple graph. The next move is by the Solver. She must assign distinct priorities to all edges, prior to making any decisions, and this order cannot change. Thus one of the edges y and z must appear first in the order. Since the set of data items is symmetrical, we assume, without loss of generality, that y appears before z in this order. The Adversary then removes edges v and w, restricting the remaining set of data items to $\Gamma_2 = \{x, y, z, u\}$. The Adversary's strategy is to wait until the Solver considers edge y before deleting any other items, and applies the following strategy after he observes Solver's decision on y:

1. If the Solver decides to reject y, then the Adversary removes z from the remain-

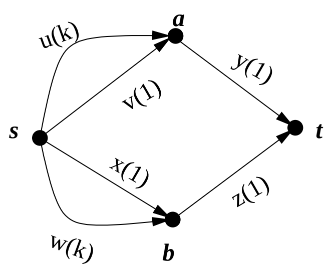

Figure 2.1: Adversary selects $\Gamma_1 = \{x, y, z, u, v, w\}$.

ing set of data items. The Adversary does not subsequently remove any other data items. Thus, the instance will be $I = \{v, x, y\}$. The Adversary outputs a solution $S = \{y, v\}$, while the Solver's solution $PS \subseteq \{v, x\}$ cannot contain any path from s to t.

2. If the Solver decides to accept y, then the Adversary never deletes any data items, making $I = \{u, x, y, z\}$. In the end game, the Adversary presents solution $S = \{x, z\}$, with cost 2. If the Solver picks edge z, then the Solver failed to satisfy the solution constraints, since no sub-graph with both y and z can be a directed rooted tree. Otherwise, $PS \subseteq \{u, x, y\}$ and thus the cost of the Solver is at least $k+1$. The approximation ratio is: $\frac{k+1}{2} > \rho$, so the Solver loses.

□

Note that almost the same Adversary strategy can force any priority algorithm to fail to produce a solution which is a simple path. The Adversary presents $\Gamma_1 = \{x, z, v, y\}$. If the algorithm accepts v, then the Adversary presents $\Gamma_2 = \{x, z\}$. If v was rejected then the Adversary presents $\Gamma_2 = \{y\}$. Either way the Solver will fail to output a valid solution. The other cases are symmetric.

We conclude that the two classes of algorithms FIXED and ADAPTIVE priority are not equivalent in power. Dijkstra's algorithm can solve the above problem exactly and belongs to the class of ADAPTIVE priority algorithms.

Dijkstra's algorithm, however, does not work on graphs with negative weight edges. Is dynamic programming necessary for this problem? Perhaps there exists an ADAPTIVE priority algorithm which can solve the Single Source Shortest Paths problem on graphs with negative weight edges, but no negative weight cycles?

Theorem 5. *No ADAPTIVE priority algorithm can solve the* SHORTEST PATH *problem for graphs with weight function* $w_e : (E) \to \mathbb{R}$, *allowing negative weights, but no negative weight cycles.*

Proof. We view the problem in the edge model. The Adversary chooses Γ_1 to be the set of directed edges shown on Figure 2.2 and $k > \rho$. Although the set Γ_1 defines a graph with a negative weight cycle, the final instances T are subgraphs of the graph shown in Figure 2.2, with a path from s to t, and no negative weight cycle. The set of decision options is $\Sigma = \{accepted, rejected\}$. We will refer to the edge from v to u of weight $-k$ as $d(-k)$, or just d, for short. The Adversary and the Solver play the combinatorial

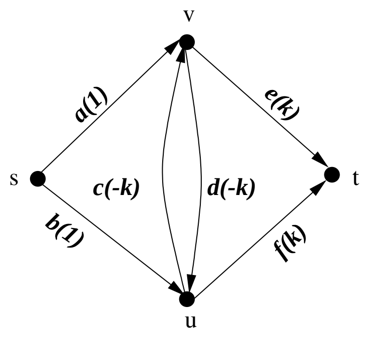

Figure 2.2: The set of edges initially selected by the Adversary Γ_1=$\{a, b, c, d, e, f\}$

game defined in Sect. 2.3. The Adversary observes the first data item selected and the decision committed by the Solver and uses the following strategy.

1. The first data item chosen by the Solver is a.

   - If the Solver decides to accept a, then the Adversary presents the instance

$$I = \{a, b, c, e, f\}$$

and a solution $S_{adv} = \{b, c, e\}$ of cost 1. The Solver is forced to either take $S_{sol} = \{a, b, e\}$ or $S_{sol} = \{a, b, f\}$ and loses the game because the approximation ratio is $\frac{\mu(S_{sol})}{\mu(S_{adv})} = \frac{k+1}{1} > \rho$.

- If the Solver decides to reject a, then the Adversary presents the instance $I = \{a, e\}$ and a solution $S_{adv} = \{a, e\}$, while the Solver would fail to construct a path and hence loses the game.

2. The first data item chosen by the Solver is c.

   - If the Solver decides to accept c, then the Adversary presents the instance $I = \{a, c, e\}$ and a solution $S_{adv} = \{a, e\}$ and wins the game, because the Solver would fail to construct a rooted tree.
   - If the Solver decides to reject c, then the Adversary presents the instance $I = \{a, b, c, e\}$ and a $S_{adv} = \{b, c, e\}$ of cost 1. The only solution left for the Solver is $S_{sol} = \{a, e\}$ of cost $k + 1$; so she loses the game because the approximation ratio is $\frac{\mu(S_{sol})}{\mu(S_{adv})} = \frac{k+1}{1} > \rho$.

3. The first data item chosen by the Solver is e.

   - If the Solver decides to accept e, then the Adversary wins the game because, he presents the instance $I = \{b, e, f\}$ and a solution $S_{adv} = \{b, f\}$, while the Solver would fail to construct a rooted tree.
   - If the Solver decides to reject e, then the Adversary presents the instance $I = \{a, e\}$ and a solution $S_{adv} = \{a, e\}$, while the Solver failed to construct a rooted tree, and hence lost the game.

The cases when the Solver considers edges b, d, or f first, have the same analysis as the cases already discussed and will be omitted. The Adversary can set k arbitrarily large, thus he can win the game for any approximation ratio ρ. $\square$

This result shows a separation between priority algorithms and dynamic programming algorithms for shortest path problems. A similar separation was shown by [14] for interval scheduling on a single machine with arbitrary profits.

2.4.2 The Weighted Vertex Cover Problem

In this section we examine the performance of ADAPTIVE priority algorithms on the weighted vertex cover problem (WVC). An instance of the problem is a graph $G = (V, E)$ with a positive weight function w on the set of vertices $w : V \to \mathbb{R}^+$. The problem is to find a subset of the vertices $C \subseteq V$ subject to the constraints that if $(u, v) \in E$, then either $u \in C$, or $v \in C$. The objective is to minimize the combined weight of the vertices in the cover set C, $\sum_{u \in C} w(u)$.

WVC is an **NP**-hard problem and unless $P = NP$, no polynomial time algorithm can approximate it with ratio better than 1.3606,[22]. The well known 2-approximation algorithm [30], fits our ADAPTIVE priority model, and we show that no ADAPTIVE priority algorithm can achieve an approximation ratio better than 2. On the other hand no algorithm is known (greedy or otherwise) to achieve an approximation ratio better than 2 for the weighted vertex cover problem in arbitrary graphs. For graphs with bounded degree, using semidefinite programming relaxation [26] achieves an approximation ratio of $2 - \frac{2 \ln \ln \Delta}{\ln \Delta}$, where Δ is the maximum degree of the graph.

Next we define the notion of an unseen node which we use in the lower bound proofs in the remaining sections of the paper.

Definition 3. (unseen node) *Let t be the round of the game between the Solver and the Adversary. Let* PI *be the instance presented by the Adversary in the endgame and let v be a vertex, such that $v \in PI$.*

- *For graph problems in the* **node model** *let $\mathbf{PI_t}$ denote the set of data items seen by the Solver up until time t. Each data item is represented as its name and adjacency list: $(u, adj(u))$. We call v unseen node at time $t + 1$ if $(v, adj(v)) \notin PI_t$.*

- *For graph problems in the* **edge model** *let* $\mathrm{PI_t}$ *denote the set of edges that have been considered by the Solver up until the* t*-th iteration of the game. A node* u *is unseen at time* $t+1$ *if* $u \notin V(PI_t)$.

Intuitively a vertex (node) is unseen at time $t+1$ if it belongs to the final instance and during iterations $1, 2, \dots, t$, the Solver hasn't made a decision about it (for node model), or if it is not being touched by an edge for which the Solver has committed to a decision (for edge model).

We consider the vertex cover problem in the node model. The data items are nodes, with their name, weight, and adjacency list. The set of options is $\Sigma = \{accept, reject\}$, meaning the vertex is added to the vertex cover or thrown away.

Theorem 6. *No ADAPTIVE priority algorithm can achieve an approximation ratio better than* 2 *for the weighted vertex cover problem.*

Proof. For any $\rho < 2$, we show a winning strategy for the Adversary in the game defined in Sect. 2.3. For a suitably large n, the Adversary picks a complete bipartite graph $K_{n,n}$ and sets T to be the set of instances with this underlying graph, where node weights are either 1 or n^2. Since the underlying graph is fixed, Γ contains two data items for each node, varying only in the node weight.

Each time the Solver selects a data item corresponding to a node v, the Adversary deletes the other such item (avoiding inconsistency). The Adversary otherwise does not delete any items until one of the following three events occurs:

- **Event 1:** The Solver accepts a node v with weight n^2.
- **Event 2:** The Solver rejects a node v (of any weight).
- **Event 3:** The Solver accepts $n-1$ nodes of weight 1 from either side of the bipartite graph.

Eventually, one of these three events must occur. If **Event 1** occurs first, then the Adversary fixes the weights of all nodes on the opposite side to 1, by deleting all data items giving weight n^2. This is possible, since previously the Solver has only accepted nodes of weight 1, so no data item giving a node weight 1 has been deleted. Similarly, for each node on the same side with two possible weights, Adversary deletes the data item of weight 1. This fixes the instance. Eventually, the Solver will consider all remaining data items, and output a solution PS with $v \in PS$, so PS has cost at least n^2. The Adversary outputs a solution S consisting of all nodes on the other side, which has cost n, winning if $\rho < n^2/n = n$.

If **Event 2** occurs, the Adversary fixes the weights of all unseen nodes on the opposite side of v to n^2 and the weights of remaining nodes (those not yet considered) on the same side to 1 (by deleting the data items of other weights.) Since neither Events 1 or 3 have previously occurred, it is feasible to define an instance with all nodes on the same side have value 1, and there are at least 2 unseen nodes on the opposite side. Eventually the Solver outputs a vertex cover PS with $v \notin PS$. Hence, all nodes on the opposite side must be in PS, for a total cost of at least $2n^2$. The Adversary outputs all nodes on the same side of v, for a cost of at most $n^2 + n - 1$. The Adversary wins if $\rho < \frac{2n^2}{n^2+n-1} = 2 - o(1)$.

If **Event 3** occurs first, the Solver has committed to all but one vertex on one side, say A, of the bipartite graph. Then the Adversary fixes the weight of the last unseen vertex in A to n^2 (by deleting the data item giving it value 1) and unseen nodes on the other side are set to weight 1.

The Solver outputs a set either containing all of A and hence having weight at least n^2, or containing all but one node of A and all nodes of the other side B, giving a total weight of $2n-1$. The Adversary presents as a solution all nodes of side B, winning if $\rho < \frac{2n-1}{n} = 2 - o(1)$.

Thus, for any $\rho < 2$, the above is a winning strategy for the Adversary for a suitably large value of n. □

The class of instances $K_{n,n}$ can be solved easily. However, what Theorem 6 shows is that a large class of greedy algorithms cannot approximate the WVC problem with approximation ratio better than 2. This bound is tight, because the known greedy heuristic achieves an approximation ratio 2, and thus is optimal in the class of adaptive priority algorithms.

It was important to our bound that nodes had weights. Boyar and Larsen ([11]) consider the unweighted version and prove a lower bound of $\frac{4}{3}$ for any priority algorithm.

2.4.3 The Metric Steiner Tree Problem

We examine the performance of the ADAPTIVE priority algorithms on the metric Steiner tree problem. The instance of the problem is a graph $G = (V, E)$, with the vertex set partitioned into two disjoint subsets, *required* and *Steiner*. There is a weight function w on the edges of the graph $w : V \times V \to \mathbb{R}^+$, such that $w(u, v) \leq w(u, x) + w(x, v), \forall u, v, x \in V$. The problem is to find a minimum cost tree, spanning the required vertices (but may contain any number of Steiner nodes). The metric property implies that any two vertices within the same connected component are connected by an edge. Furthermore if G has more than one connected components then the cost of any Steiner tree will be infinity. Hence, we assume that each valid instance of the problem is a complete graph, whose edge weights obey the triangle inequality.

Note the difference between the minimum spanning tree and the Steiner tree problems. For the former, the algorithm is presented with an undirected weighted graph, and must produce a tree, of minimal cost, which spans all vertices of the graph. Kruskal's algorithm solves the problem exactly, by considering the edges of the graph one at a time in non-decreasing order. For each edge, the algorithm decides to add the edge to the solution, if the solution remains a forest, otherwise the edge is rejected. But when we consider the Steiner tree problem, the solution needs to span *only the required nodes* of the graph, and it is not obvious how to make use of the greedy strategy above, because the Steiner nodes are optional and may or may not be included in the final solution.

We are interested in the metric version, where the edge weights obey the triangle inequality. The reason is that many greedy algorithms for the Steiner tree problem first reduce the instance to the metric version, by precomputing the shortest path distances. Thus, it is reasonable to only consider priority algorithms for the metric version, i.e., where this preprocessing has occurred. The standard 2-approximation algorithm for the Steiner tree problem discovered independently by [33] and [41] belongs to the class of FIXED priority greedy algorithms. In the restricted case when the edges of the graph have weights either 1 or 2, known as the STEINER$(1,2)$ problem, Bern and Plassmann ([9]) proved that *the average distance heuristic* ([49], [42]), is a $\frac{4}{3}$-approximation. The *average distance heuristic*, however, does not seem to fit our priority model. To our knowledge the best known polynomial-time approximation algorithm for the Steiner tree problem is [43] with approximation ratio of $1 + \frac{\ln 3}{2} \approx 1.55$ for general graphs, and approximation ratio of ≈ 1.28 for STEINER$(1,2)$ problem. [43]'s algorithm does not fit the framework of priority algorithms.

On the negative side in [9] Bern and Plassmann also showed that STEINER$(1,2)$ problem is MAX SNP-hard, implying that it is unlikely that the metric Steiner tree problem with edge weights in $[1, 2]$ has a polynomial-time approximation scheme. [18] proved that, unless $P = NP$, Steiner tree is hard to approximate within a factor of $(1 + \epsilon)$, for small $\epsilon > 0$.

Our lower bounds are for an intermediate class of Steiner problems, where edge weights are in the interval $[1, 2]$. This very local restriction implies the metric property, which helps the adversary argument. To show that we cannot get a tight bound of 2 using this restriction, we give a new priority algorithm for this restricted class, achieving an approximation ratio of 1.8.

The edge model is natural for graph problem where the solution is a path or a tree and we consider the metric Steiner tree problem in the edge model.

Theorem 7. *No ADAPTIVE priority algorithm in the edge model can achieve an approximation ratio better than* 1.2 *for the metric Steiner tree problem, even when edge weights are restricted to the interval* $[1, 2]$.

Proof. We consider the metric Steiner tree problem in the edge model. The data items are edges in the graph represented as a 5-tuple $(v_1, t_1, v_2, t_2, w(v_1, v_2))$, where v_1, v_2 are the names of the nodes; t_1, and t_2 are their type, $t_1, t_2 \in \{\mathbf{r}, \mathrm{s}\}$ ($\mathbf{r}$ for required, s for Steiner), and $w(v_1, v_2)$ is the weight of the edge. The set of decision options is $\Sigma = \{accepted, rejected\}$; S_{sol}, S_{adv} are the solutions chosen by the Solver and the Adversary, respectively.

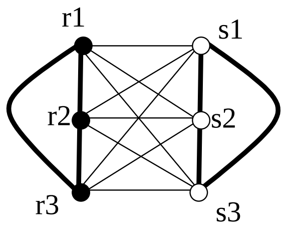

Figure 2.3: K_6 with three required and three Steiner nodes

The set of instances selected by the Adversary are graphs shown on Figure 2.3, where the vertices of the graph $V = R \cup S$. The set of required nodes is $R = \{r_1, r_2, r_3\}$ and $S = \{s_1, s_2, s_3\}$ is the set of Steiner nodes. The weight function defined on the set of edges in Γ_1 is specified as follows. An edge between two required nodes, or between two Steiner nodes has a fixed weight of 2. An edge between a Steiner and a required node can have weight 1 or $(2 - \epsilon)$, where the value of $\epsilon \in (0, 1)$ will be determined later. Therefore, in the initial set Γ_1 each edge between Steiner and required nodes will have two data items, one with weight 1 and the other with weight $2 - \epsilon$, the other edges correspond to a single data item. The set of instances T is a finite set of graphs K_6, shown on Figure 2.3, with edges from Γ_1. The Solver must chose a data item from Γ_1 and a decision for it. The goal of the Adversary is to make the decisions made by the Solver unfavorable and his strategy is described below.

Remark 8. We use shorter notation for data items when we describe the solutions of the Adversary and the Solver. Note that the set of required nodes $R = \{r_1, r_2, r_3\}$ is disjoint from the set of Steiner nodes $S = \{s_1, s_2, s_3\}$, so we can safely omit the types $\mathbf{r}, \mathrm{s}$. Instead of the 5-tuple notation $(v_1, t_1, v_2, t_2, w(v_1, v_2))$, for brevity we will use a triple. For example, a data item $(s_j, \mathbf{s}, r_i, \mathbf{r}, 1)$ will be represented by the triple $(s_j, r_i, 1)$, when we describe the solutions S_{adv} and S_{sol}.

- Suppose the first data item selected by the Solver is and edge between a Steiner and a required node of weight 1, let it be $(s_i, \mathbf{s}, r_j, \mathbf{r}, 1)$.

  – Case 1: If the Solver decides to accept it, then the Adversary makes s_j as far away from the remaining required nodes as possible, by restricting Γ_2 as follows:

    * He leaves in Γ_2 edges between s_j and the other required nodes of weight $(2 - \epsilon)$ only.
    * Then he chooses another Steiner node, say s_k, and removes from Γ_2 edges between s_k and all required nodes of weight 2–ϵ, i.e., only edges of weight 1 between s_k and the required nodes remain in Γ_2.

  The Adversary selects a Steiner tree $S_{adv} = \{(s_k, r_1, 1)(s_k, r_2, 1)(s_k, r_3, 1)\}$ of cost 3. The Solver must select a valid solution, which includes edge $(s_j, \mathbf{s}, r_i, \mathbf{r}, 1)$. The best choices[7] for her are either

$$S_{sol} = \{(s_j, r_i, 1)(s_k, r_1, 1)(s_k, r_2, 1)(s_k, r_3, 1)\}$$

  of cost 4, or

$$S_{sol} = \{(s_j, r_i, 1)(s_j, r_k, 2 - \epsilon)(s_j, r_l, 2 - \epsilon)\},$$

  where $\{r_i, r_k, r_l\} = R$, of cost $1 + 2(2 - \epsilon)$. The approximation ratio is either $\frac{\mu(S_{sol})}{\mu(S_{adv})} = \frac{4}{3}$ or $\frac{\mu(S_{sol})}{\mu(S_{adv})} = \frac{5-2\epsilon}{3}$.

  – Case 2: If the Solver decides to reject it, then the Adversary does the opposite.

[7]Note that, there are other valid solutions. For example $\{(s_j, r_i, 1), (r_i, r_j, 2), (r_j, r_k, 2)\}$ of cost 5, but the approximation ratio is even worse.

  - $*$ He makes the Steiner node s_j close to the remaining required nodes, by restricting Γ_2 to edges between s_j and the remaining required nodes of weight 1 only.
  - $*$ He further restricts Γ_2 by removing edges between the other two Steiner nodes and the required nodes of weight 1.

  The Adversary chooses solution $S_{adv} = \{(s_j, r_1, 1)(s_j, r_2, 1)(s_j, r_3, 1)\}$ of cost 3, while the best the Solver can do is to select a spanning tree of cost 4. One possible solution for the Solver is two edges of weight 1 between s_j and the other two required nodes, and one edge between two required nodes of weight 2, re-connecting r_j. The other solution is two edges between required nodes, each of weight 2. (There are two other solutions of cost $6 - 3\epsilon$, but we will ignore them because we choose $\epsilon \leq \frac{2}{3}$ as it will be seen later at the end of the proof.) The approximation ratio is $\frac{\mu(S_{sol})}{\mu(S_{adv})} = \frac{4}{3}$.

- The first data item selected by the Solver is an edge between a Steiner and a required node of weight $2 - \epsilon$, let it be $(s_j, \mathbf{s}, r_i, \mathbf{r}, 2-\epsilon)$.

  – Case 3: If the Solver decides to accept it, then the Adversary defines Γ_2 as follows:

  - $*$ He leaves in Γ_2 edges between s_j and the other required nodes of weight $(2 - \epsilon)$ only.
  - $*$ Then he selects another Steiner node, say s_k, and removes from Γ_2 all edges between s_k and all required nodes of weight $(2 - \epsilon)$.

  The Adversary chooses $S_{adv} = \{(s_k, r_1, 1)(s_k, r_2, 1)(s_k, r_3, 1)\}$ of cost 3, while the best the Solver can do[8] is select a Steiner tree of cost $3 + 2 - \epsilon = 5 - \epsilon$. The approximation ratio is $\frac{\mu(S_{sol})}{\mu(S_{adv})} = \frac{5-\epsilon}{3}$.

  – Case 4: If the Solver decides to reject it, then the Adversary restricts Γ_2 to:

[8]Again other choices are possible, for example a solution $\{(s_j, r_1, 2-\epsilon)(s_j, r_2, 2-\epsilon)(s_j, r_2, 2-\epsilon)\}$ of cost $3(2-\epsilon)$ and the approximation ratio will not be improved, since we will choose $\epsilon = \frac{2}{2}$.

  * Edges from s_j to the remaining required nodes of weight 1 only.
  * Then only edges between the remaining required and Steiner nodes of weight $(2-\epsilon)$ are left in Γ_2.

  The Adversary outputs a solution $S_{adv} = \{(s_j, r_i, 2-\epsilon)(s_j, r_k, 1)(s_j, r_l, 1)\}$, where $\{r_i, r_k, r_l\} = R$, of cost $2+2-\epsilon = 4-\epsilon$. The best the Solver can do is select a Steiner tree of cost 4 or $3(2-\epsilon)$ and the approximation ratio is $\frac{\mu(S_{sol})}{\mu(S_{adv})} = \frac{4}{4-\epsilon}$ or $\frac{\mu(S_{Sol})}{\mu(S_{Adv})} = \frac{3(2-\epsilon)}{4-\epsilon}$, respectively.

- The first edge observed by the Solver is an edge between two required nodes, say it is $(r_i, \mathbf{r}, r_j, \mathbf{r}, 2)$.

  – Case 5: If the Solver decides to accept it, then the Adversary restricts Γ_2 to edges between Steiner and required nodes of weight 1, only. The Adversary selects a Steiner tree $S_{adv} = \{(s_1, r_1, 1)(s_1, r_2, 1)(s_1, r_3, 1)\}$ of cost 3. The best the Solver can do is select a Steiner tree of cost 4. The approximation ratio is $\frac{\mu(S_{sol})}{\mu(S_{adv})} = \frac{4}{3}$.

  – If the Solver decides to reject it, then the Adversary awaits the next choice of the Solver and uses the same strategy as in Case 1, 2, 3, 4, or 5, respectively, depending on the type of edge chosen by the Solver.

- Suppose the Solver chooses an edge between two Steiner nodes and decides to accept it. The Adversary will fix the weight of all edges between Steiner and required nodes to 1, by removing data items between Steiner and required nodes of weight $2-\epsilon$ from Γ_2. The approximation ratio will become worse, because any valid solution for the Solver will have cost at least 5, while the Adversary chooses a solution $S_{adv} = \{(s_1, r_1, 1), (s_1, r_2, 1), (s_1, r_3, 1)\}$ of cost 3.

  If the Solver rejects all edges between two Steiner nodes, she still has to consider one of the previous cases, eventually, and then the Adversary uses the strategy described above. If the Solver rejects all edges then she will fail to output a solution and the approximation ratio will be infinity.

Considering the cases above, for $\epsilon \in (0,1)$, the best approximation ratio achieved is $\frac{4}{4-\epsilon}$ and $\frac{3(2-\epsilon)}{4-\epsilon}$. The Adversary chooses $\epsilon = 2/3$ and wins the game for any $\rho < \frac{6}{5} = 1.2$. $\square$

Next we present an algorithm achieving an approximation ratio of 1.8 for the metric Steiner tree problem with edge weights restricted to the interval $[1,2]$. (Note that having weights $w \in [1,2]$ implies the metric property.) First we design an algorithm called Contract, and will prove that it guarantees an approximation ratio of 1.75 (c.f. Theorem 9). The algorithm Contract makes decisions (accept or reject) about edges of the instance but it is not obvious whether it is a priority algorithm or not. Then we use Contract to devise another algorithm called Adaptive Contract, which fits the framework of ADAPTIVE priority algorithms. We show that Adaptive Contract achieves an approximation ratio of 1.8 (c.f. Theorem 12).

From here on, when we refer to a metric graph G, we also assume that the weights of the edges are in the interval $[1,2]$. (Recall that the metric property implies that and any two nodes are connected by an edge.) The input to the algorithm are the edges of the graph and each edge is represented as a five tuple: names of the two endpoints, their type Steiner or required, and the weight of the edge. The algorithm runs in two stages. In the first stage we grow a spanning forest by contracting nodes. The contraction operation modifies a graph by replacing a set of nodes, with a single contracted node. The nodes to be removed (contracted) are connected by edges in the original graph and form a connected component. The distance between any uncontracted node u and the new node becomes the shortest distance between u and a node in the contracted node.

Definition 4 (contract operation). *The $contract$ operation takes a metric graph $G = (V,E)$ and a set of nodes $C = \{v_1,\ldots,v_k\}, C \subset V$, and outputs a new metric graph $G' = (V',E')$ with reduced set of vertices $V' = V - C \cup \{c(v_1,\ldots,v_k)\}$, where $c(v_1,\ldots,v_k)$ is a single node representing the nodes $v_1,\ldots,v_k$. $E' = E - E_0 - E_1 \cup E_2$, where $E_0 = \{(u,v) \mid u,v \in C\}$, $E_1 = \{(u,v) \mid u \in V - C,\ v \in C\}$, and $E_2 = \{(u, c(v_1,\ldots,v_k)) \mid \exists\, (u,v_i) \in E_1$ and $w(u,c(v_1,\ldots,v_n)) = \min_{(u,v_i)\in E_1} w(u,v_i)\}$.*

The second stage builds a minimum cost spanning tree on the graph induced by the contracted nodes, which are considered required nodes, and the remaining (uncontracted) required nodes using edges between required nodes only. The following is a high level description of the algorithm, which we modify later to fit the priority algorithm model.

Algorithm **Contract**

1. *Contract required nodes*

   An edge between a Steiner and a required node is called *lightweight edge* if its weight is 1.4 or less. Let the *lightweight degree* of a Steiner node be the number of lightweight edges incident to it.

   Find all Steiner nodes, whose lightweight degree is five or more. Order these Steiner nodes according to their lightweight degree in non-increasing order, and choose the Steiner node first in this order, let that be x. Add all the lightweight edges incident to x to the current forest. Contract x and the required nodes connected to the Steiner node x by the lightweight edges to a single required node. Recompute the lightweight degree of each remaining uncontracted Steiner node and repeat the procedure above until all remaining Steiner nodes are connected to at most four required nodes with lightweight edges.

   We call the graph obtained at the end of this phase *contracted graph*.

2. *Create a spanning tree*

   Each contracted node from the previous phase is considered to be a single required node. Use Kruskal algorithm to build a minimum cost spanning tree on the subgraph induced by the current set of required nodes, by considering edges between two required nodes only.

end (Algorithm **Contract**)

First we argue that the algorithm builds a tree. This follows from the fact that before the algorithm starts the instance is a complete graph on Steiner and required nodes. At the end of the first stage we have contracted nodes, in which at least five required nodes and one Steiner node are connected by a path, and required nodes. The last stage connects contracted nodes, created in the previous stage, and the remaining required nodes by a spanning tree. Obviously the solution connects all required nodes and no cycle is created at any time because no edge between two already connected nodes is accepted.

Theorem 9. *The algorithm Contract is a* 1.75 *approximation algorithm for the metric Steiner tree problem on graphs with weights in the interval* $[1, 2]$.

Proof. We analyze the two phases separately. Lemmas 10 and 11 are needed to bound the cost during the contract required nodes and create a spanning tree phases, respectively.

- Analysis of contract phase.

  Let G be the instance graph. We want to compare an optimal Steiner tree in G to the tree built by our algorithm, by applying the contract operation on the nodes connected by edges chosen by our algorithm. We will use the following lemma repeatedly during the conversion.

  Lemma 10. *Let* $G = (V = R \cup S, E)$ *be any metric graph, where* R, S *are the set of required and Steiner nodes, respectively. Let* $w : E \to [1, 2]$ *be a weight function on the edges, and* $r_1, r_2 \in R$ *be any two required nodes,* τ *be any Steiner tree. There exists a Steiner tree* τ_1 *in* $G_1 = contract(G, r_1, r_2)$ *such that*

  $$Cost(\tau) \geq Cost(\tau_1) + 1. \tag{2.1}$$

  Proof. (Lemma 10) Since we consider metric graphs with edge weights in $[1, 2]$, then in any spanning tree τ, in the path from r_1 to r_2, the weight of any edge in the path is at least 1, and contracting the two nodes will necessarily remove at least one edge, along the path between them, from the Steiner tree. Therefore $Cost(\tau) \geq Cost(\tau_1) + 1$. $\square$

(Lemma 10)

Let k be the number of contractions performed by the algorithm during the first phase. Each contraction removes from the instance at least five required nodes and adds one new node representing the removed nodes as a single contracted node.

Consider a single iteration of this phase. Say the algorithm contracts a Steiner node s and required nodes $r_{1,1}, \ldots, r_{1,t}$, where $t \geq 5$, into one node, by adding edges $(s, r_{1,1}), \ldots, (s, r_{1,t})$ to its solution. The cost incurred by the algorithm during the iteration denoted as $Cost(\mathcal{A}_1)$ is $Cost(\mathcal{A}_1) \leq 1.4t$, because the algorithm adds lightweight edges whose weight is 1.4 or less, only. Let τ be the optimal Steiner tree in G, and τ_1 be the optimal Steiner tree in the contracted graph $G_1 = contract(G, \{r_{1,1}, \ldots, r_{1,t}, s\})$. Then we apply Lemma 10 $(t-1)$ times and derive that $Cost(\tau) \geq Cost(\tau_1) + t - 1 \geq Cost(\tau_1) + \frac{t-1}{1.4t} Cost(\mathcal{A}_1)$.

Note that $\alpha(t) = \frac{t-1}{1.4t},\ t \in [5, \infty)$ is an increasing function on t so the smallest α is obtained when $t = 5$. Therefore $Cost(\tau) \geq Cost(\tau_1) + (\frac{t-1}{1.4t}) Cost(\mathcal{A}_1) \geq Cost(\tau_1) + \frac{4}{7} Cost(\mathcal{A}_1)$, or $Cost(\mathcal{A}_1) \leq \frac{7}{4}(Cost(\tau) - Cost(\tau_1))$.

Let $Cost(\mathcal{A}_\mathcal{C})$ be the total cost of the algorithm incurred during the contract phase, and $G_i = contract(G_{i-1}, \{r_{i,1} \ldots r_{i,t_i}\})$, for $i = 1, \ldots, k$. Then we can show inductively that $Cost(\mathcal{A}_\mathcal{C}) = \sum_{i=1}^{k} Cost(\mathcal{A}_i) = \frac{7}{4}(Cost(\tau) - Cost(\tau_k))$, where τ_k is an optimal spanning tree in the contracted graph G_k.

- Analysis of “Create a spanning tree” phase.

  During the second phase the algorithm builds a minimum cost spanning tree on the required nodes (including contracted nodes during the previous phase, which count as required nodes) and does not use any Steiner nodes. There could be three types of edges remaining in the instance before the algorithm enters the second phase: between two Steiner nodes, between a Steiner and a required node, and between two required nodes. For short denote the three types as edges of type (s, s), (r, s), and (r, r), respectively.

Lemma 11. *Let* $G = (S \cup R, E)$ *be any metric graph with a weight function on edges*
$w : E \to [1, 2]$*, so that the lightweight degree of any Steiner node* $s \in S$ *is smaller than or equal to* 4*. Let* τ *be the minimum cost Steiner tree in* G*. Then there is a spanning tree* τ' *on* R *in* G*, such that* $Cost(\tau') \leq 1.6Cost(\tau)$*.*

Proof. (Lemma11) If τ does not have Steiner nodes, then we simply chose $\tau' = \tau$. If τ does contain Steiner nodes, then we want to convert the graph G to a graph G', where all Steiner nodes used by τ are removed. The basic idea of the conversion process is in each step to remove a set of Steiner nodes that are a connected component in τ containing only Steiner nodes. We delete the Steiner nodes in this connected component, and all the edges incident to them from τ. We reconnect the disconnected required nodes by a path with edges of type (r, r) only. (Recall that the instance is a complete graph and all possible edges exist.)

Let $\mathcal{D}_i$ and $\mathcal{A}_i$ be the edges deleted from and added to the tree τ during the i-th iteration of the conversion, respectively. Consider first $i = 1$. Let S_1 be any connected component of Steiner nodes in τ, and let $|S_1| = m$. Let R_1 be the set of required nodes adjacent to nodes in S_1, and let $|R_1| = n$. The combined cost of the edges $\mathcal{A}_1$, needed to connect the nodes in R_1 by a path is $Cost(\mathcal{A}_1) \leq 2(n - 1)$. The edges deleted from τ used to connect nodes in S_1 and R_1 in a single connected component are of two types (s, s) and (r, s). Since there are m Steiner nodes and they form a connected component, then the cost of the edges of type (s, s) is greater than or equal to $m-1$, because $m-1$ edges are needed to connect m Steiner nodes, and the weight of each edge is at least 1. Recall, that the lightweight degree of each Steiner node is at most 4 and there are n edges in the cut (S_1, R_1). To lower bound the $Cost(\mathcal{D})$ we consider the following two cases:

1. If $n \leq 4m$, then the cost of edges of type (r, s) is greater than or equal to n, because each required node is connected to a Steiner node with an edge of weight at least 1.

$$n \leq 4m \Rightarrow m \geq 0.25n, \text{ therefore } Cost(\mathcal{D}_1) \geq m - 1 + n \geq 1.25n - 1.$$

2. Otherwise, when $n > 4m$, the cost of edges in the cut (S_1, R_1) is greater than $4m + 1.4(n - 4m)$, because only $4m$ required nodes can be connected with lightweight edges to S_1, the remaining required nodes have to be connected with edges of weight strictly greater than 1.4.

$$\begin{aligned} & n > 4m \Rightarrow m < 0.25n; \\ & \qquad Cost(\mathcal{D}_1) \geq m - 1 + 4m + 1.4(n - 4m) \\ & \qquad\qquad = 1.4n - 0.6m - 1 \\ & \qquad\qquad \geq 1.4n - 0.15n - 1 \geq 1.25n - 1. \end{aligned}$$

The two sets of edges $\mathcal{D}_1$ and $\mathcal{A}_1$ are disjoint, because $\mathcal{A}_1$ consists of edges between two required nodes only, while $\mathcal{D}_1$ are edges of type (s, s) and (r, s). In both of the above cases:

$$\begin{aligned} & Cost(\mathcal{A}_1) \leq 2n - 2, \\ & Cost(\mathcal{D}_1) \geq 1.25n - 1 > \frac{5}{8}(2n - 2) \geq \frac{5}{8}Cost(\mathcal{A}_1), \\ & Cost(\mathcal{D}_1) \geq \frac{5}{8}Cost(\mathcal{A}_1). \end{aligned} \tag{2.2}$$

Similarly,

$$Cost(\mathcal{D}_i) \geq \frac{5}{8}Cost(\mathcal{A}_i), \ \forall i = 1, \ldots, l, \tag{2.3}$$

where l is the number of connected components of Steiner nodes in τ.

Let G' be the graph obtained after all connected components of Steiner nodes are removed, and the required nodes connected to them contracted to single node. Therefore G' is a graph with required nodes only. Since τ is a minimum Steiner tree in G then it must span the nodes in G'. Let τ' be a subset of edges of τ spanning the nodes of G'. Then

$$\begin{aligned} Cost(\tau) &= Cost(\tau') + \sum_{i=1}^{l} Cost(\mathcal{D}_i) \geq Cost(\tau') + \frac{5}{8}\sum_{i=1}^{l} Cost(\mathcal{A}_i) \\ &\geq \frac{5}{8}\,(Cost(\tau') + \sum_{i=1}^{l} Cost(\mathcal{A}_i)). \end{aligned}$$

We used Equation (2.3) to obtain the first inequality above.
(Lemma 11) □

Note that the graph G_k obtained from the instance G at the end of the first phase of the algorithm and the tree τ_k, which is an optimal Steiner tree in G_k do satisfy the conditions of Lemma 11. Note that the tree built by the algorithm during phase two denoted as τ_{A_2}, is a minimum cost spanning tree on R in G_k. Then

$$Cost(\tau_{A_2}) \leq Cost(\tau_k) + \sum_{i=1}^{n} Cost(\mathcal{A}_i),$$

where $Cost(\mathcal{A}_i)$ are the edges used in Lemma 11, and τ_k is the optimal tree τ. Applying Lemma 11 we conclude that $Cost(\tau_{A_2}) \leq \frac{8}{5}Cost(\tau_k) \leq \frac{7}{4}Cost(\tau_k)$.

Now we complete the proof of Theorem 9. The cost incurred by the algorithm is:

$$\begin{aligned} Cost(\mathcal{A}) &= Cost(\mathcal{A}_C) + Cost(\tau_{A_2}) \leq \frac{7}{4}(Cost(T) - Cost(\tau_k)) + \frac{7}{4}Cost(\tau_k) \\ &= \frac{7}{4}Cost(T) \end{aligned} \tag{2.4}$$

□

Does the algorithm Contract fit the model of adaptive priority algorithms? The first phase must compute the lightweight degree of each Steiner node, an operation which does not seem to fit the model of adaptive priority algorithms. Nevertheless, the ideas discussed above give rise to an adaptive priority algorithm which mimics the actions of algorithm Contract, and performs almost as well. Below follows the description of the Adaptive Contract algorithm, achieving an approximation ratio of 1.8 (c.f Theorem 12).

Algorithm **Adaptive Contract**

I. First the algorithm learns whether the instance has 0, 1, or greater or equal than 2 Steiner nodes. The edges of the instance are sorted according to the following priority function:

- $\pi(e) = 0$ if e is an edge between two Steiner nodes. If such an edge exist, it is rejected. Note that, the algorithm Contract does not use edges between Steiner nodes, and therefore they can be safely rejected. If there is such an edge, the algorithms learns that the number of Steiner nodes is greater or equal to 2, and also records their names. Then to to step II.3 below.

- $\pi(e) = 3 - w(e)$, if e is an edge between a Steiner and a required node. If the algorithm sees such an edge then, the edge will be rejected (note that e would be the heaviest weight edge). If the first data item of the instance has a priority value in the interval $[1, 2]$, then the number of Steiner nodes in the instance is exactly 1; In this case go to step II.2 below.

- $\pi(e) = 2 + w(e)$, if e is an edge between two required nodes. If the first data item is of this type, then the edge is accepted. In this case the instance graph consists of required nodes only. Note that accepting the minimum weight required-required edge is the first step of Kruskal algorithm. Go to stage II.1 below.

  The above priority function is used only once at the beginning of the algorithm.

II. Next we consider three cases, based on the number of Steiner nodes in the instance (determined in step **I**):

1. Suppose the instance has no Steiner nodes. Then the metric Steiner tree problem reduces to the problem of finding a minimum cost spanning tree in the graph, and we continue with the Kruskal algorithm.

2. Suppose the instance has exactly one Steiner node, call it s. In this case, the heaviest edge $e = (s, r_i)$ was rejected in stage I.

   (a) First accept the lowest weight edge $e = (r_i, r_j)$ between two required nodes, r_i, r_j, contracting them into a single required node, call it $c_1(r_i, r_j)$ (Note that the first step of Kruskal algorithm is the same). This is done to connect r_i. If there is no such an edge, then the instance has zero or one required nodes and the algorithm terminates without accepting any edges. Then the algorithm rejects the heaviest edge between r_i or r_j and an unseen required node, until all required nodes are seen. In doing this the algorithm computes the contracted graph $G_1 = \mathit{contract}(G, c_1(r_i, r_j))$ and learns the number and the names of all required nodes.

   (b) Reject all edges of type (r, s) of weight greater than 1.4 and calculate the lightweight degree of s. The degree is the number of required nodes computed in step 2(a) minus the number of heavyweight edges rejected.

   (c) If the lightweight degree of the Steiner node is greater or equal to five, then accept all lightweight (weight smaller than 1.4) edges of type (r, s), where $r \in R \setminus \{r_1, r_2\} \cup \{c_1\}$ and contract all those nodes to a single required node. Otherwise go to the next step.

   (d) Connect the remaining required nodes (if any) and the contracted nodes via a minimum cost spanning tree, using Kruskal's algorithm considering edges between required nodes only[9].

[9]Note that, if the instance has three, four, or five required nodes only, then the algorithm builds a minimum cost spanning tree using the Kruskal algorithm.

3. Suppose the instance has two or more Steiner nodes. First learn the number and the names of Steiner and required nodes (will become clear below). Then reject the heavyweight edges (weight bigger than 1.4) between Steiner and required nodes. Since the instance is a complete graph the lightweight degree of each Steiner node can be obtained by subtracting from the total number of required nodes the number of rejected heavyweight edges incident to that Steiner node. Then continue with the algorithm Contract. Details follow below.

   (a) Reject all edges of type (s, s) and learn the number and the names of all Steiner nodes in the instance.

   (b) Learn the names and number of the required nodes as in 2(a): accept the smallest weight edge between two required nodes and then compute the contracted graph[10].

   (c) Learn the lightweight degree of each Steiner node in the instance: Reject all heavyweight edges (weight strictly greater than 1.4) of type (r, s). For each Steiner node s dynamically compute the set of required nodes connected to it, called this set R_s. Recall that the instance is a metric graph, therefore each Steiner node s is connected to all required nodes in the instance. At the beginning of this step for each Steiner node s, R_s is initialized to the set of all required nodes. When an edge (r, s) is rejected the required node r is removed from the corresponding set of the Steiner node s and $R_s \leftarrow R_s - \{r\}$. When all heavyweight edges are removed, the lightweight degree of each Steiner node s is simply the size of set R_s.

   (d) At this point all remaining edges of type (r, s) have weight smaller than or equal to 1.4. If any Steiner node has a lightweight degree higher than five, then all edges incident to the highest degree such node, call it S, are added to the solution and the connected required nodes are contracted to a single

[10]See *Contract required nodes phase* of algorithm Contract for a formal description of the contracted graph.

required node. To update the lightweight degree of the remaining Steiner nodes we proceed as follows. Let s_1 be one such Steiner node and let R_{s_1} be the set of required nodes connected to it via lightweight edges. We remove from R_{s_1} all but one, if there are multiple required nodes members of R_S. The new lightweight degree of s_1 is the size of the new R_{s_1}.

(e) When no Steiner node has a degree higher than four all remaining edges between Steiner and required nodes are rejected.

(f) The remaining edges in the instance are between required nodes (contracted required nodes are considered required). Continue with Kruskal's algorithm to build a minimum cost spanning tree.

A formalization of the algorithm within the framework of adaptive priority algorithms can be found in Appendix A.

Theorem 12. *The Adaptive Contraction algorithm is a* 1.8*-approximation for the metric Steiner tree problem for graphs with edge weights* $[1, 2]$.

Proof. If the instance has no Steiner nodes, then the solution is optimal because the Adaptive Contract simulates Kruskal algorithm and builds a minimum cost spanning tree. If the number of required nodes is zero or one, then the solution is the empty set and therefore optimal. If the instance has two required nodes, the solution is the edge between them and thus is optimal.

If the number of required nodes r, in the instance is between three and five then the algorithm builds a minimum cost spanning tree using the Kruskal's algorithm. The cost of such a tree is at most $2(r-1)$. If the optimal Steiner tree does not use Steiner nodes then the optimal Steiner tree is a spanning tree in the graph induced by the required nodes only and therefore the choices made by the algorithm are optimal. Otherwise the cost of an optimal Steiner tree must be at least r, in which case the approximation ratio is at most $2 - \frac{2}{r} \leq 1.6$

The remaining case is when there are at least six required nodes and at least one Steiner node. Let G be an instance graph. Let the edge (r_1, r_2) be the minimum weight edge of type (r, r) in G. Let T and T_1 be optimal Steiner trees in G and $G_1 = contract(G, \{r_1, r_2\})$, respectively. Let T_{A_1} be the tree built by the algorithm Contract on G_1. Note that, the solution output by Adaptive Contract on G is $T_A = T_{A_1} \cup \{(r_1, r_2)\}$, therefore $Cost(T_A) = w(r_1, r_2) + Cost(T_{A_1}) \leq 2 + Cost(T_{A_1})$. Also note that, T_{A_1} is the result of running the algorithm Contract on G_1. Therefore, by Theorem 9 Equation(2.4) we have that $Cost(T_{A_1}) \leq 1.75Cost(T_1)$. Combining the last two inequalities we have that

$$\begin{aligned} Cost(T_A) &\leq 2 + 1.75Cost(T_1) = 1.75(Cost(T_1) + 1) + \frac{1}{4} \\ &\leq 1.75Cost(T) + \frac{1}{4}. \end{aligned}$$

We used Equation (2.1) of Lemma 10 to obtain the above inequality.

Since the instance is a graph with at least six required nodes, then the cost of any tree spanning the set of required nodes is at least five, then $1.75Cost(T) + \frac{1}{4} \leq 1.8Cost(T)$, concluding that

$$Cost(T_A) \leq 1.8Cost(T).$$

□

2.4.4 Maximum Independent Set Problem

We study the performance of ADAPTIVE priority algorithms for the MIS problem. Given an instance graph $G = (V, E)$ the problem is to find a subset $I \subseteq V$, such that if $u \in I$ and $v \in I$, then $(u, v) \notin E$. The objective is to maximize $|I|$.

In this section we prove a lower bound of $\frac{3}{2}$ on the approximation ratio for the MIS problem on graphs with maximum degree 3.

Theorem 13. *No ADAPTIVE priority algorithm in the node model can achieve an approximation ratio better than $\frac{3}{2}$ for the MIS problem, even for graphs of degree at most 3.*

Proof. We view the MIS problem in the node model. The set of data items are the vertices of the graph with their names and adjacency lists. The set of decision options is $\Sigma = \{accepted, rejected\}$. The Adversary sets T to be the graphs shown in Figure 2.4, and all isomorphic copies of these graphs. Therefore Γ_1 is all possible tuples of node name and adjacency lists of size 2, and 3. Both graphs G_2 and G_3 have same number of vertices equal to 6, and the degree of each vertex is either 2 or 3, hence the Solver cannot distinguish whether the instance is a graph isomorphic to G_2 or G_3 a priori. The Solver orders all possible data items. Based on her first choice and decision made, the Adversary plays the following strategy:

1. The first data item chosen by the Solver is a vertex of **degree** 3.

   - If the Solver decides to accept it, then the Adversary presents an isomorphic copy of graph G_3, where the node chosen by the Solver is C. The possible solutions for the Solver are $\{B, C\}$ or $\{C, E\}$, while the Adversary selects $S_{adv} = \{A, D, E\}$, with $\frac{|S_{adv}|}{|S_{sol}|} = \frac{3}{2}$. The Adversary wins the game for any approximation ratio $\rho < \frac{3}{2}$.

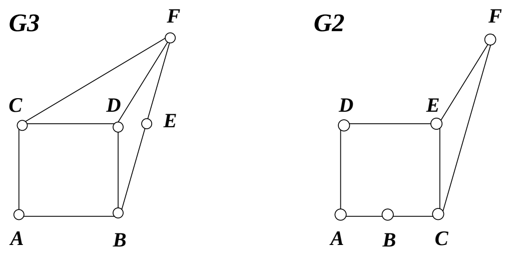

Figure 2.4: Nemesis graphs chosen by the Adversary

   - If the Solver decides to reject the node, then the Adversary presents an isomorphic copy of G_3, such that the node chosen by the Solver is D. The possible solutions for the Solver are
     $\{C, B\}, \{C, E\}, \{F, A\}, \{F, B\}$, or, $\{E, A\}$. The Adversary selects solution $\{A, D, E\}$, and wins any game for approximation ratio $\rho < \frac{3}{2}$.

2. The first data item chosen by the Solver is a vertex of **degree** 2.

   - If the Solver decides to take it, then the Adversary presents an isomorphic copy of G_2, in Figure 2.4, in which the data item chosen by the Solver is A. Any solution for the Solver has size at most 2, and the possibilities are $\{A, C\}$, $\{A, E\}$, $\{A, F\}$, while the Adversary chooses $\{B, D, F\}$ and wins the game for any ratio $\rho < \frac{3}{2}$.
   - If the Solver decided to reject the node of degree 2, then the Adversary presents an isomorphic copy of graph G_3 in Figure 2.4, in which the node chosen by the Solver is A. The possible solutions for the Solver are $\{B, C\}$, $\{B, F\}$, $\{C, E\}$, or $\{D, E\}$ of size 2, while the Adversary chooses $\{A, D, E\}$ and wins the game for any approximation ratio $\rho < \frac{3}{2}$.

□

2.5 Memoryless Priority Algorithms

Memoryless priority algorithms are a subclass of adaptive priority algorithms. Although the model is general, it can only be applied for problems where the decision options are $\Sigma = \{accept, reject\}$. Problems having this form include some scheduling problems, some graph optimization problems (vertex cover, Steiner trees, maximum clique, path problems, etc.), and also facility location, set cover, and several network optimization problems. Memoryless priority algorithms have a restriction on what *part of the instance* (data items considered in the past) they can remember. We would like to think of the decision of the algorithm to reject a data item as a *'no-op'* instruction. The state of the algorithm and the remaining data items and their priorities do not change, but the current data item is "forgotten" by the algorithm and removed from the remaining sequence of data items. The algorithm stores in its memory (state) only data items that were accepted. Each decision made by the algorithm is based on the information presented by the current data item and the state. An example of a memoryless priority algorithm is Kruskal's algorithm for building minimum cost spanning tree. The algorithm considers the edges of the graph in non-decreasing order one at a time. The "state" of the algorithm is the current spanning forest $\mathcal{F}$, namely those edges that have been accepted. Initially $\mathcal{F} = \emptyset$. At each step the algorithm considers the next edge e, according to the order and accepts e, provided $\mathcal{F} \cup \{e\}$ remains a forest. If the edge e is accepted, then the state of the algorithm is updated $\mathcal{F} \leftarrow \mathcal{F} \cup \{e\}$, otherwise the edge is rejected and forgotten. The algorithm terminates when all edges in the graph are considered.

The formal framework of a memoryless adaptive priority algorithm is:

```
MEMORYLESS PRIORITY ALGORITHM
Input: instance $I \subseteq \Gamma$, $I = \{\gamma_1, \ldots, \gamma_d\}$
Output: solution $S = \{(\gamma_i, \sigma_i) |\ \sigma_i = accept\}$

    - Initialize: a set of unseen data items $U \leftarrow I$, a partial solution $S \leftarrow \emptyset$, and a
      counter $t \leftarrow 1$
    - Determine an ordering function: $\pi_1 : \Gamma \rightarrow \mathbb{R}^+ \cup \{\infty\}$
    - Order $\gamma \in U$ according to $\pi_1(\gamma)$

Repeat
    • Observe the first unseen data item $\gamma_t \in U$
    • Make an irrevocable decision $\sigma_t \in \{accept, reject\}$
    • If $(\sigma_t = accept)$, then
        – update the partial solution: $S \leftarrow S \cup \{\gamma_t\}$
        – determine an ordering function: $\pi_{t+1} : \Gamma \rightarrow \mathbb{R}^+ \cup \{\infty\}$
        – order $\gamma \in U - \{\gamma_t\}$ according to $\pi_{t+1}$
    • If $(\sigma_t = reject)$, then
        – Forget $\gamma_t$, i.e., delete it from current state
    • Remove the processed data item $\gamma_t$ from $U$; $t \leftarrow t + 1$
Until ($U = \emptyset$)
Output $S$
```

The differences between adaptive priority algorithms and memoryless algorithms are:

- **Reordering the inputs:** Priority algorithms with memory can reorder the remaining data items in the instance after each decision, while memoryless algorithms can reorder the remaining input after they *accept* a data item.
- **State:** Memoryless algorithms *forget* data items that were rejected, while memory algorithms keep in their state all data items and the decisions made.
- **Decision making process:** In making decisions, memory algorithms consider all processed data items and the decisions made, while memoryless algorithms can only use the information about data items that were accepted.

On one side, memoryless algorithms are intuitive. Consider Prim's and Dijkstra's algorithms. Both algorithms are adaptive priority algorithms and grow a tree by adding an edge at each iteration. In the case of Prim's algorithm, when all nodes of the graph are connected by the currently grown spanning tree the algorithm rejects all remaining edges of the graph. In this sense Prim's algorithm is memoryless, since the priority function and the decisions made depend only on the edges added to the current spanning tree. Note that once this algorithm rejects an edge, it never accepts another edge (in this sense any algorithm with this structure can trivially be regarded as a memoryless algorithm). Similarly, Dijkstra, and the known greedy heuristics for the facility location, set cover, and vertex cover problems can be classified as memoryless.

On the other hand, memoryless algorithms can be considered counterintuitive, in the sense that the algorithm could explore the structure of the instance by giving lower priority to "unwanted" data items and rejecting them, and thus could achieve better performance. For example, consider the weighted independent set problem on cycles. If an algorithm rejects the smallest weight node, then the algorithm has learned 1) the value of the smallest weight; 2) no other vertex of the instance has a smaller weight; 3) the name of the vertex with the smallest weight; 4) the names of the neighbors of the node with the smallest weight. Perhaps exploring this knowledge could give the algorithm more power? To characterize the power of memoryless adaptive priority algorithms we first define a combinatorial game and use the game to prove lower bounds on the performance of all algorithms in the class.

2.5.1 A Memoryless Adaptive Priority Game

We define a two person game between the Solver and the Adversary to characterize the approximation ratio achievable by a deterministic memoryless adaptive priority algorithm.

Let Π be a maximization problem, with priority model defined by μ, Σ, and Γ, where μ is the objective function, Γ is the type of a data item, and $\Sigma = \{accepted, rejected\}$ is the set of decision options available for each data item. Let T be a finite collection of instances of Π. The game between the Solver and the Adversary for a fixed approximation ratio $\rho \geq 1$ is as follows:

Memoryless Game (Solver, Adversary, ratio ρ)

1. The Solver initializes an empty memory M, $M \leftarrow \emptyset$, and defines a total order π_1 on Γ

2. The Adversary picks a finite subset $\Gamma_1 \subseteq \Gamma$ with at least one instance $I \subseteq \Gamma_1$, $I \in T$; $R \leftarrow \emptyset$; $t \leftarrow 1$

3. **repeat until** ($\Gamma_t = \emptyset$)
   begin (Round t)

   (a) Let γ_t be the next data item in Γ_t according to π_t

(b) The Solver picks a decision σ_t for γ_t

- if ($\sigma_t = accepted$) then the Solver does the following
  - $M \leftarrow M \cup \{\gamma_t\}$
  - $\Gamma_t \leftarrow \Gamma_t - \gamma_t$
  - define a new total order π_{t+1} on Γ_t
- else ($\sigma_t = rejected$)
  - The Adversary remembers the rejected data item $R \leftarrow R \cup \{\gamma_t\}$
  - The Solver forgets γ_t; $\Gamma_t \leftarrow \Gamma_t - \gamma_t$

(c) The Adversary chooses $\Gamma_{t+1} \subseteq \Gamma_t$

$t \leftarrow t+1$

end; (Round t)

4. Endgame: The Adversary presents an instance $I \in T$ with $M \subseteq I \subseteq M \cup R$, and a solution S_{adv} for I. The Solver wins the game if one of the two events happen
   - No such instance I exists
   - If I exists and it is a valid instance and Solver presents a valid solution S_{sol} for I such that $M \subseteq S_{sol}$ and $\frac{\mu(S_{adv})}{\mu(S_{sol})} \leq \rho$.

Lemma 14. *Let Γ_1 be any finite set of data items. There is a winning strategy for the Solver in the Memoryless game defined above for a ratio ρ if and only if there is a memoryless adaptive priority algorithm that achieves an approximation ratio ρ on every instance of Π in T.*

Proof. • Suppose the Solver has a winning strategy for the game with a ratio ρ on all instances in T. We describe a memoryless adaptive priority algorithm which achieves an approximation ration ρ.

Let I be the input instance. The memoryless algorithm plays the role of the Adversary in the game between the Solver and the Adversary. Each round of the game corresponds to one iteration of the loop of the memoryless algorithm and the memoryless algorithm maintains the invariant that: *1) At the beginning of each round the data item next in the order according to π is a data item from I; 2) The decision of the Solver and the memoryless algorithm during the round are the same ($M = S$).*

Suppose the invariant is maintained for the first i rounds. During round $i{+}1$, let γ_{i+1} be the first item in the order defined by π_{i+1}, where $\gamma_{i+1} \in I$. If the Solver accepts γ_{i+1} so does the memoryless algorithm, $S = S \cup \{\gamma_{i+1}\}$. The memoryless algorithm observes π_{i+2} defined by the Solver, and computes the new Γ_{i+2} as follows. Let $\gamma_{i+2} = \min_{\gamma \in I-\{\gamma_1,\ldots,\gamma_i,\gamma_{i+1}\}} \pi_{i+2}(\gamma)$, where $\{\gamma_1, \ldots, \gamma_i\}$ are the i data items of I processed during rounds $1, \ldots, i$. Then $\Gamma_{i+2} = \Gamma_{i+1} - \{\gamma : (\gamma \in \Gamma_{i+1}) \wedge (\pi_{i+2}(\gamma) < \pi_{i+2}(\gamma_{i+2}))\}$. The memoryless algorithm removed from Γ_{i+1} all data items with π_{i+2} values smaller than $\pi_{i+2}(\gamma_{i+2})$ which therefore would have appeared in the π_{i+2} order before γ_{i+2}.

If the Solver rejects γ_{i+1} the memoryless algorithm rejects as well. The order of $\Gamma_{i+1} - \{\gamma_{i+1}\}$ does not change. A new Γ_{i+2} is computed as follows. Let $\gamma_{i+2} = min_{\gamma \in I-\{\gamma_1,\ldots,\gamma_{i+1}\}} \pi_{i+1}(\gamma)$, then $\Gamma_{i+2} = \Gamma_{i+1} - \{\gamma : (\gamma \in \Gamma_{i+1} \wedge \gamma \notin I) \wedge (\pi_{i+1}(\gamma) < \pi_{i+1}(\gamma_{i+2}))\}$.

The invariant is maintained during round $i+1$, the decisions made are the same and the data item according to the order π during the next round is a data item from I. By induction the invariant is maintained until the end of the game. The memoryless algorithm terminates the game when all data items from I are processed, say that happens at round t, then the memoryless algorithm truncates $\Gamma_t = \emptyset$.

During the Endgame the memoryless algorithm presents an instance I and a solution $S = M$. Since the Solver has a winning strategy for the game with a ratio ρ, $\frac{\mu(S_{adv})}{\mu(S)} \leq \rho$, then the approximation ratio secured by the algorithm is at most ρ, because $\rho \cdot \mu(S) \geq S_{adv}$, for any valid S_{adv}.

- For the converse, suppose there is a memoryless adaptive priority algorithm that achieves an approximation ratio ρ on every instance in T. We show a winning strategy for the Solver in the game defined above. The Solver simulates the actions of the memoryless adaptive priority algorithm during each round of the game. Thus the strategy of the Solver is to maintain the invariant that *after the first t rounds of the game, the decisions made by her are the same as the decisions made by the memoryless priority algorithm, the ordering functions on data items π_t, and the memory $M = S$ are also the same.*

  During round $t+1$ the Solver simulates the memoryless algorithm on the data item γ_{t+1}, which is the first data item according to π_{t+1}. If the memoryless algorithm accepts γ_{t+1}, then the Solver updates her memory M accordingly, and defines the new total order on Γ according to the ordering π_{t+2} used by the memoryless algorithm. If the memoryless algorithm rejects γ_{t+1}, so does the Solver and the next round begins. The invariant is maintained during round $t+1$ and by induction it holds until the Endgame phase.

  During the Endgame the Adversary presents an instance I subject to the constraints $M \subseteq I \subseteq M \cup R$. If that is not the case, the Adversary loses the game and the Solver wins. Otherwise the Solver simulates the memoryless algorithm on the portion of I not processed thus far. Say S is the solution output by the memoryless algorithm, then the Solver commits to solution $M \cup S$. Because the adaptive priority algorithm achieves approximation ratio ρ on any instance then $\rho \cdot \mu(M \cup S) \geq \mu(S_{adv})$, for any valid solution S_{adv} for the instance I, and the Solver wins the game.

$\square$

Corollary 15. *If there is a winning strategy for the Adversary in the game defined above for a ratio ρ, then there is no memoryless adaptive priority algorithm that achieves an approximation ratio better than ρ.*

Theorem 16. *There is an optimal strategy for the Solver for the Memoryless game, that* *has the following property: once the Solver rejects one data item, she never accepts any later data items.*[11]

Proof. To prove the theorem it suffices to show that any winning strategy for the Solver for the Memoryless game for a fixed ρ, in which accepting and rejecting decisions are not separated into distinct phases, can be converted to a winning strategy for the game with same ρ, in which the Solver never accepts after she has rejected a data item. Intuitively, since Π is a maximization problem, whose decision options are $\{accept, reject\}$, the Solver gains nothing after rejecting a data item (the value of the objective function of her solution depends only on the data items accepted and hence does not increase when a data item is rejected). Furthermore, the Adversary can choose to either add the data item to the final instance during the Endgame or not, depending on whether he would gain from accepting the data item or not. Thus the action of the Solver to reject a data item in the model of memoryless algorithms could only benefit the Adversary's strategy but Solver's.

Let S be any strategy for the Solver, and π_i be the ordering functions used by the Solver during the rounds ($i = 1, 2, ..$) of the game. We construct a new strategy S' with the properties defined in the theorem and ordering functions π'_i. The idea is to give priority values of infinity for all data items that were rejected (denoted by the set R in the conversion algorithm below) under the strategy S.

Conversion Algorithm

```
Input: $S$ a strategy for the Solver
Output: $S'$ modified strategy for the Solver
1. Initialize: $R \leftarrow \emptyset$; $t \leftarrow 1$; $\Gamma_1 = \Gamma$.
2. repeat until $(\Gamma_t \subseteq R)$
    • Obtain $\pi_t : \Gamma_t \rightarrow \mathbb{R}^+$ from $S$;
      compute the first $\gamma_t \in \Gamma_t$ according to $\pi_t$, such that $\sigma_{\gamma_t} = accept$ under $S$.
```

[11] *Similar result was proved in [14],[11],[7]. [11] refers to memoryless priority algorithms as "acceptance-first" adaptive priority algorithms.*

- $R \leftarrow R \cup \{\gamma \,|\, (\gamma \in \Gamma) \wedge (\pi(\gamma) \leq \pi(\gamma_t)) \wedge (\sigma_\gamma = reject)\}$
- Set the ordering function and decision for S' as follows:
  $\pi'(\gamma) = \pi(\gamma)_{|\Gamma_t - R}$
  $\pi'(\gamma) = \infty, \;\; \forall \gamma \in R$
  $\sigma'_{\gamma_t} = accept$
- $\Gamma_{t+1} \leftarrow \Gamma_t - R - \{\gamma_t\}$
  $t \leftarrow t + 1$

3. $\forall \gamma \in R$, $\sigma_\gamma = reject$.

Now we argue that the two strategies S and S' have the same decisions on data items and thus the ratio of the value of the objective function μ for the solution output by Solver is the same regardless which strategy she plays. Therefore, if S is a winning strategy for the Memoryless game with a fixed ratio ρ, then S' will be a winning strategy with the same ratio ρ. Let A_S and $A_{S'}$ be the set of data items accepted by strategies S and S', respectively. We claim that $A_S \subseteq A_{S'}$. Suppose there exists $\gamma \in A_S$ and $\gamma \notin A_{S'}$. This means that the strategy S accepted γ and S' rejected it. S' rejects γ if and only if $\gamma \in R$. But $\gamma \in R$ if and only if S has rejected γ. Thus there is no such γ and $A_S \subseteq A_{S'}$. Similar argument establishes $A_{S'} \subseteq A_S$, thus concluding the proof. $\square$

2.5.2 Separation Between the Class of Adaptive Priority Algorithms and Memoryless Adaptive Priority Algorithms

In this section we show that the class of adaptive priority algorithms are more powerful than memoryless adaptive priority algorithms. Subsequently to the work in this paper, [39] proves separations between different sub-classes of priority algorithms in the context of Job Scheduling problems and defines a hierarchy for priority algorithms of bounded memory.

We consider the weighed independent set problem on cycles (WIS-2), where the weight of the nodes is 1 or k only, for some non-negative integer k. Note, that the independent set problem in cycles with arbitrary weights can be solved exactly in polynomial time, however we can use the problem to separate the power of priority algorithms and the class of memoryless priority algorithms. We show that memoryless adaptive priority algorithms cannot achieve an approximation ratio better than 2. Then we will show an adaptive priority algorithm (not memoryless) which achieves an approximation ratio of $(1+\frac{2}{k-1})$.

We first address the lower bound. We view the WIS-2 problem in the node model, with the following priority model: $\Gamma = (name, weight,$ *adjacency list*$)$, $weight \in \{1, k\}$, $|$*adj. list* $| = 2$, $\Sigma = \{accepted, rejected\}$.

Theorem 17. *No memoryless adaptive priority algorithm can achieve an approximation ratio better than* 2 *for WIS-2 problem.*

Proof. The Adversary chooses the set of instances T to be the graph shown on Figure 2.5 and all isomorphic copies (the exact weight of each node will become clear shortly).

The Solver must order the possible data items, and until she casts an accepting decision, she cannot reorder the data items. The Adversary's strategy is to wait until the Solver accepts a data item.

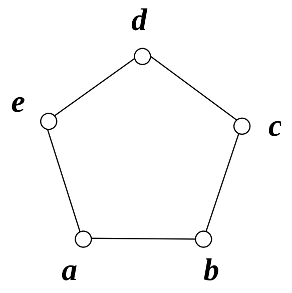

Figure 2.5: The nemesis graph for MIS problem.

- Case 1: If the first data item accepted by the Solver has weight k, then, the Adversary presents an isomorphic copy of the graph on Figure 2.5, where the vertices a, b, c, d, e have weights $\{(a,k), (b,k), (c,1), (d,1), (e,k)\}$, and the data item accepted by the Solver is (a,k). The Adversary chooses a solution $S_{Adv}(I) =$

$\{b, e\}$, while the best the Solver can do is $S_{Sol}(I) = \{a, c\}$, and the approximation ratio is $\rho = \frac{S_{Adv}}{S_{Sol}} \geq \frac{2k}{k+1} = 2 - \frac{2}{k+1}$.

The Adversary can choose k arbitrarily large, thus as $k \to \infty, \ \ \rho \to 2$.

- Case 2: If the first data item accepted by the Solver has weight 1, then the Adversary presents an isomorphic copy of the graph with the following weight assignments $\{(a, 1), (b, k), (c, 1),$
$(d, 1), (e, k)\}$ and the accepted by the Solver data item is $(a, 1)$. The Adversary chooses a solution $S_{Adv}(I) = \{b, e\}$, while the best the Solver can do is output $S_{Sol}(I) = \{a, c\}$. The approximation ratio is $\rho = \frac{S_{Adv}}{S_{Sol}} \geq \frac{2k}{2} = k$, the constant k can be chosen to be greater than 2.

If the Solver never accepts a data item, then the Adversary will select any non-empty independent set, and the approximation ratio will be infinity. Therefore the Adversary wins the game for any approximation ratio $\rho < \frac{3}{2}$. $\square$

To show the separation between the models we present an adaptive priority algorithm (non-memoryless) achieving approximation ratio $(1+\frac{2}{k-1})$ for the WIS-2 problem. We give an informal description of the algorithm first, and then a formalization as an adaptive priority algorithm, called ADAPTIVE WIS.

The algorithm first rejects the node with the smallest weight. Then it chooses a direction in which to traverse the cycle, by identifying what node will be considered next. The algorithm considers the two neighbors of the rejected node and selects the one with the bigger weight. If the weights of the two neighbors are the same, then the lexicographically first node is chosen. Let a and r point to the node to be considered (the current node), and the one rejected last, respectively. Note that once the direction of traversal is chosen, r and a would point to exactly one node of the instance, each. At any time the algorithm makes a decision about a and the neighbor of a, which has not been considered yet. Let that node be n, see Figure 2.6. The decision of the algorithm depends on the weights of a and n. If a has weight k, then a is added to the independent

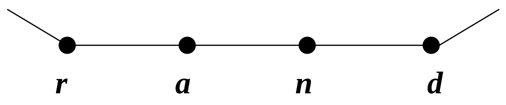

Figure 2.6: Fraction of the cycle considered.

set and n is rejected. For the next operation n becomes the next rejected node. If a has weight 1 and n has weight 1, then the algorithm behaves as before, that is a is added to the independent set, n is rejected, and n becomes the next r. If n has weight k, then n is added to the independent set. Let the other (unseen) neighbor of n be d (see Figure 2.6). Then d becomes the new r node. To distinguish between nodes a and n (refer to Figure 2.6) the ADAPTIVE WIS algorithm uses a predicate P. P evaluates to true when node n has weight k and a has weight 1. In this case, we want the algorithm to accept n and reject a.

Below we give a formalization of ADAPTIVE WIS algorithm in the framework of priority algorithms. We use the following notation: $w(\gamma)$ denotes the weight of data item γ and γ_r is the last rejected node.

Algorithm **ADAPTIVE WIS**

Input: Sequence of nodes $I \subset \Gamma$.

Output: $S \subset I$, S is an independent set in I.

Step 1 Initialization: $M \leftarrow \emptyset$, $S \leftarrow \emptyset$

Step 2 Reject the smallest weight node:

- $\pi(\gamma) = w(\gamma)$; $\gamma_r = \min_{\gamma \in I} \pi(\gamma)$; $\sigma_{\gamma_r} = reject$.
- update memory: $M \leftarrow M \cup \{\gamma_r\}$.
- a is set to be the neighbor of γ_r with largest weight, or the lexicographically first node if both neighbors have same weight.

Step 3 **repeat until** $(I = M)$

- Define a predicate $P = (|\{adj(v)\} \backslash M| = 2) \wedge (\{adj(v) \cap adj(\gamma_r)\} \neq \emptyset) \wedge (w(v) = k)$

$$\pi(v) = \begin{cases} 0 & \text{if } |\{adj(v)\} \setminus M| = 0 \\ 1 & \text{if } (|\{adj(v)\} \backslash M| = 1) \wedge (w(v) = k) \\ 2 & \text{if } P \text{ holds} \\ 3 & \text{if } |\{adj(v)\} \backslash M| = 1 \wedge (w(v) = 1) \\ \infty & \text{otherwise} \end{cases}$$

- $a = \min_{\gamma \in I, \gamma \notin M}\{\pi(\gamma)\}$. If $\pi(a) \leq 3$, then accept a and reject its neighbors $(adj(a))$. Add a and $adj(a)$ to M:

```
– if      $\pi(a) = 0$ then $\sigma_a = accept$
  else if $\pi(a) = 1$ then $\sigma_a = accept$
  else if $\pi(a) = 2$ then $\sigma_a = accept$
  else if $\pi(a) = 3$ then $\sigma_a = accept$
– $S \leftarrow S \cup \{a\}$
– $R = \{\gamma \mid \gamma \in adj(a) \backslash M\}$;  $\forall \gamma \in R$:  $\sigma_\gamma = reject$
  $\gamma_{last} = adj(a) \backslash adj(\gamma_r)$ ; $\gamma_r \leftarrow \gamma_{last}$
– $M \leftarrow M \cup \{a\} \cup \{adj(a) \backslash M\}$
```

Output S;

End.

Theorem 18. *ADAPTIVE WIS achieves an approximation ratio of* $(1+\frac{2}{k-1})$.

Proof. Let S_0 be the first node rejected by the algorithm. Let S_i be the set of nodes either accepted or rejected by the algorithm during the i-th iteration of the loop. Let

$$ALG_i = w(\text{ADAPTIVE WIS}(S_i))$$

be the combined weight of the nodes in S_i, accepted by the algorithm. Then clearly $ALG_0 = 0$, because the algorithm rejected the first node. And let $OPT_i = w(S_i)$

be the combined weight of those nodes in S_i accepted by the optimal solution. $I = S_0 \cup S_1 \cup \ldots \cup S_n$ is the set of all nodes in the instance graph.

$$OPT(I) = OPT_0 + \sum_{i=1}^{n} OPT_i$$

$$ALG(I) = ALG_0 + \sum_{i=1}^{n} ALG_i = \sum_{i=1}^{n} ALG_i$$

We will analyze the performance of the algorithm during Step 2, during which the algorithm rejects the first node, and Step 3 separately.

Suppose we can bound from above the ratio

$$\frac{OPT_i - ALG_i}{OPT_i} \leq x, \text{ for each iteration } i \in \{1, 2, \ldots, n\} \text{ of the loop during Step 3,}$$

for some constant x, whose value will be determined shortly. Then we have:

$$\frac{OPT_i - ALG_i}{OPT_i} \leq x, \ \forall i \in \{1, \ldots, n\}$$

$$\begin{aligned}
&1 - \frac{ALG_i}{OPT_i} \leq x \\
&(1-x)OPT_i \leq ALG_i \\
&(1-x)\sum_{i=1}^{n} OPT_i \leq \sum_{i=1}^{n} ALG_i = ALG(I)
\end{aligned}$$

If we choose $x = \frac{1}{k+1}$, then we have $ALG(I) \geq \frac{k}{k+1}\left(\sum_{i=1}^{n} OPT_i\right)$

Lemma 19. *During each iteration of repeat-until loop in Step 3,* $\frac{OPT_i - ALG_i}{OPT_i} \leq \frac{1}{k+1}$ *holds.*

Proof. (Lemma 19) To prove the claim we consider the four cases for π, which determine the decision and the payoff for the algorithm. We show that $\frac{OPT_i - ALG_i}{OPT_i} \leq \frac{1}{k+1}$ in all cases.

Case 0: $\pi(a) = 0$

Assuming the input is a valid instance (a cycle) of the problem, then if the case ever occurs, it can only happen while processing the last data item, and the algorithm does not lose anything (the algorithm always takes that element).

$$ALG_n \geq OPT_n, \frac{OPT_n - ALG_n}{OPT_n} \leq 0,$$

and the claim holds trivially.

Case 1: $\pi(a) = 1$

In this case the algorithm adds one node, say a, with weight k to the independent set and removes from the unseen sequence a's neighbor. Thus $ALG_i = k$ and $OPT_i \leq k$.

$\frac{OPT_i - ALG_i}{OPT_i} \leq 0$, and the claim holds trivially.

Case 2: $\pi(a) = 2$

The algorithm takes a node, say a, with priority 2, only if Case 1 cannot happen. The algorithm adds one node of weight k to its independent set and removes the neighbors of a (two nodes), from the remaining unseen sequence, whose combined weight is at most $k + 1$. Thus $ALG_i = k$ and $OPT_i \leq k + 1$:

$\frac{OPT_i - ALG_i}{OPT_i} \leq \frac{k+1-k}{k+1} = \frac{1}{k+1}$, and the claim holds .

Case 3: $\pi(a) = 3$

The algorithm takes node with priority 3 only when no node has priority 1, or 2. In this case the algorithm adds a node of weight 1 and removes a node of weight 1. Thus $ALG_i = 1$ and $OPT_i \leq 1$:

$\frac{OPT_i - ALG_i}{OPT_i} \leq 0$, and the claim holds trivially.

Case 4: $\pi(a) = \infty$

We claim this case never happens. Every valid instance of the problem is a cycle and a singleton is not a valid instance. The algorithm initially removes the node with the minimum weight. If the number of nodes in the instance is greater or equal to five, then after the removal of the node with the smallest weight, there will be nodes with degree one and two left, thus there will be nodes with π values $1, 2$, or 3. Now assume the number of nodes in the instance is smaller than five. The algorithm performs optimally on cycles of size 2 and 3, where the independent set is one node, by selecting the heaviest weight node. On squares, the independent set is of size 2 and algorithm's worse performance is observed on instance, $\{(a, 1), (b, 1), (c, k), (d, 1)\}$, where the algorithm selects an independent set with weight k, while OPT gets $k + 1$ and the claim holds.

This concludes the proof of the claim. □

We finish the proof of Theorem 18 by evaluating the performance of the ADAPTIVE WIS during Step 2. Initially, the algorithm removes a node of weight 1 or k. If the algorithm removed a node with weight k, then every node in the cycle has weight k and the algorithm performs optimally by selecting every other node as an independent set. Similarly the algorithm is optimal on cycles where all nodes have weight 1. Thus we have to analyze the performance of the algorithm on instances whose weights are k and 1. For those cycles it is obvious that $ALG_0 = 0$ and $OPT_0 = 1$. What we have so far is:

$$ALG(I) = ALG_0 + \sum_{k=1}^{n} ALG_i \ \geq \ \frac{k}{k+1} \sum_{i=1}^{n} OPT_i \tag{2.5}$$

$$OPT(I) = OPT_0 + \sum_{i=1}^{n} OPT_i = 1 + \sum_{i=1}^{n} OPT_i \tag{2.6}$$

$$\sum_{i=1}^{n} OPT_i = OPT(I) - 1 \tag{2.7}$$

Equations (2.5), (2.6), and (2.7) imply:

$$ALG(I) \geq \frac{k}{k+1}(OPT(I) - 1) \tag{2.8}$$

We can trivially bound the weight of the independent set for OPT from below in this case (instances are cycles with weights 1 and k): $OPT(I) \geq k$. Thus $1 \leq \frac{OPT(I)}{k}$ and $OPT_0 = 1 \leq \frac{1}{k}\, OPT(I)$.

$$ALG(I) \geq \frac{k}{k+1}\,(OPT(I) - \frac{1}{k}\,OPT(I))$$

$$ALG(I) \geq (1 - \frac{2}{k+1})\,OPT(I)$$

$$\left(1 + \frac{2}{k-1}\right)\,ALG(I) \geq OPT(I)$$

□

Note that as $k \rightarrow \infty$, $\quad \frac{ALG(I)}{OPT(I)} \rightarrow 1$.

2.6 Notes

The work presented in this chapter initially appeared in 2004 SIAM SODA conference "Models of Greedy Algorithms for Graph Problems" by Sashka Davis and Russell Impagliazzo [20]. A full version was published in 2007 in Algorithmica [21] "Models of Greedy Algorithms for Graph Problems" by Sashka Davis and Russell Impagliazzo.

Chapter 3

Prioritized Branching Tree and Prioritized Free Branching Tree Models

In this chapter we focus on two very important optimization techniques the backtracking approach and dynamic programming algorithms. First we discuss a model for backtracking and dynamic programming designed by [1], called prioritized Branching Tree Algorithms. Next we present a new general model called prioritized Free Branching Tree Algorithms. We design a lower bound technique which we use to derive negative results for both models for the 7-SAT problem.

Backtracking is an important technique because it is general enough and can be applied to both search and optimization problems. At a very high level, a backtracking algorithm inspects the search space for a problem in depth first search manner and at each step attempts to determine whether some partial solution is promising or not. If the node is promising then it is explored, otherwise the whole subtree rooted at it is pruned. The state space tree could be traversed in either DFS or BFS manner. The Branch-and-Bound technique is a refinement of the backtracking approach which is applied only to optimization problems. Branch-and-Bound explores the search space tree, however it tries to choose the order traversal wisely so as to minimize the number of the branches visited and uses a *bound* (for a minimization problem that would be a lower bound and for a maximization problem we need an upper bound) for pruning the search.

It is hard to overstate the importance of dynamic programming paradigm as a technique for designing both exact and approximation algorithms. It has been successfully applied to solve exactly many graph optimization problems, scheduling problems, problems in bioinformatics and network optimization. The dynamic programming paradigm has been used to solve many problems approximately as well. [31] described the approach as follows: "To set about developing an algorithm based on dynamic programming, one needs a collection of subproblems derived from the original problems... such that the solution to the original problem can be easily computed from the solutions to the subproblems..There is a natural ordering on subproblems from *smallest* to *largest*... that allows one to determine the solution to a subproblem from the solutions to some number of subproblems." [1] designed a formal model of backtracking and dynamic programming algorithms which encompasses the backtracking approach and some of the known dynamic programming algorithms. The string edit distance algorithm, many scheduling algorithms, the standard knapsack approximation algorithm and others seem to fit the model. The DP-simple class of algorithms of Woeginger [51], are seen to fit the framework of fixed (order) pBT algorithms. A DP-simple algorithm on an input instance $X_1, \ldots, X_n$ "...goes through n phases. The k-th phase ($k = 1, \ldots, n$) processes the input piece X_k and produces a set $\mathcal{S}_k$ of states,... every state in the set $\mathcal{S}_k$ encodes a solution to the subproblem specified by the partial input $X_1, \ldots, X_k$,... and the state space $\mathcal{S}_k$ is obtained from the state space S_{k-1}."

Before we present the formal model for backtracking and dynamic programming algorithms of [7] let us look at an example. Consider the problem of finding the maximum weighted independent set on line graphs. The instance can be encoded as an array of elements $I = I_1, I_2, \ldots, I_n$, where the i-th element of the array, I_i, is the weight of the i-th element of the line graph. There are two decision options per node either to add the node to the independent set or not. Obviously not all possible decisions would result in a valid solution. Consider an instance of size n, where I_n denotes the weight of the last node. Let OPT_i be the maximum weighted independent set of the first i elements. The i-th node I_i is either part of the optimal solution or it not, therefore:

$$OPT_i = \max\{OPT_{i-1},\ OPT_{i-2} + I_i\} \tag{3.1}$$

The recurrence relation (3.1) gives rise to a backtracking algorithm for finding the maximum weight independent set on line graphs. For example, consider and instance of the

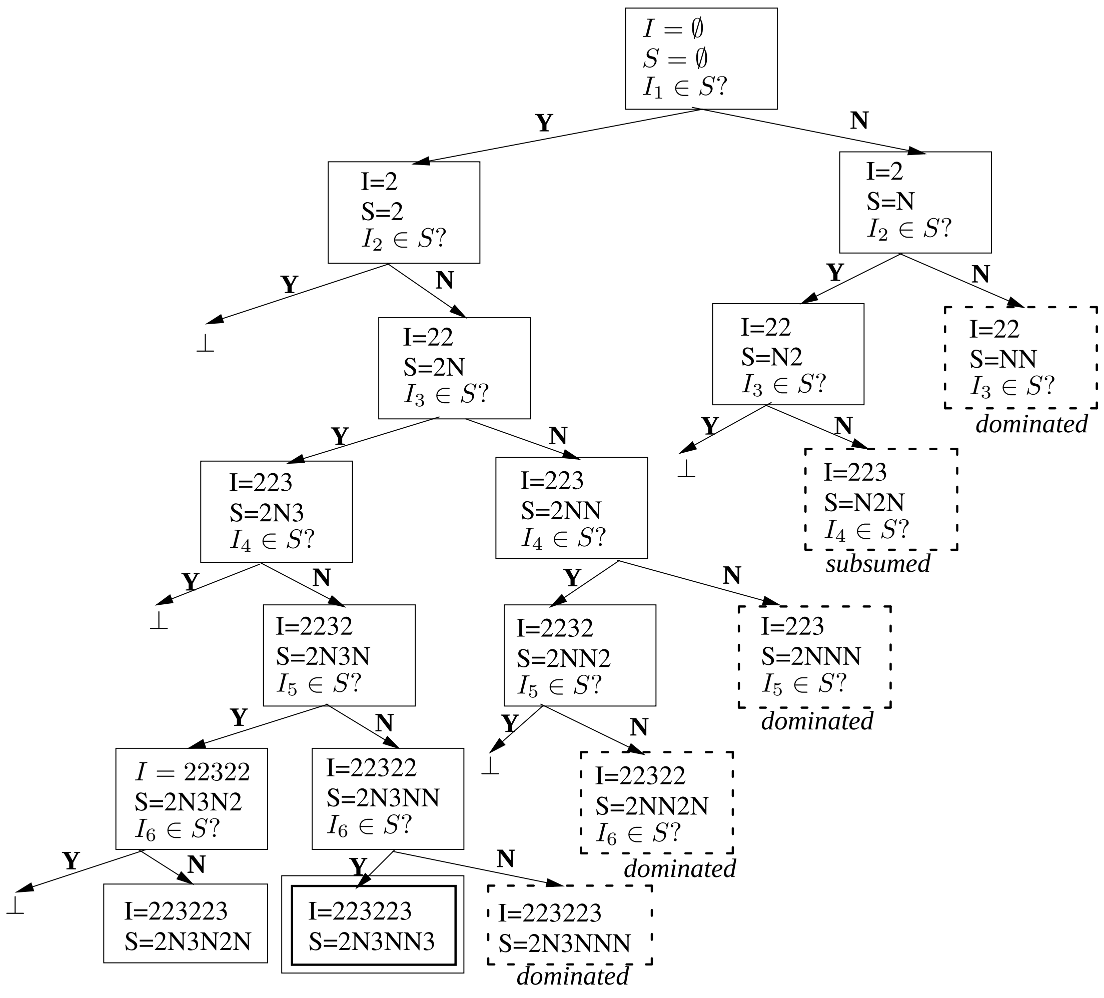

Figure 3.1: Example of a computation of a backtracking algorithm on an instance $I = 223223$.

problem $I = 223223$. Figure 3.1 shows the computation of a backtracking algorithm with pruning, which examines the solution space in BFS order. There are three situations in which a backtracking algorithm can terminate a branch and hence prune the whole subtree rooted at the terminated node. All solutions in which the algorithm chooses to accept two adjacent nodes are illegal, because such solutions will no longer encode valid independent sets. All these branches are terminated with $\perp$ on Figure 3.1. Another type of pruning occurs at nodes whose partial solution is *dominated* by another node. For example, at level two of the backtracking tree, the solution $S = NN$ with partial weighted independent set of weight 0 is dominated by the partial solution $S = 2N$ whose weight is 2. Note that any legal solution which can extend $S = NN$ will be able to extend $S = 2N$ as well, and the former will achieve a higher weight of the independent set. An example of the last type of pruning is seen on the third level of the backtracking tree. The branch encoding solution $S = N2N$ is terminated because there is another branch at the same level, $S = 2NN$, such that both branches encode solutions with the same partial weight of 2 and any legal extension of branch $S = N2N$ will also be a legal extension of branch $2NN$. Therefore branch $S = N2N$ can be terminated because it cannot deliver solutions with higher weights than already existing branch. The optimal solution is the double box solution $S = 2N3NN3$ of weight 8. For this problem, the natural ordering in which partial solutions are build is defined by the order of elements in the line. Initially we inspect the first element and commit to solutions about it. At level j we inspect the j-th node of the graph and will extend the live branches with the two possible decisions for I_j. Branches corresponding to illegal, dominated or subsumed solutions will be pruned. The backtracking algorithm for weighted independent set on line graphs runs in polynomial time because there are only two branches alive at any given time: a branch with optimal weight such that the currently inspected node is added to the solution and a branch, where the node is not added to the solution. Each partial solution extends one of the two previous partial solutions. When the algorithm has committed to decisions for all nodes of the line graph, the optimal solution is the one whose objective function value is the best. In our case the optimal solution is chosen from $S = 2N3NN3$ and $S = 2N3N2N$. The maximum weighted independent set has weight eight (denoted by a box with double border on Figure 3.1). The prioritized Branching Tree model of [1] is a formal model for backtracking and dynamic programming, and the example algorithm on Figure 3.1 does fit the mode. In what follows we compare the pBT model to the Priority model, described in Chapter 2 and then will formally define the pBT model.

The *prioritized Branching Tree* model of [1] is related to the Priority model. Recall the Priority model from the previous chapter. We could view the computation of a priority algorithm (on an instance of some optimization problem) as a path, in which each node chooses the next data item from the instance (using a priority function) and then commits to *one* irrevocable decision about it. [1] asked the questions: What if the algorithm is allowed to explore more than one option per data item? Would such algorithms be more powerful than the class of Priority algorithms? The pBT model of [1] extends the priority model by allowing the algorithm to explore multiple options for a given data item and thus to maintain and extend more than one solution to a problem at a given time. The pBT model resembles the priority model and both models explore the instance one data item at a time. A pBT algorithm branches when it decides to explore more than one option for a given data item, so the computation of a pBT algorithm on a given instance resembles a tree. We could view each individual path in the computation tree as a priority algorithm. The priority functions used by a pBT algorithm have only information theoretic restrictions and no computational limits, as is the case for priority algorithms.

3.1 The prioritized Branching Tree Model

Before we formally describe the pBT model of [1] lets look at another example. Consider the 3-SAT problem. A valid instance is a Boolean formula in conjunctive normal form (CNF), such that each clause has at most three literals and the problem is to assign values to the variables such that all clauses of the formula are satisfied. To describe the priority model for 3-SAT we need to specify the set of data items and the set of decision options. A data item is represented[1] as a variable name, together with full description of all clauses in which the variable participates in. For the 3-SAT problem each clause must have at most three literals. For example, data item $(x_1, (x_1 \vee x_k \vee x_l), (\neg x_1 \vee, x_i \vee \neg x_j))$ corresponds to variable x_1, where x_1 participates in two clauses. For 3-SAT formulas on n variables, the set of data items $\mathcal{D}$ has to contain all possible data items needed to describe all possible CNF formulas on n variables. The set of decision options is $\Sigma = \{0, 1\}$. An n-variable Boolean formula in CNF form (satisfiable or not) is encoded as a set of n data items. As it was the case for the *node priority model* for graph problems presented in Chapter 2, for the 3-SAT problem not all sets of data items encode valid instances. There is a dependence between different data items and the description of same clauses must agree in all data items. To clarify, suppose ψ is a Boolean formula in CNF, and suppose $C_1 = (x_1 \vee \neg x_2 \vee x_k) \in \psi$, then the three data items corresponding to the variables x_1, x_2, x_k must be consistent and have to contain C_1 in their clause lists.

While a priority algorithm chooses exactly one option for each data item of the instance, the pBT model extends the priority model by allowing a pBT algorithm to explore multiple assignments for each variable, thus a pBT algorithm maintains multiple partial assignments. The computation of a pBT algorithm is a rooted tree in which partial paths are identified by the *partial instance* and the *partial solution*. The partial instance is a set of data items observed (for 3-SAT those will be some set of variables and the clauses in which they participate), and the partial solution is the decisions made (for 3-sat, that will be assignments to the variables observed).

The next node in the path orders all possible data items, views the first data item in this order from the instance formula, say that is $(x_i, C_{i_1}, \ldots, C_{i_k})$. Then the node can choose to explore one of the two possible assignments $x_i = 1$ or $x_i = 0$, or both, or could terminate ($\perp$), the branch[2]. If $\psi \in$ 3-SAT then a correct pBT algorithm must have

[1]Weaker models are defined by not revealing the full description of the clauses in which the data item occurs and reveal only their name. Intuitively the more information is revealed the stronger the model and the more interesting the lower bounds will be.

[2]Note that pBT algorithms have the power to terminate a branch without justification.

at least one path in the tree, such that the choices made by the algorithm along this path define a satisfying assignment.

[1] gave the following formal definition of pBT algorithms.

Definition 5 (pBT Algorithm). *A pBT algorithm $\mathcal{A}$ for a search or optimization problem Π with priority model $(\mathcal{D}, \Sigma)$ and a family of objective functions $f^n : \mathcal{D}^n \times \Sigma^n \mapsto \{0, 1\}$ when Π is a search problem ($f^n : \mathcal{D}^n \times \Sigma^n \mapsto \mathbb{R}$ when Π is an optimization problem) is described by:*

- *Ordering (priority) functions: $\pi^k_{\mathcal{A}} : \mathcal{D}^k \times \Sigma^k \mapsto \mathcal{O}(\mathcal{D})$, where $\mathcal{D}^k, \Sigma^k$ describe the data items observed the decisions committed, respectively.*
- *Choice functions: $c^k_{\mathcal{A}} : \mathcal{D}^{k+1} \times \Sigma^k \mapsto \mathcal{O}(\Sigma \cup \{\perp\})$.*

Suppose $I \subset \mathcal{D}^n$ is an instance of Π then the computation of $\mathcal{A}$ induces an oriented[3] *computation tree $\mathcal{T}_{\mathcal{A}}(I)$ with the following recursive definition:*

- *The root of the tree $\mathcal{T}_{\mathcal{A}}(I)$ has an empty label.*
- *Each node u at level k is labeled by $(D^u_1, \ldots, D^u_k, \sigma^u_1, \ldots, \sigma^u_k)$, where $D^u_i, i = 1, \ldots, k$ are the data items observed by the algorithm along the path from the root to node u, and $\sigma^u_1, \ldots, \sigma^u_k$ are the corresponding decision options.*
- *If node u at level $k < n$ has a label $(D^u_1, \ldots, D^u_k, \sigma^u_1, \ldots, \sigma^u_k)$ and suppose D^u_{k+1} is the first data item from $I \setminus \{D^u_1, \ldots, D^u_k\}$ according to the total order $\pi^k_{\mathcal{A}}(D^u_1, \ldots, D^u_k, \sigma^u_1, \ldots, \sigma^u_k)$ and suppose $c^k(D^u_1, \ldots, D^u_k, D^u_{k+1}, \sigma^u_1, \ldots, \sigma^u_k) = (c_1, \ldots, c_d, \perp, c_{d+1} \ldots), c_i \in \Sigma, \forall i$, then u has d children (if $d = 0$ then, u has no children). When $d > 0$, then the children of u from left to right are $u_1, \ldots, u_d$ where the label of u_i is*

  $$(D^{u_i}_1, \ldots, D^{u_i}_k, D^{u_i}_{k+1}, \sigma^{u_i}_1, \ldots, \sigma^{u_i}_k, \sigma^{u_i}_{k+1}) = (D^u_1, \ldots, D^u_k, D^u_{k+1}, \sigma^u_1, \ldots, \sigma^u_k, c_i).$$

[3] *Meaning that the leaves of the tree are ordered from left to right.*

Naturally when Π is a search problem then a pBT algorithm $\mathcal{A}$ is correct if given an instance $I = \{D_1, \ldots, D_n\}, I \in \Pi$ such that if I is a YES instance, the tree $\mathcal{T}_{\mathcal{A}}(I)$, generated by $\mathcal{A}$ on input I, contains at least one leaf node labeled $(D_{i_1}, \ldots, D_{i_n}, \sigma_1, \ldots, \sigma_n)$ such that both conditions are satisfied:

1. $\{D_{i_1}, \ldots, D_{i_n}\} = \{D_1, \ldots, D_n\}$,
2. $f^n(D_{i_1}, \ldots, D_{i_n}, \sigma_1, \ldots, \sigma_n) = 1$.

[1] defined three submodels, which differ on how the priority functions are defined, i.e., what they are allowed to depend on.

- *Fixed-order pBT algorithms* the priority functions r^k cannot depend on k or on its arguments - the data items observed and the decisions made.
- *Adaptive-order pBT algorithms* have order functions r^k which depend on k and only on the data items observed $D_1, \ldots, D_k$.
- *Fully-adaptive-order pBT algorithms* have priority functions which depend on both arguments, that is r^k depends on both $D_1, \ldots, D_k$ and $\sigma_1, \ldots, \sigma_k$.

[1] showed separations between fixed-order and adaptive-order algorithms.

3.1.1 Complexity Measure for pBT Algorithms

[1] measured the efficiency of pBT algorithms by their width. Given a pBT tree $\mathcal{T}_{\mathcal{A}}(I)$ built by a pBT algorithm A, on an instance I, the width of $\mathcal{T}_{\mathcal{A}}(I)$ is the maximum over all levels of the number of nodes at a given level and will be denoted as $W_{\mathcal{A}}(I)$. This measure is natural because the width of the tree is directly related to the complexity of BFS search for backtracking algorithms and the space complexity of dynamic programming algorithms. On an instance of size n, any branch will have to read and commit to a decision for each item, hence has length at least n, and therefore the number of states in the tree is polynomially related to the maximum width of the

tree. The width of a given pBT algorithm $\mathcal{A}$, for instances of a given fixed size n is $W_{\mathcal{A}}(n) = \max\{W_{\mathcal{A}}(I) \mid |I| = n\}$.

3.2 The prioritized Free Branching Tree Model

The prioritized Free Branching Tree model (pFBT) takes the pBT model one step further. In addition to allowing multiple decision options to be explored, it allows multiple priority functions to be explored (no bounds on the fan-out). A pBT algorithm builds a tree in which each node of the tree uses an ordering/priority function to choose the next data item, and then commits to decisions about it. What if a node is allowed to use not one but many ordering functions, where each function selects the next data item from the instance, independently on the other functions, and therefore the algorithm could branch not on one but many data items from the instance? It could be possible that such an algorithm could detect inconsistent solutions earlier and could be able to terminate those fruitless branches earlier and hence be more efficient in solving hard problems. Allowing branching on many data items also increases the independent parallelism of the algorithm, which could be helpful when solving some problems more efficiently than the pBT model. The pBT model does not allow branching on more than one priority function at a time and the prioritized Free Branching Tree model extends the pBT model by inheriting all its capabilities, in addition it allows 'free branching' on many priority functions.

A prioritized Free Branching Tree algorithm has two types of states *free branching* states and *input reading* states. A free branching state makes a guess (we could call the free branching states also guessing states). The guess is an integer $i \in \mathbb{N} = \{1, 2, \ldots\}$ and cannot depend on the actual instance, meaning that the free branching state cannot read data items from the input instance and the guess making should be instance independent. The free branching state specifies its successors, which could be many, but the number of successors should be at most the value of the guess. Each successor is an input reading state, which has the same semantics as a state in the pBT model. An input reading state has a label, and invokes a priority function which orders all data items, then reads the next data item from the input instance according to the order and commits to a set of options. An input-reading state is augmented by appending the guess of its free branching parent to its label.

Below we give a definition of a prioritized Free Branching algorithm. A pFBT algorithm $\mathcal{A}$ for a search (or optimization) problem Π with priority model $(\mathcal{D}, \Sigma)$ and a family of objection functions $f^n : \mathcal{D}^n \times \Sigma^n \mapsto \{0,1\}$ (or $f^n : \mathcal{D}^n \times \Sigma^n \mapsto \mathbb{R}$ when Π is an optimization problem) is described by:

- Guessing functions at level k output an integer $g_k \in \mathbb{N}$.
- Ordering/priority functions at level k $\pi^k_{\mathcal{A}} : \mathcal{D}^k \times \Sigma^k \times \mathbb{N}^k \mapsto \mathcal{O}(\mathcal{D})$, where $\mathcal{D}^k$ describes the k data items observed so far, Σ^k describes the decisions committed, $\mathbb{N}^k$ describes the guesses made so far.
- Choice functions at level k $c^k_{\mathcal{A}} : \mathcal{D}^{k+1} \times \Sigma^k \mapsto \mathcal{O}(\Sigma \cup \{\perp\})$.

Let $I \subset D^n$ be an instance of Π, then the computation of $\mathcal{A}$ induces an ordered (from left to right) computation tree $\mathcal{T}_{\mathcal{A}}(I)$ such that, each level $k < n$ of the tree is composed of two sublevels. First is the sublevel of the branching nodes at level k followed by the sublevel of the input reading nodes at level k.

Formally the tree is defined recursively as follows:

- The root of the tree is at level 0 and is a branching state whose label is empty.
- A free branching state at level k has a label of type $(\mathcal{D}^k, \Sigma^k, \mathbb{N}^k)$.

  Suppose u_f is a free branching state at level $k < n$ whose label is:

$$(D_1^{u_f}, \dots, D_k^{u_f}, \sigma_1^{u_f}, \dots, \sigma_k^{u_f} g_1^{u_f}, \dots, g_k^{u_f})$$

  and suppose u_f makes the guess $g \in \mathbb{N}$, then u_f has g_{k+1} successors, which are input reading nodes $u_{r_1}, \dots, u_{r_{g_{k+1}}}$. The labels of the successors are of type

$(\mathcal{D}^k, \Sigma^k, \mathbb{N}^{k+1})$ and label of the $i-$th successor u_{r_i} is:

$$\begin{aligned}&(\mathcal{D}_1^{u_{r_i}}, \ldots, \mathcal{D}_k^{u_{r_i}}, \sigma_1^{u_{r_i}}, \ldots, \sigma_k^{u_{r_i}}, g_1^{u_{r_i}}, \ldots, g_{k+1}^{u_{r_i}})\\ &\quad = (\mathcal{D}_1^{u_f} \ldots, \mathcal{D}_k^{u_f}, \sigma_1^{u_f}, \ldots, \sigma_k^{u_f}, g_1^{u_f}, \ldots, g_k^{u_f}, i),\end{aligned}$$

for $i = 1, \ldots, g$.

- Let u_r be an input reading node at level $k < n$ with a label

  $$(\mathcal{D}_1^{u_r} \ldots, \mathcal{D}_k^{u_r}, \sigma_1^{u_r}, \ldots, \sigma_k^{u_r}, g_1^{u_r}, \ldots, g_{k+1}^{u_r}).$$

  Suppose $\mathcal{D}_{k+1}^{u_r}$ is the first data item from $I \setminus \{\mathcal{D}_1^{u_r}, \ldots \mathcal{D}_{k+1}^{u_r}\}$ according to the total order

  $$\pi_{\mathcal{A}}^k(\mathcal{D}_1^{u_r}, \ldots, \mathcal{D}_k^{u_r}, \sigma_1^{u_r}, \ldots, \sigma_k^{u_r}, g_1^{u_r}, \ldots, g_{k+1}^{u_r}),$$

  and suppose

  $$c^k(D_1^{u_r}, \ldots, D_{k+1}^{u_r}, \sigma_1^{u_r} \ldots, \sigma_k^{u_r},) = (c_1, \ldots, c_d, \perp, c_{d+1} \ldots)$$

  then, reading state u_r has d children. Each child is a free branching state at level $k + 1$. The children of u_r are $v_1, \ldots, v_d$, such that the label of v_i is

  $$\begin{aligned}&(\mathcal{D}_1^{v_i}, \ldots, \mathcal{D}_{k+1}^{v_i}, \sigma_1^{v_i}, \ldots, \sigma_{k+1}^{v_i}, g_1^{v_i}, \ldots, g_{k+1}^{v_i})\\ &\quad = (\mathcal{D}_1^{u_r}, \ldots, \mathcal{D}_{k+1}^{u_r}, \sigma_1^{u_r}, \ldots, \sigma_k^{u_r}, c_i, g_1^{u_r}, \ldots, g_{k+1}^{u_r})\end{aligned}$$

Note that the pFBT model subsumes the pBT model. The guesses used by the pFBT algorithm are integers. If the values of all guesses is 1, then every free branching state has exactly one child and then we have the pBT model.

The metric for measuring efficiency of the computation of a a pFBT algorithm is the same as the one used for pBT algorithms.

Lemma 20. *[Width bound lemma] If $\mathcal{A}$ is a pFBT algorithm for some problem Π such that $W_{\mathcal{A}}(I) < w$ then on all valid instances $I \in \Pi$ the pFBT tree $\mathcal{T}_{\mathcal{A}}(I)$ has width bounded by w: $|W_{\mathcal{T}_{\mathcal{A}}}(I)| \leq w$, for any $I \in \Pi$. Then w is a bound on the fan-out of any free branching state used by $\mathcal{A}$.*

3.3 Lower Bound for pBT Algorithms for 7-SAT

In this section we consider the 7-SAT problem. The 7-SAT problem is given a Boolean formula in CNF such that each clause has at most seven literals, find an assignment which satisfies all clauses. We will develop a lower bound technique which can be used to show negative results for all algorithms which fit the pBT and the pFBT models and will use it to establish a lower bound of $2^{\Omega(n)}$ on the width of any pBT algorithm for solving the 7-SAT.

The computation of a pBT algorithm on a valid instance of a problem creates a binary tree. Along each branch of the tree the algorithm learns information about the instance.

Definition 6. *(partial instance PI^{in}) The set PI^{in} is the set of all data items queried along a path of the tree and found to belong to the instance.*

Definition 7. *(partial instance PI^{out}) PI^{out} is the set of data items found to not be in the instance.*

The revealed partial information along a branch of the tree is $PI = PI^{in} \cup PI^{out}$. PS will denote the partial solution committed along a branch, which consists of decisions about items in PI^{in}. Let n be the size of an instance, hence each valid instance consists of n data items. The general technique for proving lower bounds is defined as follows.

1. Define a *finite* family of valid and hard instances, call it $\mathcal{F}_n$ and a probability distribution $\mathcal{P}$ on the instances in the family $\mathcal{F}_n$. Each valid instance in $\mathcal{F}_n$ must have a unique solution. The branches of the computation tree $\mathcal{T}_{\mathcal{A}}(I)$ of any pBT algorithm $\mathcal{A}$ on an instance $I \in_{\mathcal{P}} \mathcal{F}_n$ are divided into two types: the *good paths* and the *bad paths*. Let l be a parameter, which is the maximum number of data items read by the algorithm. The parameter l depends on the problem and the hardness of the instances. Let N be a parameter which is much larger than the number of non-items a pBT algorithm can typically learn, without examining

and committing to decisions on l data items. Branches in the tree $\mathcal{T}_{\mathcal{A}}(I)$ are uniquely identified by PI^{in}, PI^{out} and PS.

Definition 8 (good paths). *Good paths are those paths of $\mathcal{T}_{\mathcal{A}}(I)$, where the algorithm has learned fewer than N items, which do not belong to the instance (non-items), by the time it has committed to decisions for exactly l data items. Hence, if a path defined by the partial information PI_g^{in}, PI_g^{out} is good then $|PI_g^{in}| = l$ and $|PI^{out}| < N$.*

Definition 9 (bad path). *A path defined by PI_b^{in}, PI_b^{out} is bad if $|PI_b^{in}| < l$ and $|PI_b^{out}| \geq N$. Conversely by the time $|PI^{out}| = N$, the algorithm has read and committed to a decisions to fewer than l data items from the instance, $|PI^{in}| < l$.*

We analyze branches of the tree generated by any pBT algorithm $\mathcal{A}$ such that, the partial instance PI^n is of size at most l.

Definition 10 (consistent path). *Let p be a branch in $\mathcal{T}_{\mathcal{A}}(I)$, determined by the revealed information $PI^{in} \cup PI^{out}$ and the partial solution PS. We call p consistent with respect to an instance I if $PI^{in} \subseteq I$ and $PI^{out} \cap I = \emptyset$.*

Remark 21. We assume that the algorithms we analyze maintain only consistent paths, hence $\mathcal{T}_{\mathcal{A}}(I)$ contains only consistent branches, for all $\mathcal{T}_{\mathcal{A}}(I)$.

Let $I \in_{\mathcal{P}} \mathcal{F}_n$ be an instance and $S(I)$ be its unique solution.

2. Prove that along any good path the solution committed PS is unlikely to be consistent with the unique solution $S(I)$.

Lemma 22. *[Good path bound]*

Let $I \in_{\mathcal{P}} \mathcal{F}_n$. Let p_k be **any good** *path, defined by the sets PI_k^{in}, PI_k^{out}, and PS_k. Prove that:*

$$\Pr_{I \in_{\mathcal{P}} \mathcal{F}_n}\left[PS_k \text{ consistent with } S(I) \mid (PI_k^{in} \subseteq I) \wedge (PI_k^{out} \cap I = \emptyset)\right]$$
$$\leq O\left(\frac{1}{f_1(n)}\right).$$

3. When analyzing bad paths we need to establish that they are highly unlikely to exists and that their total probability is small.

   (a) To show that on a typical instance from the probability distribution bad paths are unlikely to exist we use a super decision. The super decision tree $\mathcal{T}_{\mathcal{A}}$ contains all trees $\mathcal{T}_{\mathcal{A}}(I)$, for all instances $I \in \mathcal{F}_n$. Each node in the super decision tree represents the query "Does data item d belong to the instance I ($d \in I$)?". There are two possibilities (hence the super decision tree is binary and each node has two children). The state of the algorithm is described by the pair (PI^{in}, PS). Each node of the super decision tree is described by tuple where the first element describes the current state of the algorithm; the second element describes the set of non-items observed: $((PI^{in}, PS), PI^{out})$. Initially the state of the algorithm is $(PI^{in}, PS) = (\emptyset, \emptyset)$ and $PI^{out} = \emptyset$. A data item can contradict the partial instance PI^{in}, PI^{out} if no valid instance $I \in \mathcal{F}_n$ exists which is consistent with the partial information PI^{in}, PI^{out}. A data item d contradicts the partial information PI^{in}, PI^{out} if there does not exists a valid instance I, such that $d \in I$ and I is consistent with PI^{in}, PI^{out}. If the data item is not contradictory, then it could either belong to the instance I or not. The root node of the super decision tree is described by the triple $((\emptyset, \emptyset), \emptyset)$. Suppose a decision node of the decision tree $\mathcal{T}_{\mathcal{A}}$ is described by the triple $((PI^{in}, PS), PI^{out})$.

      - A decision node with descriptor $((PI^{in}, PS), PI^{out})$ does the following:
        - Invokes the ordering function $\pi^{|PS|}(PI^{in}, PS)$;
        - Searches for the first data item in the order which is not contradictory with the partial information and is not part of the partial instance observed so far.

  while (($d_k \in \{PI^{in} \cap PI^{out}\}$) $\vee$ (d_k contradicts PI^{in}, PI^{out}))

  $k \leftarrow k + 1.$

  - makes the query "$d_k \in I$?".

  Each decision node has two children. Right edge leaving the node is labeled YES and corresponds to the query answer "$d \in I$", while left edge leaving the node is labeled NO and corresponds to the query answer "$d \notin I$".

- A *right child* of the node updates the state of the algorithm and invokes the appropriate choice function. PI^{out} remains unchanged and will be propagated to the next node.

  - State update:$PI^{in} \leftarrow PI^{in} \cup \{d_k\}$.
  - Choice function invocation: $c^{|PS|}(PI^{in}, PS)$. Say the two choices are c_1, c_2.

  A right child node will then simulate the algorithm and will have two children which are decision nodes whose description is:

  - $((PI^{in}, PS \leftarrow PS \cup \{d_k = c_1\}), PI^{out})$
  - $((PI^{in}, PS \leftarrow PS \cup \{d_k = c_2\}), PI^{out})$

  Recurse on each decision child[4].

- A *left child* does not change the state of the algorithm, but updates PI^{out}.

  - $PI^{out} \leftarrow PI^{out} \cup d_k$.

A path is bad if $|PI^{in}| < l$ and $|PI^{out}| \geq N$. A path is good if $|PI^{in}| = l$ and $|PI^{out}| < N$. The number of bad paths in the super decision tree $\mathcal{T}_\mathcal{A}$ is a pessimistic upper bound of the number of bad paths in any $\mathcal{T}_\mathcal{A}(I)$ because some bad paths in $\mathcal{T}_\mathcal{A}$ would not be consistent with the specific instance I.

[4]invoke ordering function $\pi^{|PS|}(PI^{in}, PS)$ and make decision about 1-st data item in the corresponding list of data items.

Lemma 23. *[Any fixed bad path is unlikely to exist]*
Let $I \in_{\mathcal{P}} \mathcal{F}_n$. Let p_b be **any bad** *path. Let p_b be defined by sets PI_b^{in}, PI_b^{out}. Prove that*

$$\Pr_{I \in_{\mathcal{P}} \mathcal{F}_n} \left[(|PI_b^{out}| > N) \wedge (PI_b^{in} \subseteq I) \wedge (PI_k^{out} \cap I = \emptyset) \right] \leq f_2(n).$$

(b) Show that the total probability of bad paths is small.

3.3.1 Proof Framework

We will define the family of hard formulas $\mathcal{F}_n$, the sampling distribution on $\mathcal{F}_n$, the good and the bad paths, and will setup the framework of the proof. We define the decision tree associated with a pBT algorithm[5] which we use in the analysis and will relate it to the computation tree of the algorithm.

[1] used uniquely satisfiable 3-SAT formulas encoded by a linear system of $\bmod 2$ equations $Ax = b$, where $A \in \{0,1\}^{n \times n}$ is a non-singular matrix and a very strong expander and b is a uniformly chosen binary vector of dimension n. Similarly as in [1] the hard formulas we use for the lower bound are 7-SAT instances with unique solutions defined by linear system over expanders. To define $\mathcal{F}_n$ we need a matrix $M \in \{0,1\}^{n \times n}$ which has to satisfy all of the following conditions:

1. M is non-singular (has rank n).

2. Each row sums to seven, hence each row has exactly seven ones and exactly $(n-7)$ zeros.

3. Each column of M sums up to at most K, where K is a constant and does not depend on n.

4. M is $(r, 7, c)$ boundary expander, with $c > 1$ and $r \in \Theta(n)$:

[5]The decision tree of each algorithm is unique, and contains as a subtree the pBT tree generated by the algorithm on any instance. Later we will relate the two trees.

Definition 11. *(boundary expander) A $(r, 7, c)$ boundary expander*[6] *is a binary matrix in which each row contains exactly 7 ones and $(n-7)$ zeros, and satisfies the following property: for every set of rows I, such that $|I| \leq r$, the size of the unique neighbors of I called the boundary of I, denoted as $\partial_M(I)$, is at least $c|I|$ (an element is in the boundary of I, ∂I, if the element has exactly one neighbor in I).*

In Section 3.5 we prove that a square non-singular binary matrix M which satisfies all of the above conditions exists and it is $(r, 7, 3)$ boundary expander.

From now on we assume that $M \in \{0,1\}^{n \times n}$ is a fixed non-singular matrix, which is also $(r, 7, 3)$ boundary expander, where $r \in \Theta(n)$ and each column of M sums up to a constant K. Once M is fixed it is used to define all instances in the family $\mathcal{F}_n$. Perhaps it will be more appropriate to denote the family by $\mathcal{F}_n(M)$ but for shortness of we will use $\mathcal{F}_n$ instead. Let $b \in_u \{0,1\}^n$ be an arbitrary vector, then the system $Mx = b$ encodes a 7-SAT instance in the following way. Each row defines one $\mod 2$ equation which is satisfied by half (64 of the 128) of all clauses. All the clauses that satisfy the equation belong to the formula. Thus each variable of the formula appears in equal number of clauses in negative and positive orientation. A data item corresponds to a variable, and is defined by the index of the variable, and a list of all clauses in which it participates in. Consider the linear system $Mx = b$. The data item corresponding to variable x_i is defined by the i-th column vector of M and reveals those bits b in which the corresponding column vector has a one. Because each column of M sums up to K, then each data item is determined by the exact value of at most K bits of b and a variable reveals at most K bits of b. A data item is describable as a pair $d = (i, g)$, where the first component $i \in [n]$ identifies the variable, so here d describes x_i. The second component is a vector $g \in \{0, 1, X\}^n$, where $(g_1, \ldots, g_n)$, describe the guesses for the corresponding components of b. Since each column has at most K ones, then g

[6] *If M was just an expander then it would have had to satisfy the weaker expansion condition that any set of rows I such that $|I| \leq r$ the neighbors of I are at least $c|I|$: any set of row I, such that $|I| \leq r$ the neighbors of I are at least $c|I|$ (no uniqueness requirement).*

should have X in at least $(n - K)$ positions and at most K zeros or ones. A data item $d = (i, g)$ is consistent with the matrix M and hence belongs to $\mathcal{D}$ if:

1. for all $j \in [n]$ if $(M)_{j,i} = 0$ then $b_j = X$;

2. for all $j \in [n]$ if $(M)_{j,i} = 1$ then $b_j \in \{0, 1\}$.

The set of all data items $\mathcal{D}$, needed to describe all formulas in $\mathcal{F}_n$ has cardinality $|\mathcal{D}| \leq 2^K n$. The family of hard formulas $\mathcal{F}_n$, contains all formulas defined by $\mod 2$ equations in the system $Mx = b$, and $b \in_u \{0, 1\}^n$, hence $|\mathcal{F}_n| = 2^n$. The distribution on instances we use is the uniform and we choose an instance I at random from $\mathcal{F}_n$ by choosing a vector $b \in_u \{0, 1\}^n$. We assume that $\mathcal{D}$ contains only data items consistent with the matrix M hence if $(i, g) \in \mathcal{D}$ then $\forall i \in [n] \ \ (g_i = X) \Leftrightarrow ((M)_{i,j} = 0)$. If $g_i = X$ then the data item does not depend on the i-th bit of b. Otherwise $g_i \in \{0, 1\}$ and the guess bit g_i is compared against the real bit b_i. If $((M)_{i,j} = 1) \wedge (g_i \neq X) \wedge (g_i = b_i)$ then the guessed correct, otherwise it is not $((M)_{i,j} = 1) \wedge (g_i \neq X) \wedge (g_i \neq b_i)$. A data item can contradict the current knowledge of vector b, or if it does not, then it can belong to the instance, or not. Select an instance $I \in_u \mathcal{F}_n$ by randomly choosing $b \in_u \{0, 1\}^n$. I is encoded by the linear system $Mx = b$, and the unique solution is $M^{-1}b$. A data item belongs to a 7-SAT instance I if it is consistent with vector b. A data item $d = (j, (B_{j,1}, \ldots, B_{j,n}))$ is consistent with b if and only if for all $k \in [n]$ the following holds true:

1. $((M)_{k,j} = 0) \Rightarrow (B_{j,k} = X)$

2. $((M)_{k,j} = 1) \Rightarrow (B_{j,k} = b_k)$

The first condition says that the data item must be valid and should agree with the matrix M, while the second condition requires all guesses for the corresponding bits of vector b to be correct.

We assume that a pBT algorithm knows the matrix M but does not know the vector b. The distribution on instances we use is the uniform distribution on the formulas defined by $Mx = b$, for $b \in_u \{0,1\}^n$. A data item describing variable x_i is determined exactly by the value of at most K bits of b, corresponding to the rows in which the i-th column vector of M has values 1. Since b is chosen uniformly at random from $\{0,1\}^n$, then given the matrix M, the probability of a data item is at least $\frac{1}{2^K}$. Any pBT algorithm for the 7-SAT instances will be specified by the ordering and choice functions as in Section 3.1, Definition 5. We need to establish a lower bound on the expected width of the tree $\mathcal{T}_{\mathcal{A}}(I)$, where $I \in_u \mathcal{F}_n$. The algorithm $\mathcal{A}$ is specified by defining all choice and ordering functions and the computation of A on an instance I generates a binary tree (see Definition 5), call it $\mathcal{T}_{\mathcal{A}}(I)$. We consider the tree $\mathcal{T}_{\mathcal{A}}(I)$ at depth l, such that along all branches exactly l variables of I are assigned values. To analyze paths in $\mathcal{T}_{\mathcal{A}}(I)$ we define a super decision tree $\mathcal{T}_B$, associated with a pBT algorithm $\mathcal{A}$, for learning the bits of vector b. Each data item is represented by a tuple of the type (j, g), where j is a selector for the column vector of M, and g is an n-vector where $g_i \in \{0, 1, X\}, \forall i \in \{1, \dots, n\}$.

Let $b = (b_1, \dots, b_n)^T$ be the vector defining the instance formula ($Mx = b$), let $\overline{b} = (\overline{b}_1, \dots, \overline{b}_n)^T$ denote the bits of vector b learned by the algorithm. Initially none of the bits of b are known and $\overline{b} = (\perp, \dots, \perp)^T$. A data item $d = (j, g)$:

- is *contradictory* with respect to the partial knowledge $\overline{b}$ if:
  $\exists k \in [n]$ such that $((M)_{k,j} = 1) \wedge (\overline{b}_k \neq \perp) \wedge (g \neq X) \wedge (\overline{b}_k \neq g_k)$. If an item is contradictory then it cannot belong to the instance.

- *belongs to the instance* if for all $k \in [n]$:

  1. $((M)_{k,j} = 0) \Leftrightarrow (g_k = X)$;

  2. $((M)_{k,j} = 1) \Rightarrow (g_k = b_k)$.

- *does not belong to the instance* if both conditions hold:

  1. $\forall k \in [n]\ ((M)_{k,j} = 0) \Leftrightarrow (g_k = X)$;

  2. $\exists k \in [n]\ |((M)_{k,j} = 1)(g_k \neq b_k)$.

Let (PI^{in}, PS) be the state of the algorithm. Let PI^{out} be the set of data items confirmed to not be in the instance. The super decision tree $\mathcal{T}_{\mathcal{B}}$ has two kinds of nodes *query* and *algorithmic*. A *query* node maintains $\overline{b}$, (the current knowledge of the algorithm about the instance I). An *algorithmic* node simulates the algorithm $\mathcal{A}$ and commits assignments to variables. A node of $\mathcal{T}_{\mathcal{B}}$ is described by a tuple $((PI^{in}, PS), PI^{out}, \overline{b})$. The root node of the tree $\mathcal{T}_{\mathcal{B}}$ has descriptor $((\emptyset, \emptyset), \emptyset, (\perp, \ldots, \perp))$.

- A *query* node with descriptor $((PI^{in}, PS), PI^{out}, \overline{b})$ does the following:

  1. Invokes the ordering function $\pi^{|PS|}(PI^{in}, PS) = (d_1, d_2, \ldots, d_{|\mathcal{D}|})$;
     $k \leftarrow 1$;
  2. while $(d_k$ contradicts $\overline{b}) \vee (d_k \in \{PI^{in} \cup PI^{out}\})$ k++;
     Let $d_k = (j, (g_1, \ldots, g_n))$; $i \leftarrow 1$;
  3. while $(i \leq n) \ \wedge \ ((g_i = X) \vee (g_i = \overline{b}_i))$ i++;
  4. branch on "$(g_i == b_i)$?"

     (a) YES: label right edge "YES"
     if $(g_i$ is the last guessing bit of $g)$[7]

     (i) **then** Right child is an *algorithmic* node with descriptor
     $((PI^{in} \circ \ d_k, PS), PI^{out}, \overline{b}_{|_{\overline{b}_i \leftarrow g_i}})$.

     (ii) **else** Left child is a *query* node with descriptor
     $((PI^{in}, PS), PI^{out}, \overline{b}_{|_{\overline{b}_i \leftarrow g_i}})$.

     (b) NO: label right edge "NO"
     Left child is a *query* node with descriptor
     $((PI^{in}, PS), PI^{out} \cup \{d_k\}, \overline{b}_{|_{\overline{b}_i \leftarrow \neg g_i}})$.

- An *algorithmic* node

  1. Calls the choice function $c^{|PS|}(PI^{in}, PS)$.
     – If the choice function is empty, then the branch is terminated.

[7]This happens when either $i > n$, or $i < n$ and $g_j = X$, for all $j \in [i..n]$.

- If the choice function has one possibility, call it V, then create one query node as a child with descriptor $((PI^{in}, PS \circ \{x_j = V\}), \overline{b})$, label the edge $(x_j = V)$.
- If the choice function specifies both assignments, T and F, then create two children:
  * Right child with descriptor $((PI^{in}, PS \circ \{x_j = T\}), \overline{b})$ and label the edge $(x_j = T)$;
  * Left child with descriptor $((PI^{in}, PS \circ \{x_j = F\}), \overline{b})$ and label the edge $(x_j = F)$;

To analyze the tree $\mathcal{T}_{\mathcal{A}}(I)$ at depth l we consider the branches of the tree $\mathcal{T}_{\mathcal{B}}$ of length r or less. A data item which belongs to the instance reveals at most K bits for two reasons. First, each column vector of M contains at most K ones, therefore at most K bits defining a data item are guessing bits. Second, as more bits of $\overline{b}$ become set, a data item might be defined by some bits already known, hence such a data item will reveal fewer bits. A data item (l, g) which could have been in the instance but it is not (hence belongs to PI^{out}) would reveal one bit, if the first bit queried at the branching point disagrees. It could reveal (in the extremely rare case) K bits when it depends on exactly K bits of b, and none of the bits have been queried before, and all but the last guessing bits agree with b. In this unlikely case the algorithm will learn K bits of b from an item which belongs to PI^{out}, a.k.a non-item. The probability of an item which belongs to the instance is at least $\frac{1}{2^K}$ therefore the probability of a non-item is at most $1 - \frac{2}{K}$. Paths in $\mathcal{T}_{\mathcal{B}}$ are good or bad based on which of the following two events happens first:

- (Event 1) There are l algorithmic nodes along the path, therefore at the last algorithmic node $|PI^{in}| = |PS| = l$.
- (Event 2) r bits of $\overline{b}$ are set (to zero or one), and since a non-item reveals at most K bits, then the number of non-items along the branch are at least N and $|PI^{out}| \geq \frac{r}{K} = N$.

Definition 12. *(good path) A path is good when Event 1 happens before Event 2, therefore a good path in $\mathcal{T}_{\mathcal{B}}$ is characterized by $|PI^{in}| = |PS| = l$, $\overline{b}$ has fewer than r bits set, and $|PI^{out}| < N$.*

Definition 13. *(bad path) A path is bad if Event 2 happens before Event 1. Along a bad path in $\mathcal{T}_{\mathcal{B}}$ we have $|PI^{out}| \geq N$, $|PI^{in}| = |PS| < l$.*

Next we establish relations between paths in the trees $\mathcal{T}_{\mathcal{B}}$ and $\mathcal{T}_{\mathcal{A}}(I)$.

Lemma 24. *Let g_1 be a good path in $\mathcal{T}_{\mathcal{B}}$ and suppose g_1 is consistent with the instance I. Let (PI_l^{in}, PS_l) be the state of the algorithm at the time exactly l variables (data items) have been assigned values along g_1. Then there exists a path g at level l in $\mathcal{T}_{\mathcal{A}}(I)$, such that the state of the algorithm at that point is (PI^{in}, PS).*

Proof. Let $I = i_1, \ldots, i_n$, be the input instance and $l < n$. Since g_1 is good in $\mathcal{T}_{\mathcal{B}}$ then it contains l algorithmic nodes. For $i = 1, \ldots, l$ let (PI_i^{in}, PS_i) be the state of the algorithm at the l algorithmic states along g_1. We claim that there is a path g in $\mathcal{T}_{\mathcal{A}}(I)$ of length l such that the state of the algorithm at each state in the tree $\mathcal{T}_{\mathcal{A}}(I)$ along the path is (PI_m^{in}, PS_m), for $m = 1, \ldots, l$. The proof is by induction on i. The root node of the tree $\mathcal{T}_{\mathcal{B}}$ has descriptor $((\emptyset, \emptyset), \emptyset, \overline{b})$, and will invoke the ordering function $\pi^0(\emptyset, \emptyset) = \mathcal{O}(\mathcal{D})$, which totally orders $\mathcal{D}$. Let i_{i_1} be the first data item from I in the order $\mathcal{O}(\mathcal{D})$ and let $\pi^0(\emptyset, \emptyset) = (d_1, \ldots, d_m, i_{i_1}, \ldots)$. Then in the tree $\mathcal{T}_{\mathcal{B}}$ the path g_1 will have a sequence of query nodes which would process each data item d_i whose position in the order π^0 is before i_{i_1}. Since $d_i \notin I$ then they contradict the vector b defining I and none of the items d_i will be assigned values. The query node which will process data item i_{i_1} will have a descriptor $(\emptyset, \emptyset), PI^{out}, \overline{b})$. Since $i_{i,1} \in I$ then all guess bits defining $i_{i,1}$ agree with b and some query node processing i_{i_1} along the path g_1 in $\mathcal{T}_{\mathcal{B}}$ will reach point **4.(a).(i)** and will create an algorithmic state with descriptor $((i_{i_1}, \emptyset), PI^{out}, \overline{b})$. The state will invoke the choice function $c^0(i_{i_1}, \emptyset)$. Since g_1 is not terminated then it will correspond to some decision $V \subseteq \{c^0(i_{i,1}, \emptyset)\}$. So the first algorithmic state of g_1 in $\mathcal{T}_{\mathcal{B}}$ will create a query state with descriptor $((\{i_{i,1}\}, \{V\}), \{d_1, \ldots, d_k\}, \overline{b})$. Now let us consider the the root node of the tree $\mathcal{T}_{\mathcal{A}}(I)$. It will call the same ordering function $\pi^0(\emptyset, \emptyset)$, since $i_{i,1}$ is the first data item of I in the order then the state will set the partial instance $PI_1^{in} = \{i_{i,1}\}$, and will invoke choice function $c^0(i_{i,1}, \emptyset)$, which is the same as the choice function called by the first algorithmic node along g_1. Hence the choice $V \subset T, F$ will have to be explored in $\mathcal{T}_{\mathcal{A}}(I)$ as well. Continuing the process we reach the last algorithmic state of g_1 which will have some descriptor $((PI_l^{in}, PS_l), PI^{out}, \bar{b})$ and (PI_l^{in}, PS_l) since both paths call the same ordering and choice functions. $\square$

Lemma 25. *Let g_1 and g_2 be two good distinct paths in $\mathcal{T}_{\mathcal{B}}$ consistent with the instance I. Then both correspond to paths in $\mathcal{T}_{\mathcal{A}}(I)$ ending at different nodes at level l.*

Proof. Since both paths are good, then they have committed values of l variables. By Lemma 24 they will correspond to a node at level l in the tree $\mathcal{T}_{\mathcal{A}}(I)$. Since g_1 and g_2 are distinct in $\mathcal{T}_{\mathcal{B}}$, then they split somewhere. If they split at a left query node then one of them is not consistent with I, but they are consistent with I, therefore that cannot happen. Otherwise g_1 and g_2 split at a right node. That implies that the algorithm $\mathcal{A}$ has committed to different decisions, hence g_1 and g_2 have distinct partial solutions say $PS_1, PS_2, PS_1 \neq PS_2$, hence they correspond to different nodes at level l in $\mathcal{T}_{\mathcal{A}}(I)$. $\square$

Corollary 26. *The number of good paths consistent with an instance I in $\mathcal{T}_{\mathcal{B}}$ is at most the width of the algorithm $\mathcal{A}$, $W_{\mathcal{A}}(n)$.*

Lemma 27. *Every path at level l of the tree $\mathcal{T}_{\mathcal{A}}(I)$ has a corresponding path in $\mathcal{T}_{\mathcal{B}}$ (consistent with I).*

We characterize the branches of the tree $\mathcal{T}_{\mathcal{B}}$ according to how many bits of vector b the algorithm has learned along the path. Intuitively along a good path the algorithm has fewer than N bits and has assigned values to l variables. Along a bad path the algorithm has learned N bits but has committed to values of fewer than l variables. When we analyze the good paths, we use the hardness of the instances in $\mathcal{F}_n$ and show that learning fewer than r bits of b, reveals insufficient information about the unique solution and there are exponentially many solutions consistent with the revealed partial information. The unique solution is $M^{-1}b$ and depends on all bits of b.

Definition 14. *(intermediary) A branch is called an intermediary if at most l variables are committed along it and the bits of vector b revealed along it are consistent with the instance.*

Definition 15. *(good intermediary) A branch is called a good intermediary if it is an intermediary, i.e., exactly l variables are committed, and the number of bits of b learned is less than or equal to r.*

Definition 16. *(bad intermediary) A branch is called a bad intermediary if the algorithm has learned the values of r_0 bits of b but has committed the values of fewer than l variables along it.*

Note that, the system $Mx = b$ has always a solution, which is unique because M is non-singular. Hence, for any pBT tree built by any pBT algorithm we have:

$$\Pr_{b \in_u \{0,1\}^n} \left[\exists \text{ alive branch consistent with } M^{-1}\vec{b} \right] \leq 1.$$

If the algorithm is correct then this probability will be one. Note that we are evaluating the probability of a good and bad intermediary already in the reduced sample space of partial paths that corresponding to partial instances consistent with the final instance.

$$\begin{aligned} 1 \quad &\leq \Pr_{b \in_u \{0,1\}^n} \begin{bmatrix} \exists \text{ alive branch} \\ \text{consist. with } M^{-1}b \end{bmatrix} \leq \Pr_{b \in_u \{0,1\}^n} \begin{bmatrix} \exists \text{ alive intermediary} \\ \text{consist. } M^{-1}b \end{bmatrix} \\ &\leq \Pr_{b \in_u \{0,1\}^n} \begin{bmatrix} \exists \text{ bad intermediary} \\ \text{consist. with } M^{-1}b \end{bmatrix} + \Pr_{b \in_u \{0,1\}^n} \begin{bmatrix} \exists \text{ good intermediary} \\ \text{consist. with } M^{-1}b \end{bmatrix}. \end{aligned}$$

3.3.2 Analysis of Good Intermediaries

Along a good intermediary the algorithm has learned fewer than r bits of b and has committed to the values of exactly l variables. To show that any good intermediary, has an exponentially small probability of being consistent with the unique solution we will use the hardness of the hard formulas in $\mathcal{F}_n$. Suppose that the algorithm has examined l data items and has learned in total $r_0 < r$ bits of $\vec{b}$. Let $b_1, \ldots, b_{r_0}$ be the partial information about $\vec{b}$ known to the algorithm. We consider a fixed branch of the pBT tree, defined by the committed decisions for the variables examined along the branch, say those are $x_1, \ldots, x_l$, and the partial knowledge of $\vec{b}$ is $b_1, \ldots, b_{r_0}$.

Lemma 28. *Let M be $(r, 7, c)$ be boundary expander with $c \geq 3$. Consider the system $Mx = b$, where $b \in_u \{0,1\}^n$, $x_1, \ldots, x_l$ be a partial solution committed by the algorithm and $b_1, \ldots, b_{r_0}$ be the bits from b known to the algorithm where $r_0 < r$. Then*

$$\Pr_{\vec{b} \text{ consistent with } b_1, \ldots, b_{r_0}} \left[x_1, \ldots, x_l \text{ consistent with } M^{-1}\vec{b} \right] \leq 2^{-\Omega(l)}.$$

Proof.

$$\Pr_{\substack{\vec{b} \text{ consist.} \\ \text{with } b_1, \ldots, b_{r_0}}} \left[x_1, \ldots, x_l \text{ consist. } M^{-1}\vec{b} \right] = \Pr_{\substack{\vec{x} \text{ consis. with} \\ Ax = (b_1, \ldots, b_{r_0})^T}} \left[\vec{x} \text{ consist. } x_1, \ldots, x_l \right].$$

Consider the $\mod 2$ equations defined by the r_0 bits of $\vec{b}$ known to the algorithm and let the corresponding rows of M be denoted as I, $|I| = r_0$. We need to find the number of solutions to the subsystem of equalities $Ix = (b_1, \ldots, b_{r_0})^T$, consistent with a specific value of $x_1, \ldots, x_l$, committed by the algorithm ($l \leq r_0$). We seek to find the column rank of $\{x_1, \ldots, x_l\}$. In order to estimate the column rank of $x_1, \ldots, x_l$ in I, we reduce the system from I equations to I_0 equations, such that column rank of $\{x_1, \ldots, x_l\}$ in I and the column rank $\{x_1, \ldots, x_l\}$ in I_0 are the same. We do that by eliminating equations one at a time. Let $Fixed = \{x_1, \ldots, x_l\}$ be the set of the variables whose assignment is fixed along the path. The boundary variables of I is denoted by $\partial(I)$. For a boundary variable $x_i \in \partial(I)$, let I_i be the row in which x_i appears.

```
GAUSSIAN-LIKE ELIMINATION OF ROWS
$I_0 \leftarrow I$
while   $(cr_0 > l)$
        1.    pick $x_B \in \partial(I_0) \setminus Fixed$
        2.    $x_B \longleftarrow$ value that satisfies $(I)_B \cdot x_B = (b)_B$
        3.    $I_0 \longleftarrow I_0 - (I)_B$
        4.    $r_0 \longleftarrow r_0 - 1.$
```

Each iteration of the loop selects a boundary variable x_B, whose boundary status is determined with respect to current set I_0, not the original I. The boundary variable x_B occurs in exactly one row and $x_B \notin Fixed$. Each iteration of the procedure maintains the invariant that $column\_rank\{x_1, \ldots, x_l\}$ in I_0 does not change, hence at the end it is the same as the $column\_rank\{x_1, \ldots, x_l\}$ in I. Consider the $\mod 2$ equation defined by row I_{x_B}. Because x_B does not appear anywhere else, then there are no restriction on its value and we can set it anyway it is needed to satisfy the equation. Since $c > 1$, and $r_0 \leq r$, we can always find a boundary variable in the body of the loop, because $|\partial(I_0)| \geq c|I_0|, c > 1$.

The Gaussian-like elimination of rows terminates when $c \cdot r_0 \leq l$. Consider the reduced system $I_0 \cdot x = b_0$, after the Gaussian-like elimination. Let $b_0 = (b_1, \ldots, b_{r_0})$ denote the bits whose corresponding to the equations that have survived the Gaussian-like elimination. The reduced system has l boundary variables, which appear in exactly one equation and has a full row rank, which is $r_0 \leq \frac{l}{c}$. Then the number of solutions to the reduced system is at least $2^{l-\frac{l}{c}}$. None of the fixed variables $Fixed = \{x_1, \ldots, x_l\}$ have been eliminated, hence the particular assignment to the variables $\{x_1, \ldots, x_l\}$ committed by the algorithm is one of the $2^{l-\frac{l}{c}}$ possible assignments and has probability $2^{-(l-\frac{l}{c})}$.

$$\Pr_{\vec{b}\text{ consistent with }b_1,\ldots,b_{r_0}} \left[x_1, \ldots, x_l \text{ consistent with } M^{-1}\vec{b} \right] \leq 2^{-\frac{l(c-1)}{c}}. \tag{3.2}$$

But M is $(r, 7, 3)$ boundary expander, hence $c = 3$, and the probability of any particular assignment is at most $2^{-2l/3} = 2^{-\Omega(l)}$.

(Lemma 28) □

3.3.3 Analysis of Bad Intermediaries

A path in the tree is a bad intermediary if the branch is consistent with the instance and $|PI^{in}| \leq l-1$ and $|PI^{out}| \geq N$. Bad intermediaries are paths in the tree $\mathcal{T}_{\mathcal{B}}$ of length r with at most $l-1$ variables committed. We argue that the probability of a bad path is exponentially small.

Let (i, g) be one data item, where $g \in \{0, 1, X\}^n$. Let the number of $\{0, 1\}$ components of g be $k \leq K$ (where K is the maximum column sum of M). If (i, g) is in conflict with $\bar{b}$, then it cannot be in the instance and the algorithm learns nothing. On the other hand if a data item is in the instance the algorithm learns at most $k < K$ bits of b, because some bits could have already been revealed. The probability of such data item is at least $\frac{1}{2^K}$. A data item that could have been in the instance but is not reveals between 1 and at most[8] $k < K$ bits of b. Such an item has a probability at most $1 - \frac{1}{2^K}$. Let N be the number of data items examined, $N = \frac{r}{K}$ ($r \in \Theta(n)$ is the 1-st parameter of the boundary expander M). Let $D_1, D_2, \ldots, D_N$ be indicator random variables defined as follows:

$$D_i = \begin{cases} 1, & \text{if the i-th data item examined is in the instance,} \\ 0, & \text{otherwise.} \end{cases}$$

Conditioning on $\{D_1, \ldots, D_{i-1}\}$, data item D_i is not independent and the dependence is captured by the matrix A. However, note that $\Pr[D_i = 1] \geq \frac{1}{2^K}$, for all i. Our goal is to prove that it is unlikely that the algorithm has learned more than the expected number of bits of $\vec{b}$. A bad branch in the tree $\mathcal{T}_{\mathcal{B}}$ defines a walk of length r, hence at least $N = \frac{r}{K}$ non-items are seen along it and $l-1$ or fewer variables are set. Fix an assignment for the variables in PI^{in}. The assignment would determine the direction of the walk when we reach an algorithmic node in $\mathcal{T}_{\mathcal{B}}$, F for the left child and T for the right child (encoded as 0 and 1, respectively). The path is bad and at the time when the N-th data item was examined at most $l-1$ variables were committed and $|PI^{in}| \leq l-1$ We express the

[8]Revealing k bits is extremely unlikely and will happen if none of the k bits has been revealed and all but the last guess bits of g agree with b.

number of data items found as the sum of the indicator variables $\sum_{i=1}^{N} D_i$. When the branch of the tree is a bad intermediary then many of the indicators evaluated to 0, at most $l-1$ evaluated to 1, hence $\sum_{i=1}^{N} D_i < l$. If $l < \frac{\epsilon N}{2^K}$ then the probability of a bad intermediary is

$$\Pr\left[\sum_{i=1}^{N} D_i < l\right] < \Pr\left[\sum_{i=1}^{N} D_i \leq \frac{\epsilon N}{2^K}\right].$$

Lemma 29. *[Probability of a fixed bad intermediary]*
Assume that $l < \frac{\epsilon N}{2^K}$ then the probability of a bad branch is at most

$$\Pr_{b\in\{0,1\}^n}\left[\sum_{i=1}^{N} D_i < l\right] < \Pr_{b\in\{0,1\}^n}\left[\sum_{i=1}^{N} D_i < \frac{\epsilon N}{2^K}\right] < e^{-\Omega(N)}.$$

Proof. Define

$$Z_i = \sum_{j=1}^{i} D_j.$$

Note that (Z_i) forms a submartingale because

$$\begin{aligned} \mathrm{E}[\, Z_i \mid D_0, \ldots, D_{i-1}\,] \quad &= \mathrm{E}[\, Z_{i-1} \mid D_0, \ldots, D_{i-1}\,] + \mathrm{E}[\, D_i \mid D_j, j < i\,] \\ &= Z_i + \mathrm{E}[\, D_i \mid D_j, j < i\,] \geq Z_i. \end{aligned}$$

Further note that $|Z_i - Z_{i-1}| = D_i$, hence $|Z_i - Z_{i-1}| \in [0,1]$, $Z_0 = 0$, and $\mathrm{E}[\, D_i \mid D_j, j < i\,] \geq \frac{1}{2^K}$. Therefore the conditions of Theorem 50 are satisfied, with $\mu_i = \frac{1}{2^K}, \forall i$ and $\mu = \sum_i^N \mu_i = \frac{N}{2^K}$. By Theorem 50 for $\delta \in (0,1)$ we have

$$\Pr_{b\in\{0,1\}^n}\left[Z_N < (1-\delta)\frac{N}{2^K}\right] < e^{-\frac{\delta^2 \mu}{2}} = e^{-\frac{\delta^2 N}{2^K}}.$$

□

In our case $N = \frac{r}{K}$, and $r \in \Theta(n)$.

Lemma 30. *[Total probability of a bad intermediary]*

$$\Pr_{b\in_u\{0,1\}^n}[\,\exists \textit{ bad intermediary}] = o(1).$$

Proof. Recall that n is the number of variables in the formula, l is the number of variables committed along a good path and r is the parameter of the expander matrix M. M is $(r, 7, 3)$ boundary expander and $r \in \Theta(n)$, $N = \frac{n}{K}$, where K is a large constant which does not depend on n. We guess that $l = \epsilon_1 n$, where $\epsilon \in (0, 1)$ is also a small constant and does not depend on n.

To bound the probability that there exist a bad living intermediary we use a union bound. We need an upper bound on the number of bad intermediaries. Consider the super decision tree $\mathcal{T}_{\mathcal{B}}$, associated with any pBT algorithm $\mathcal{A}$. Each node in the super decision tree checks the guess bits of a data item against the real values of vector b. If all guesses agree then the data item belongs to the instance the set PI^{in} is updated and the algorithm $\mathcal{A}$ is simulated, and assignment for x_i is committed. Otherwise the data item (i, g) does not belong to the instance because some guess bit of g disagrees with b. The number of new bits of b learned is at lest 1 and at most K.

Note that the computation of the pBT algorithm on any particular instance I is a subtree of the super decision tree $\mathcal{T}_{\mathcal{B}}$. Hence the number of bad paths in any pBT tree $\mathcal{T}_{\mathcal{A}}(I)$ is at most the number of bad paths in $\mathcal{T}_{\mathcal{B}}$ which are consistent with I. We bound the total number of bad paths in $T_{\mathcal{B}}$, by counting both paths consistent with the instance and those which are not. A bad path has r bits of b read and fewer than l variables committed along it or N data items inspected and fewer than l variables committed. In the martingale in Lemma 29, we fixed the assignment of the variables found to belong to the instance. For a fixed assignment to the variables, by Lemma 29 the probability of a path of length greater or equal to N with fewer than l variables committed is $e^{-\frac{\delta^2 N}{2^K}}$. Since there are at most $l - 1$ variables assigned along a bad paths, hence the possible number of assignments is at most 2^{l-1}, therefore by a union bound

$$\begin{aligned}\Pr_{b \in_u \{0,1\}^n}[\,\exists \text{ bad intermediary}] &\leq 2^l \cdot \Pr_{b \in_u \{0,1\}^n}[Z_N < (1-\delta)\frac{N}{2^K}] \\ &\leq 2^l \cdot e^{-\frac{\delta^2 N}{2^K}} = 2^l \cdot e^{-\frac{\delta^2 N}{2^K}}.\end{aligned}$$

Choose $l = \frac{\delta^2 r}{10K2^K}$, then $l < \frac{\delta^2 N}{2^K} = \frac{\delta^2 r}{K2^K}$. In our case the boundary expander M has

$r < n, r \in \Theta(n)$, therefore the probability that there exists a bad intermediary is $o(1)$:

$$\Pr_{b \in_u \{0,1\}^n} [\exists \text{ bad intermediary}] \leq 2^{\left(\frac{\delta^2 r}{10K2^k}\right)} \cdot e^{-\left(\frac{\delta^2 r}{K2^K}\right)} = o(1).$$

□

Theorem 31 (pBT width lower bound for 7-SAT). *Any pBT algorithm $\mathcal{A}$ solving the 7-SAT problem on instances selected uniformly at random from $\mathcal{F}_n$ requires width $W_{\mathcal{A}}(n) \geq 2^{\Omega(n)}$.*

Proof. Let $W_{\mathcal{A}}(n)$ be the width of any pBT algorithm $\mathcal{A}$ on instances in $\mathcal{F}_n$. Let $I \in_u \mathcal{F}_n$, where I is defined by $Mx = b$, with unique solution $M^{-1}b$. We consider paths of length l as above, $l < n, l \in \Theta(n)$ in $\mathcal{T}_{\mathcal{A}}(I)$. In the sequence of inequalities below "cont. w." stands for "consistent with".

$$\begin{aligned}
1 &\leq \Pr_{b \in_u \{0,1\}^n} \left[(\exists \text{ path } p \in \mathcal{T}_{\mathcal{A}}(I)) \wedge (p \text{ const. w. } M^{-1}b) \wedge (p \text{ const. w. } I)\right] \\
&= \Pr_{b \in_u \{0,1\}^n} \left[(\exists \text{ path } p \in \mathcal{T}_{\mathcal{B}}) \wedge (p \text{ const. w. } M^{-1}b) \wedge (p \text{ const. w. } I)\right] \\
&= \Pr_{b \in_u \{0,1\}^n} \left[(\exists \text{ good path } p \in \mathcal{T}_{\mathcal{B}}) \wedge (p \text{ const. w. } M^{-1}b) \wedge (p \text{ const. w. } I)\right] \\
&\quad + \Pr_{b \in_u \{0,1\}^n} \left[(\exists \text{ bad path } p \in \mathcal{T}_{\mathcal{B}}) \wedge (p \text{ const. w. } M^{-1}b)\right] \\
&\leq \sum_{\substack{g \text{ good} \\ g \in \mathcal{T}_{\mathcal{B}}}} \Pr_{b \in_u \{0,1\}^n} \left[\begin{array}{l} \text{assign. along g} \\ \text{const. w. } M^{-1}b \end{array} \middle| \, g \text{ const. w. } I\right] \cdot \Pr_{b \in_u \{0,1\}^n} [g \text{ const. w. } I] \\
&\quad + o(1) \\
&\leq \sum_{\substack{g \text{ good} \\ g \in \mathcal{T}_{\mathcal{B}}}} 2^{-\Omega(l)} \cdot \Pr_{b \in_u \{0,1\}^n} [g \text{ const. w. } I] + o(1) \\
&= 2^{-\Omega(l)} \sum_{\substack{g \text{ good} \\ g \in \mathcal{T}_{\mathcal{B}}}} \Pr_{b \in_u \{0,1\}^n} [g \text{ const. w. } I] + o(1) \\
&\leq 2^{-\Omega(l)} \cdot \mathrm{E}[\# \text{ of paths at level } l \text{ in } \mathcal{T}_{\mathcal{A}}(I)] + o(1) \leq 2^{-\Omega(l)} \cdot W_{\mathcal{A}}(n) + o(1).
\end{aligned}$$

We have used Lemmas 30 and 28 in lines four and five to establish the inequality. The very last line implies that

$$W_{\mathcal{A}}(n) \geq 2^{\Omega(l)}(1 - o(1)).$$

We have chosen $l = \frac{\delta^2 r}{10K2^K} = \Theta(n)$, (because $r \in \Theta(n)$ and K is a constant), then $l \in \Theta(n)$ and $W_{\mathcal{A}}(n) = 2^{\Omega(n)}$. We conclude that any pBT algorithm solving 7-SAT instances requires exponential width to solve the problem correctly. $\square$

3.4 Lower Bound for pFBT Algorithms for 7-SAT

We use the same hard instances $\mathcal{F}_n$ and uniform distribution on them. The analysis of good intermediary is the same. The only difference is in the analysis of total probability of bad intermediary. When a pFBT algorithm reads a data item from the instance then it can branch on many ordering functions on the remaining data items. While the computation tree of a pBT algorithm has a fan-out of two, the computation tree of a pFBT algorithm has no bound on the fan-out of a branching node. By Lemma 20 if a pFBT algorithm $\mathcal{A}$ is a width w algorithm, then on any instance the width of the tree built by $\mathcal{A}$ is bounded by w, and therefore w is also a bound on the maximum fan-out of a free branching node.

The super decision tree $\mathcal{T}_{\mathcal{B}}$ for a pFBT algorithm will differ from the super decision tree of a pBT algorithm. An algorithmic node makes a decision about a data item which belongs to the instance. It has $0, 1$, or 2 children depending on the choice function. A child of an algorithmic node is a free branching node. The fan-out of a free-branching node is bounded by w by Lemma 20, where the children will be query nodes.

Along a bad intermediary r bits of b are derived and the values of fewer than l variables are set. We derive a bound on the total probability of a bad intermediary using Lemma 29. Fix an assignment for those variables in the super decision tree which belong to the instance. Then by Lemma 29 the probability of a bad path is $\Pr[\sum_{i=1}^{N} D_i < l] < e^{-\frac{\delta^2 N}{2^K}}$. We need to find values for l and w so that the total probability of all bad intermediaries denoted as P_{ALL_BAD} is at most say $\frac{1}{4}$. In the super decision tree the total number of paths with at most l variables being assigned is $2^l \cdot w^l$, hence by union bound and Lemma 29:

$$\begin{aligned} P_{ALL\_BAD} &\leq 2^l \cdot w^l \cdot \Pr\left[\text{a fixed bad intermediary}\right] \leq 2^l \cdot w^l \cdot \Pr_{b \in_u \{0,1\}^n}\left[\sum_{i=1}^{N} D_i < l\right] \\ &\leq w^l \cdot 2^l \cdot e^{-\frac{\delta^2 N}{2^{K+1}}} \leq 2^{l + l \log w} \cdot e^{-\frac{\delta^2 r}{2K 2^K}} \end{aligned}$$

Recall that $r \in \Theta(n)$ and if we choose $l = \sqrt{n}$ and $w = 2^{\alpha\sqrt{n}}$ then,

$$P_{ALL\_BAD} \leq w^l \cdot 2^l \cdot e^{-\frac{\delta^2 r}{2K2^K}} \leq (2^{\alpha\sqrt{n}})^{\sqrt{n}} \cdot 2^{\sqrt{n}} \cdot e^{-\Omega(n)} = 2^{-\Omega(n)},$$

for sufficiently small α.

Theorem 32. *The width of any pFBT algorithm $\mathcal{A}$ is at least $W_{\mathcal{A}}(n) \geq 2^{\Omega\left(\sqrt{n}\right)}$.*

Proof.

$$\begin{aligned} 1 &\leq \Pr_{b \in_u \{0,1\}^n}\left[(\exists \text{ path } p \in \mathcal{T}_{\mathcal{A}}(I)) \wedge (p \text{ const. w. } M^{-1}b) \wedge (p \text{ const. w. } I)\right] \\ &= \Pr_{b \in_u \{0,1\}^n}\left[(\exists \text{ path } p \in \mathcal{T}_{\mathcal{B}}) \wedge (p \text{ consist. w. } M^{-1}b) \wedge (p \text{ const. w. } I)\right] \\ &\leq \sum_{\substack{g \text{ good} \\ g \in \mathcal{T}_{\mathcal{B}}}} \Pr_{b \in_u \{0,1\}^n}\left[\begin{array}{c} \text{assign. along g} \\ \text{const. w. } M^{-1}b \end{array} \middle| \, g \text{ const. w. } I\right] \cdot \Pr_{b \in_u \{0,1\}^n}[g \text{ const. w. } I] \\ &\quad + P_{ALL\_BAD} \\ &\leq \sum_{\substack{g \text{ good} \\ g \in \mathcal{T}_{\mathcal{B}}}} 2^{-\Omega(l)} \cdot \Pr_{b \in_u \{0,1\}^n}[g \text{ const. w. } I] + \frac{1}{4} \\ &= 2^{-\Omega(l)} \sum_{\substack{g \text{ good} \\ g \in \mathcal{T}_{\mathcal{B}}}} \Pr_{b \in_u \{0,1\}^n}[g \text{ const. w. } I] + \frac{1}{4} \\ &\leq 2^{-\Omega(l)} \cdot \mathrm{E}[\# \text{ of paths at level } l \text{ in } \mathcal{T}_{\mathcal{A}}(I)] + o(1) \\ &\leq 2^{-\Omega(l)} \cdot W_{\mathcal{A}}(n) + \frac{1}{4}. \end{aligned}$$

We conclude

$$W_{\mathcal{A}}(n) \geq 2^{-\Omega(l)} = 2^{-\Omega(\sqrt{n})}.$$

□

The lower bounds established in here and in Section 3.3 imply lower bounds on the number of branches visited by both BFS and DFS traversals. It is so because we showed that any branch has either exponentially small probability of being in the tree or subexponential probability of being consistent with the unique solution.

3.5 Boundary Expander Matrix

Definition 17. *A matrix A is (r, k, c) boundary expander if each row has at most k ones and for any set I, such that $|I| \leq r$, then the number of unique neighbors $|\partial_A(I)|$ is at least $c|I|$: $|\partial_A(I)| \geq c|I|$.*

Here we need to prove that a square matrix $A \in \{0, 1\}^{n \times n}$ exists such that

1. Each row sums to exactly 7.
2. Each column sums up to 7Δ - a constant.
3. A is of full rand and hence is non-singular.
4. A is a $(r, 7, c)$ boundary expander for $c > 1$ and $r \in \theta(n)$.

Let Δ be a constant, whose exact value will be determined later. First we define a distribution on $\Delta n \times n$ binary matrices, such that each row has exactly 7 ones and each column sums up 7Δ. Then we show that a random matrix from that distribution is a good boundary expander. Next we argue that with high probability a matrix from that distribution has a full column rank. Then we are done because any n linearly independent rows of that matrix will inherit the expansion of the big matrix and is non-singular. Furthermore, each column will sum up to 7Δ which is a constant and does not depend on n, and each row sums to exactly 7.

The columns of the matrix are the variables. Each variable has 7Δ distinct equation in which it should appear. The rows are the equations with exactly 7 variables each. The total number of 1s in the matrix is $7\Delta n$. How do we match variables to equation and preserve the constraint? Model the matrix as a bipartite graph. Left side are the equations there are Δn of them each has 7 slots (for *distinct* variables). On the other side we have n variables each has 7Δ slots for the 7Δ equations in which a variable participates it. The distribution on matrices is all random perfect matchings of this bipartite graph.

Pick a row at random. From the available unmatched variables pick one at random, remove it (we sample from the columns without replacement). Decrease the number of remaining slots the selected variable has for future selection. Repeat until all 7 variables are chosen. We claim that any set I of at most r rows according to this distribution has boundary greater than $c|I|, c > 1$.

Lemma 33. *Let Δ be a constant whose value will be set later. For any sufficiently large n there exists a matrix $A \in \{0,1\}^{\Delta n \times n}$, which is also a $(r, 7, 3)$ boundary expander.*

Proof. We claim that the probability over the random choices without replacement of the variables in each row, that A is a $(r, 7, c), c = 3$ boundary expander is greater than 0. The set of rows I, $|I| = t \leq r$, define t equations and $7t$ variables of which B are boundary and the remaining D are duplicates. Initially, before the sampling begins each variable has all its 7Δ slots available. Once a variable is chosen it becomes a boundary variable and has only $7\Delta - 1$ slots free. Consider the walk of length $7t$ from Figure 3.2 beginning at state $B = 0/D = 0$ and assume that $r \in \Theta(n)$. At time i, i variables are picked and i of the $7\Delta n$ slots are used. Suppose at time i the number of boundary variables is D and the duplicates is D, respectively. The total slots available to from the boundary variables is $B(7\Delta - 1)$. If a boundary variable is picked again then it becomes a duplicate. The number of boundary variables decreases by 1, the set of duplicates increases by one, and slots of duplicates increases by $7\Delta - 2$. Each duplicate has at most $7\Delta - 2$ slots left and each time a duplicate is chosen again to participate in an equation, the number of available slots decreases. The remaining variables $n - B - D$ have all their slots empty and available to be chosen, that is $7\Delta(n - B - D)$ slots. We need to estimate the probability that after $7t$ steps the chain will end in a state where

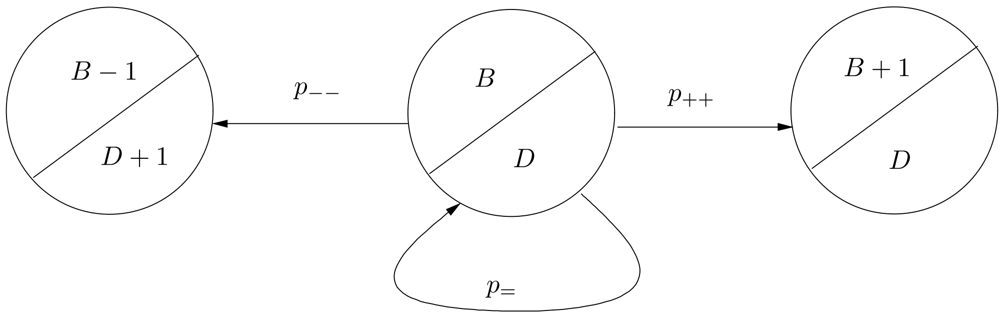

Figure 3.2: Transition function of state B/D.

$B < 3t$. Assume that after $i \leq 7t$ steps the current state is B/D. Only i slots are occupied and $B + 2D \leq i \leq r$, $r \in \Theta(n)$. With probability

$$p_{++} = \frac{7\Delta(n - B - D)}{7\Delta n - i} > 1 - \frac{B + D}{n}$$

the next step will increase the boundary and we go to a state $B + 1$. With probability

$$p_{--} = \frac{(7\Delta - 1)B}{7\Delta n - i} < \frac{B}{n}$$

the next step will decrease the boundary and we go to a state $B - 1$. With probability

$$p_{=} \leq \frac{(7\Delta - 2)D}{7\Delta n - i} < \frac{D}{n}$$

the next step will not change the boundary and the duplicate sets.

From here on when we say *forward* for the direction of the walk, we mean increase the boundary, and as an overestimate when we say *backwards* we mean decrease or do not change the boundary[9]. Although those probabilities depend on the current state we will simplify the analysis by modeling the experiment as a Binomial random variable $B(7t; \hat{p})$. We define the success probability $\hat{p}$ to be the largest probability that a walk of length $7t$ can finish in a state with $B \leq 3t$. Note that any walk of length $7t$ where we go to the right, that is we increase the boundary for $5t$ steps or more will not be able go back to $B < 3t$ state. Then $\hat{p} < 1 - (1 - \frac{B+D}{n}) = \frac{B+D}{n} \leq \frac{5t}{n}$. Now

[9]Note that this assumption leads to not tight analysis because some walks "backwards" will not finish in a state $B < 3t$ but will be counted as such. We trade precise estimate for simplicity.

the event that after $7t$ steps we end up in a state with less then $3t$ boundary elements happens with probability less than the probability that in $7t$ independent trials we have $5t$ forward steps and $2t$ backward steps (or $2t$ successes and rest failures), which is:

$$\binom{7t}{2t}\hat{p}^{2t}(1-\hat{p})^{5t} < \binom{7t}{2t}\hat{p}^{2t}.$$

To find the probability p_t that there exists a set I of size at most t whose boundary is at most $3t$ we use the union bound:

$$\begin{aligned} p_t &< \binom{\Delta n}{t}\binom{7t}{2t}\hat{p}^{2t} \le \left(\frac{e\Delta n}{t}\right)^t \left(\frac{7e}{2}\right)^{2t} \left(\frac{5t}{n}\right)^{2t} \\ &\le \Delta^t \cdot e^{3t} \cdot 20^{2t} \cdot \left(\frac{t}{n}\right)^t \le \left[e^3 \cdot 20^2 \left(\frac{\Delta t}{n}\right)\right]^t \\ &< \left[(20e)^3 \left(\frac{\Delta r}{n}\right)\right]^t \end{aligned}$$

Then the probability that there exists a set of size $t \le r$ whose boundary is at most $3t$ is

$$p = \sum_{t=1}^{r} p_t.$$

If we choose $r = \frac{n}{3\Delta\cdot(20e)^2} \in \Theta(n)$, then $p_1 < \frac{1}{2}$ and $p < 1$, hence such an expander exists. $\square$

Next we need to show that with high probability a random matrix chosen from the distribution has a full column rank.

Lemma 34. *Let A be a random $\{0,1\}^{\Delta n \times n}$ matrix with exactly 7 ones in each row and exactly 7Δ ones in each column. Then with high probability $rank(A) = n$.*

Proof. Suppose not. Suppose the rows of the matrix are contained in a proper subspace (a subspace of dimension $k < n$). Let

$$x_{i_1} + \cdots + x_{i_k} = B \tag{3.3}$$

be the subspace equation, where $B = 1$ or $B = 1$. To be a valid subspace the 0 vector must satisfy Eq. (3.3), hence it cannot be the case that $B = 1$, and any valid subspace equation for the matrix A (whose rows define $\mod 2$ equations) is of the form:

$$\sum_{i \in S \subset [n]} x_i = 0. \tag{3.4}$$

Furthermore, each row equation of A must satisfy Eq. (3.4) therefore each row of A must contain exactly two, four, or six variables which also participate in the subspace equation. Any equation from the matrix which has odd number of variables from the subspace equation will contradict Eq. (3.4). Hence we need to bound the probability that a fixed subspace equation is satisfied by each matrix equation of A, given the constraints. We consider two cases when the number of variables in the subspace equation is small and when it is large.

1. Say the number of variables in the subspace equation is $t \leq \frac{n}{12}$ (it must be the case that $t \in \Theta(n)$ otherwise the probability all Δn matrix equations satisfy the subspace equation, given the distribution of the matrix A is zero). Intuitively, no such equation exists because when t is small the probability that the subspace equation will be satisfied by all rows is small.

   Fix the variables in the subspace equation. Let those be $F = \{x_{i_1}, \ldots, x_{i_t}\}$. Then the subspace equation is:

   $$x_{i_1} + x_{i_2} + \cdots + x_{i_t} = 0. \tag{3.5}$$

   So need to express the probability that a fixed equation is a valid subspace equation for A. Now we evaluate the probability that Eq. (3.5) is satisfied by all rows from the matrix A. Meaning each row of A must have at least two but even (four or six) number of variables from F, given the distribution that no variable in F can participate in more than 7Δ equations and that each equation has exactly 7 variables.

   Now consider the matrix equations one by one. Each matrix equation must have two, four, or six variables from F. But also each variable in F can participate in exactly 7Δ equations (no more!).

Definition 18. *Call a matrix equation surviving if it has an even number of variables sampled from F without replacement.*

Each variable in each equation is chosen at random without replacement from the remaining available variables. Initially the t variables in F have all their $7\Delta t$ slots available and the probability that the first matrix equation survives is at most the probability that at least one of the remaining six variables is sampled from F. Then by union bound the probability the first equation survives is

$$6 \times \frac{7\Delta t}{7\Delta n} = \frac{6t}{n}.$$

Say at time i when the first i equations have survived the variables from F have at most $7\Delta t - 2i$ slots available and in total $7\Delta n - 7i$ slots are used. The the probability that the $i+1$-st equation survives by union bound is bounded by

$$\frac{6(7\Delta t - 2i)}{7\Delta n - 7i} \leq \frac{6t}{n}.$$

Now the probability that Eq. (3.5), is satisfied by all Δn equations of the matrix A is at most the probability that the first Δt equations have survived and the probability of that is at most $(\frac{6t}{n})^{\Delta t}$.

Hence by a union bound the probability that there exists an equation of t variables which is satisfied by all equations of A is bounded above by:

$$\binom{n}{t} \cdot \left(\frac{6t}{n}\right)^{\Delta t} < \left(\frac{en}{t}\right)^{t} \cdot \left(\frac{6t}{n}\right)^{\Delta t} = (6e)^t \cdot \left(\frac{6t}{n}\right)^{(\Delta-1)t}$$

For $t \leq \frac{n}{12}$ and $\Delta > 5$ the probability above is $2^{-\Omega(n)}$.

2. Consider the case when the subspace equation has greater than $\frac{n}{12}$ variables. We argue that a reasonable fraction of all matrix equations are likely to contradict the subspace equation because we show that it is likely that all variables from the matrix equation are also in the subspace equation. Hence the probability that many equations equations survive is exponentially small.

Initially the variables in the subspace equation have all their slots available, in total $7\Delta t$ slots. Refer to the slots for the variables in the subspace equation as T, $|T| = t > \frac{n}{12}$.

Definition 19. *We say that an equation is in violation if all its variables are from T, and thus will contradict the subspace equation. If an equation contradicts the subspace equation then we say that it cannot survive.*

We seek to bound from above the probability that at least the first $\frac{\Delta n}{12}$ equations are not in violation by bounding from below the probability they contradict the subspace equation.

Consider the first matrix equation. The probability that the first variable is from T is $\frac{7\Delta t}{7\Delta n} \geq \frac{1}{13}$ because $t > \frac{n}{12}$. The probability that the second variable is also from T is at least $\frac{7\Delta t - 1}{7\Delta n - 1} > \frac{1}{13}$. Then the probability that the first equation is in violation and all seven variables are from T is at least $\left(\frac{1}{13}\right)^7$. And the probability that the first equation survives is at most $1 - (\frac{1}{13})^7$.

Given that the i-th matrix equation has survived, then the probability that the $i+1$-st matrix equation is in violation is at least

$$\prod_{j=1}^{7} \left(\frac{7\Delta t - 7i - j}{7\Delta n - i - j} \right) \geq \left(\frac{1}{13} \right)^7 ,$$

And for $i \leq \frac{\Delta n}{13^2}$ the probability that the i-th equation survives is at most

$$\left(1 - \left(\frac{1}{13} \right)^7 \right) .$$

Fix a subspace equation with more than $n/12$ variables. The probability that all matrix equation survive the subspace equation is at most the probability that the first $i = \frac{\Delta n}{13^2}$ equations survive, which is at most

$$\left(1 - \left(\frac{1}{13} \right)^7 \right)^{\frac{\Delta n}{13^2}} \leq e^{-\frac{\Delta n}{13^9}}$$

Then by union bound the probability that there exists a matrix equation with $t > \frac{n}{12}$ variables is:

$$\binom{n}{t} \cdot e^{-\frac{\Delta n}{13^7}} < 2^n \cdot e^{-\frac{\Delta n}{13^9}}$$

As long as $\Delta > 13^9$ the above probability is exponentially small.

□

By Lemmas 33 and 34 for reasonably large n there exists a non-singular binary matrix which is $(r, 7, 3)$ boundary expander in which each column sums up to a constant $K = 7\Delta$. Choose $\Delta = 2 \cdot 13^9$, then $K = 14 \cdot 13^9$.

3.6 Notes

The pFBT model presented here and the 7-SAT lower bounds for pBT and pFBT algorithms are part of manuscript “A General Model for Backtracking and Dynamic Programming Algorithms”, by Josh Buresh-Oppenheim, Sashka Davis and Russell Impagliazzo, [16]. Jeff Edmonds recently obtained a pFBT algorithm solving the 7-SAT instances in the family $\mathcal{F}_n$ of width bounded by $O(2^{\sqrt{n}\log n})$. The upper bound and Theorem 31 imply an exponential separation between the power of pBT and pFBT algorithms.

Chapter 4

Can pBT Algorithms Solve the Shortest Path Problem?

In this chapter we present another technique for proving lower bounds on the width of pBT algorithms. We use this technique to obtain an exponential lower bound on the width of fully adaptive pBT algorithms for shortest path problem where the weights on the edges can be negative. The shortest path problem we consider is defined as follows:

Definition 20. *(SSSP) Given a directed graph $G = (V, E)$ and a node $s \in V(G)$. Find the simple shortest path from s to a node in G.*

The lower bound technique we use here is different and not as general as the technique used in Chapter 3. The technique we define in this chapter can only be used for problems where the set of decisions is $\Sigma = \{accept, reject\}$. Let $\mathcal{A}$ be a pBT algorithm for some problem and consider the tree $T = \mathcal{T}_{\mathcal{A}}(I)$ for some fixed instance of this problem. Each state s in T is defined by the partial solution PS_s and the partial information $PI_s = \{PI_s^{in}, PI_s^{out}\}$. PI_s^{in} are data items which belong to the instance and are assigned decisions along the path from the root to s. A data item e belongs to PI_s^{out}, if there is a state s' along the path from the root to s, and the ordering function at the state s' is $\pi(s') = \{\ldots, e, \ldots, e' \ldots\}$ such that e' is the first data item in the total order

$\pi(s')$ which belongs to the instance I. If $e \in PI^{out}$ then it is the case that $e \notin PI^{in}$. PS_s is that subset of PI_s^{in} that has been accepted.

Definition 21. *(consistency) We call an instance I consistent with a partial information $PI_s = \{PI_s^{in}, PI_s^{out}\}$ if $(PI_s^{in} \subseteq I)$ and $(I \cap PI_s^{out} = \emptyset)$.*

Definition 22. *(unique extension) Let PS_s be the set of data items accepted along a branch. Then PS_s is* uniquely extendible *with respect to PI_s if there is an instance I_s consistent with PI_s such that, the only solutions to the problem on I_s are equal to PS_s when restricted to PI_s.*

Definition 23. *(agreeable data item) A new item e* agrees *with the partial information PI_s and the partial solution PS_s if there is a valid instance I_s consistent with PI_s which has some optimal solution consistent with PS_s that contains e.*

Definition 24. *(competing data item) If a data item is not agreeable then it is competing.*

The lower bound is defined as a game played by a Solver and an Adversary, which we call the Q-improbability game. The game proceeds in rounds.

1. The Adversary privately selects an instance I, which might not be a valid instance. Then sets $Q_0 \leftarrow 1$; $t \leftarrow 1$.

   The Solver initializes empty revealed information and partial solution: $PI_0^{in}, PI_0^{out}, PS \leftarrow \emptyset$.

2. round t begins

   (a) The Solver picks a data item $d_t \notin PI_{t-1}$.

   (b) The Adversary reveals whether $d_t \in I$ or not.

   - If $d_t \notin I$ then the Solver updates $PI_t^{out} \leftarrow PI_{t-1}^{out} \cup \{d_t\}$ and next round begins.

- If $d_t \in I$, then the Solver updates $PI_t^{in} \leftarrow PI_{t-1}^{in} \cup \{d_t\}$. The Adversary picks probability q_t.
  With probability q_t, $PS \leftarrow PS \cup \{d_t\}$; $Q_t \leftarrow Q_{t-1} \times q_t$;
  With probability $1-q_t$, PS remains unchanged; $Q_t \leftarrow Q_{t-1} \times (1-q_t)$;

   round t ends; $t \leftarrow t+1$.

3. Game ends when at the beginning of some round t, $Q_t \leq Q$. Let PI and PS denote the partial information and solution at the end of the game.

4. The Adversary wins if there exists a *valid* instance I' consistent with PI so that every optimal solution in I' agrees with PS, otherwise the Solver wins.

Lemma 35. *Let Π_n be a problem with a finite set of data items D_n. If there is a width w pBT algorithm for Π_n, then Solver wins with probability $1 - Qw$.*

Proof. Let $\mathcal{A}$ be a width w pBT algorithm for Π_n, as specified in Chapter 3. Solver simulates $\mathcal{A}$ to pick the next data item as follows: Initially, Solver runs $\mathcal{A}$ and obtains the ordering $\pi_{S_0}(\mathcal{D})$. Let d be the first data item in the ordering, then Solver sets $d_1 = d$. At round i Solver maintains PI_i^{in}, PI_i^{out}, and PS. PS and PI_i^{in} uniquely define a branch of the BT tree and a state in it, let that state be s. Then the Solver uses π_s and picks d_i to be the first item in the ordering. The game ends when the probability of the current path (determined uniquely by $PI = PI^{in}, PI^{out}, PS$) falls below Q.

The Adversary loses when the branch corresponding to PS and PI at the end of the game is not part of any BT tree $\mathcal{T}_{\mathcal{A}}(I)$ built by $\mathcal{A}$ on any valid instance I. But $\mathcal{A}$ is width w algorithm, so the probability the branch defined by PS and PI is in the tree is at most Qw, hence Solver wins with probability $1 - Qw$. □

Corollary 36. *If Adversary can win with probability greater than $1/2$ then all pBT algorithms have width $w \geq \frac{1}{2Q}$.*

Definition 25. *(competing edge) An edge (u, v) is competing with respect to a given partial solution PS if there exists edge $(u, x) \in PS$ or there exists edge $(w, v) \in PS$. Otherwise it is agreeable.*

A competing edge cannot be part of the solution because, adding such an edge to PS will destroy its validity (PS will no longer be a simple path).

Now we describe the strategy for the Adversary for the shortest path in graphs with negative weights but no negative cycles. Let t and p parameters which depend on n such that $t = n^{1/9}$ and $p = n^{-3/4}$. The Adversary picks an instance I at random from $G(n, p)$. Each edge if present has a weight of -1. Let $PI_i = PI_i^{in}, PI_i^{out}$ and PS_i be the revealed information and the partial solution after round i has finished. At round $i+1$ the Solver selects an edge e: If e is competing with PI_i^{in} then $q_{i+1} = 0$. Otherwise $q_i = 1/2$. Then at the end of the round we have $Q_{i+1} = \frac{1}{2^{t_i}}$, where t_i is the number of agreeable edges seen so far. Without loss of generality, we assume that the Solver views all competing edges after each edge e is added to PS. The Q-improbability game is played for $Q = \frac{1}{2^t}$. In what follows we first characterize the random instance I chosen by the Adversary before the first round of the Q-improbability game. Then we show that there exists another valid instance with the same revealed information as the revealed information at the end of the game, such that the partial solution at the end of the game is consistent with the unique solution in the valid instance, hence the Adversary wins the game with probability greater than $1/2$, establishing exponential lower bound on the width of any pBT algorithm for the shortest path problem.

Lemma 37. *Let $PI = (PI^{in}, PI^{out})$ be the partial information and PS be the partial solution. If they satisfy the following three invariants, then PS is uniquely extendible with respect to PI:*
[P.1] PI^{in} contains no cycles.
[P.2] $|PI^{in}|$ is $o(\sqrt{n})$.
[P.3] Any node in PS with in(out)-degree 0 in PS has in(out)-degree $o(n)$ in $PI^{in} \cup PI^{out}$. Any node in $V(PI^{out}) \setminus V(PI^{in})$ has in(out)-degree $o(n)$ in $PI^{in} \cup PI^{out}$.

Proof. We will exhibit a set of edges M such that, M is disjoint from PI and $PS \cup M$ is the unique longest path starting at s in $PI^{in} \cup M$.

Since PI^{in} contains no cycles then PI^{in} is a directed acyclic graph. View PS

as a set of disjoint simple directed paths, let those be $P_1, \ldots, P_k$ such that, P_i goes from, say, u_i to v_i. If PS doesn't touch s, then we assume P_1 begins at u_1 and ends at v_1, where $u_1 = v_1 = s$ otherwise, $u_1 = s$ and $v_1 \neq s$ be the end of the path in PS leaving s. Since PI^{in} is a directed acyclic graph it defines a partial order on the paths P_i. Let $\prec$ be a total order on the paths $P_1, \ldots, P_k$ consistent with the partial order defined by PI^{in} such that, $P_i \prec P_j \Rightarrow i < j$. Let V be the nodes not touched by PI^{in}, $V = V(G) \setminus V(PI^{in}) = [n] \setminus V(PI^{in})$, and let E be the set of all possible edges minus those in $PI^{in} \cup PI^{out}$. We construct k disjoint simple directed paths $Q_1, \ldots, Q_k$ such that, Q_i goes from v_i to u_{i+1} for $i < k$ and Q_k goes from v_k to some node in V. Note that the edges in $\{Q_1, \ldots, Q_k\}$ respect the order $\prec$. The set of simple paths $\{Q_1, \ldots Q_k\}$ satisfies the following conditions:

- $|Q_i| > |PI^{in}|, \forall i = 1, \ldots, k$,
- the intermediate nodes of each Q_i are from V,
- the edges of each Q_i are all from E.

M will be the set of all edges in $\{Q_i\}$. Note that $PI^{in} \cup M$ is cycle free because the paths Q_i respect the order $\prec$.

Claim 38. *$P = PS \cup M$ is the unique longest path in $PI^{in} \cup M$.*

Proof. Let P' be any other path $P' \neq P$. They both must originate at s and since they are distinct they must diverge at some node $d \in V(P_1 \cup \cdots \cup P_k)$. It could not be the case that $d \in \{Q_1 \cup \cdots \cup Q_k\}$ because each Q_i is a simple path. Suppose $d \in P_i$. Let (d, u) be the edge in P' then $(d, u) \in PI^{in}$. If the path P' never rejoins with P, then it must skip at least one Q_i, but $|Q_i| > |PI^{in}|$, hence P' is shorter than P. The path P' cannot rejoin P within the segment P_i because PI^{in} has no cycles. Also note that P' cannot rejoin P at some segment P_m, for $m < i$ because to do so we need an edge which disagrees with the order $\prec$, but all edges in PI^{in} agree with the order $\prec$ and the edges in $\{Q_1, \ldots, Q_k\}$ also respect the order. Hence no such edge exists. On the other hand if the section of P' after node d rejoins with P then it must be at a segment P_j with $j > i$, thus missing all edges in Q_i, and because $|Q_i| > |PI^{in}|$ then P' is shorter. $\square$

To construct the paths $\{Q_i\}$ from E we need the following claim:

Claim 39. *Let $H = (S, E(S))$ be a graph of order m, $|S| = m$ such that, each node $x \in S$ has at least $m - o(m)$ in neighbors and at least $m - o(m)$ out neighbors in S. Let $u, v \in S$ be two nodes such that*[1] $|\Gamma(u)_{out}| > \frac{m}{2}$ *and and* $|\Gamma(v)_{in}| > \frac{m}{2}$, *then there exists at least one node w such that $(u, w) \in E$ and $(w, v) \in E$.*

Proof. By the pigeon hole principle. $\square$

Initially $H = (V, E)$. Also note that $u_2, \dots u_k$ have $n - o(n)$ in-neighbors in V and every $v_1, \dots, v_k$ has $n - o(n)$ out-neighbors in V. Let $|PI^{in}| = L \in o(\sqrt{n})$. We need to construct k paths of length $L+1$, where $k \leq t+1$, and we know that $L \in o(\sqrt{n})$. Note that invariant [P.3] implies that any node with zero in(out)-degree in PS has $o(n)$ out(in)-degree in PI. Nodes with zero out-degrees in PS are nodes $v_1, \dots v_k$ and they will have at least $n - o(n)$ degree in E. Similarly nodes $u_2, \dots, u_k$ have zero in-degree in PS and by invariant [P.3] are guaranteed to have $n - o(n)$ degree in E. Also note, that all other nodes $V(PI) \setminus V(PS)$ have zero in or out degree in PS and again by invariant [P.3] are guaranteed to have $n - o(n)$ degree in E. The remaining nodes $[n] \setminus V(PI)$ have in and out degrees n in E.

We construct paths $Q_1, \dots, Q_k$, such that $Q_i = v_i, w_1^i, \dots, w_L^i, w^i, u_{i+1}$. We begin with Q_1. Choose $w_1^1, w_1^2 \in V$ such that $(v_1, w_1^1), (w_1^1, w_2^1) \in E$, remove w_1 from V. Continue until we have a path $v_1, w_1^1, w_2^1, \dots, w_L^1$ of length L. Because both w_L^1 and u_2 have had $n - o(n)$ in-out neighbors in E at the beginning and $L \in o(\sqrt{n})$, then by Claim 39 there exists a node w^1 such that $(w_L^1, w^1) \in E$ and $(w^1, u_2) \in E$. Let $Q_1 = v_1, w_1^1, \dots w_L^1, w^1, u_2$, then the length of Q_1 is $|Q_1| = L + 2 > |PI^{in}| = L$. Note that we have removed $L + 1 \in o(\sqrt{n})$ nodes from V and hence the in and out degrees of nodes V in E remain $n - o(n)$. The remaining paths $Q_2, \dots, Q_k$ are built in the same

[1] *We use the standard mathematical notation: $\Gamma(x)_{in}$ is the set of neighbors of x such that $(u, x) \in E$ and $\Gamma(x)_{out}$ is the set of neighbors of x such that $(x, u) \in E$.*

way as the path Q_1. The procedure is possible because each path removes $L + 1 \in o(n)$ nodes from V, repeated at most $t + 1$ times, in total at most $n^{1/9+1/2} = o(n)$ nodes are removed. Therefore at any stage during the construction of the paths Q_i the remaining nodes will still have in and out degrees of at least $n - o(n)$ in E.

$\square$

Our goal is to show that at the end of the Q-improbability game PI and PS satisfy invariants [P.1], [P.2], [P.3] with high probability. Edges which were rejected but were agreeable with the current partial solution PS are denoted by R. Whenever $\mathcal{A}$ accepts an edge (u, v) and adds it to PS, then all edges of G out of u and into v are revealed. Those edges are called competing with PS and are denoted by C. At the end of the Q-improbability game the edges present in the graph are $PI^{in} = R \cup C \cup PS$ and the revealed information $PI = PI^{in}, PI^{out}$. Define $t = |R| + |PS|$ (t is the number of agreeable items seen at the end of the Q-improbability game). We show tight bounds on the sizes of the sets of edges PS, R, C, and PI^{out}. Recall that $t = n^{1/9}$ and $p = n^{-3/4}$. The Adversary stops the game when t reaches $n^{-1/9}$.

Consider the partial solution PS. Let X_i for $i = 1, \ldots, t$ be a family of indicator random variables such that $X_i = 0$ if the i-th edge was in PS and 0 otherwise. By the path picking strategy $\Pr[X_i = 0] = \Pr[X_i = 1] = 1/2$. Then the expected number of accepted edges is:

$$\mathrm{E}\left[|PS|\right] = \mathrm{E}\left[\sum_{i=1}^{t} X_i\right] = \sum_{i=1}^{t} \Pr[X_i = 1] = \frac{t}{2}.$$

Because $X_1, \ldots, X_t$ are independent, then for any $0 < \delta_1 \leq 1$ and $\delta_2 \leq 2e - 1$ we can apply Chernoff's bound, $\Pr\left[|PS| < (1 - \delta_1)\frac{t}{2}\right] < e^{-\frac{t\delta_1^2}{8}} = e^{-\Omega(n^{1/9})}$ and

$$\Pr\left[|PS| > (1 + \delta_2)\frac{t}{2}\right] < e^{-\frac{t\delta_2^2}{8}} = e^{-\Omega(n^{1/9})}. \tag{4.1}$$

Same argument establishes that the expectation of the number of rejected edges and the probability that this number deviates from the expectation.

$$\mathrm{E}[|R|] = \frac{t}{2}$$

$$\Pr[|R| > (1+\delta_4)\frac{t}{2}] < e^{-\frac{t\delta_4^2}{4}} = e^{-\Omega(n^{1/9})}. \tag{4.2}$$

Now we consider the set edges C competing with PS (revealed after an agreeable edges was accepted by $\mathcal{A}$). Let D_i for $i = 1, \ldots, t$ be the number of edges incident to the nodes of the i-th edge, then for all $i = 1, \ldots, t$ we have $\mathrm{E}[D_i] = np$ and $\Pr[D_i > (1+\delta)2np)] < e^{-\delta^2 np} = e^{\Omega(n^{1/4})}$, for any $\delta \in [1, 2e-1]$.

$$\Pr\left[\sum_{i=1}^{t} D_i > (1+\gamma)2tnp\right] < e^{-\frac{2\gamma^2 tnp}{2}}.$$

The number of competing edges is $|C| = \sum_{i\in A} D_i$ and $\mathrm{E}[|C|] = \sum_{i=1}^{t}\mathrm{E}[D_i]$:

$$\Pr\left[\frac{\sum_{i=1}^{t} D_i}{2} > \frac{(1+\gamma)2tnp}{2}\right] = \Pr\left[\sum_{i=1}^{t} D_i > (1+\gamma)2tnp\right],$$

hence

$$\Pr[|C| > (1+\gamma)2tpn] = \Pr\left[\frac{\sum_{i=1}^{t} D_i}{2} > (1+\gamma)tnp\right] < e^{-\frac{2\gamma^2 tnp}{2}} = e^{-\Omega(n^{13/36})}. \tag{4.3}$$

Next we look at the size of PI^{out}. Let X_i, for $i = 1, \ldots, t$, be a family of random variables, such that X_i is the number of non-existing edges learned by the algorithm $\mathcal{A}$ prior to querying the i-th agreeable edge (we have in total t agreeable edges along P) of the instance along the path P. Note that X_i's are independent and identically distributed geometric random variables with parameter p, and the number of nonexistent edges is $|PI^{out}| = \sum_{i=1}^{t} X_i$, with expectation $\mathrm{E}[|PI^{out}|] = \frac{t}{p}$. Let X be a geometric random variable with parameter p. Then

$$\Pr\left[X < \frac{1+\delta}{p}\right] = 1 - \sum_{i\geq\frac{1+\delta}{p}} (1-p)^i = 1 - \frac{(1-p)^{\frac{1+\delta}{p}}}{p} > 1 - \frac{e^{-(1+\delta)}}{p}.$$

Choose $\delta = 2\ln\frac{1}{p}$, then $\Pr\left[X < \frac{1+2\ln\frac{1}{p}}{p}\right] > 1 - \frac{p}{e}$. We will use the Hoeffding's inequality ([28]), which states that if $Z_1, ..., Z_t$ are independent random variables assuming values in the range $[a_i, b_i]$, respectively, and $S = \sum_i Z_i$ then

$$\Pr[S - \mathrm{E}[S] \geq x] \leq \exp\left(-\frac{2x^2}{\sum_{i=1}^{t}(b_i - a_i)^2}\right).$$

We define a new random variable $\tilde{X} = \begin{cases} X, & \text{if } X < \frac{1+\delta}{p}; \\ 0, & \text{otherwise.} \end{cases}$

Then with probability $1 - \frac{e}{p}$ we have that $\sum_{i=1}^{t} \tilde{X}_i = \sum_{i=1}^{t} X_i$ and $\mathrm{E}[\tilde{X}_i] < \mathrm{E}[X_i] = \frac{1}{p}$, for all i. Now we apply the Hoeffding's inequality for $\sum \tilde{X}_i$.

$$\Pr\left[\sum \tilde{X}_i > (1+\gamma)\frac{t}{p}\right] < \exp\left(-\frac{2(\gamma t/p)^2}{t(\frac{1+\delta}{p})^2}\right) = e^{-\frac{2\gamma^2 t}{(1+\delta)^2}} = e^{-\Omega\left(\frac{n^{1/9}}{\ln^2 n}\right)}. \tag{4.4}$$

We prove the validity of the invariants out of order.

Lemma 40. *With probability* $1 - o(1)$ *invariant [P.2] holds and* $|PI^{in}| \in o(n)$.

Proof. $PI^{in} = R \cup PS \cup C$ and the expected size of PI^{in} is

$$\mathrm{E}[|R|] + \mathrm{E}[|PS|] + \mathrm{E}[|C|] = t + 2tpn = n^{1/9} + n^{13/36} = o(\sqrt{n}).$$

Then by Equations (4.2), (4.1), and (4.3) the probability $|PI^{in}|$ exceeds $o(\sqrt{n})$ is strictly smaller than $2e^{-\Omega(n^{1/9})} + e^{-\Omega(n^{13/36})}$. □

Lemma 41. *[P.1] With probability* $1 - o(1)$ PI^{in} *is cycle free (in undirected sense).*

Proof. Invariant [P.1] holds true with probability $1 - e^{-\Omega(n^{13/36})}$ and the size of PI^{in} is $t + 2tpn < 3tpn$. We ignore direction of edges here and will reason about PI^{in} as an undirected graph. If PI^{in} is a single connected component then a cycle is created if an edge between any two nodes is added to PI^{in}. If PI^{in} is a forest of connected components then adding an edge between two nodes touched by PI^{in} might or might not create a cycle. Therefore a cycle exists in PI^{in} with probability at most $\binom{3tpn}{2}p < 9t^2p^2n^2p = 9(tn)^2p^3 = 9n^{-1/36} \in o(n)$.

□

Lemma 42. *With probability* $1 - o(1)$ *invariant [P.3] holds true.*

Proof. Fix a node, let it be x, such that $x \in V(PS)$. Suppose x's in-degree in PS is 0. If x's in degree in PI^{in} is 0 then x must have a very high degree in PI^{out}. By Equation (4.4) with probability $e^{-\Omega\left(\frac{n^{1/9}}{\ln^2 n}\right)}$ the number of non-edges revealed during the whole game exceed $\frac{2t}{p} = n^{31/36} = o(n)$. The total size of PI^{out} is a bound on the number of edges in PI^{out} incident to x. The expected number of non-edges is $\mathrm{E}[|PI^{out}|] = \frac{t}{p}$, and each non-edge appears independently with probability $1-p$, hence the probability a fixed node x has more than than $2t/p = 2n^{31/36} = o(n)$ is at most $(1-p)^{2t/p} = e^{-2t}$. $e^{-\Omega\left(\frac{n^{1/9}}{\ln^2 n}\right)}$.

If x's degree in PI^{in} is non-zero then all edges coming into x must have been queried and rejected since none of the edge in PI^{in} coming into x would be competing and revealed (for free during the game). The number of rejected edges coming into x is at most $t = n^{1/9} = o(n)$ and this happens with probability 2^{-t}, therefore such a node will have a high in-degree in $PI^{in} \cup PI^{out}$ if it has a high degree in PI^{out}. Similarly as before the probability that x has a degree at least $2t/p$ in PI^{out} is at most e^{-2t}. Then by a union bound the probability there exists a node which violates invariant [P.3] is at most $ne^{-\Omega(t)}$. □

Lemma 43. *Let $p = n^{-3/4}$ and $t = n^{1/9}$. Consider the Q-improbability game, for $Q = \frac{1}{2^t}$, played by the Solver and the Adversary for the shortest path problem on a random graph $G \sim \mathcal{G}_{dir}(n,p)$. The probability, over random G and the random coin tosses of the Adversary, that PS is not uniquely extendible with respect to PI is $o(1)$.*

Proof. By Lemma 37 the partial information and solution at the end of the Q-improbability game is uniquely extendible if all three invariants [P.1],[P.2],[P.3] are satisfied. By Lemmas 41, 40, and 42 the probability that the partial information PI and the partial solution PS will violate any of the invariants is: $n^{-5/18} + 2e^{-\Omega(n^{1/9})} + 2^{-\Omega(n^{13/36})} + ne^{-\Omega\left(\frac{n^{1/9}}{\ln^2 n}\right)} = o(1)$. Hence with probability $1 - o(1)$ at the end of the Q-improbability game the partial instance is uniquely extendible. □

Theorem 44. *Any fully adaptive pBT algorithm for shortest path with negative weights and no negative cycles on graphs of size n requires width $\Omega(2^{n^{1/9}})$.*

Proof. Lemma 43 guarantees a single graph G for which the result of the

$2^{n^{1/9}}$-improbability game is uniquely extendible with probability $1 - o(1)$. The theorem now follows directly from Lemma 35. $\square$

4.1 Notes

The lower bound proof presented in this chapter is part of manuscript “A Stronger Model for Dynamic Programming Algorithms” by Josh Buresh-Oppenheim, Sashka Davis, and Russell Impagliazzo, [17], which is being prepared for publication.

Chapter 5

prioritized Branching Programs

Although the pBT model of [1] captures large class of dynamic programming algorithms, in Chapter 4 we proved that no pBT algorithm can solve the single source shortest path problem in graphs with negative weights. In this chapter we will build a formal model which inherits the power of pBT algorithms but also is capable of solving efficiently the shortest path problems in general graphs.

In the framework of Priority algorithms an instance is represented as a set of items. The algorithm orders the data items from the instance according to a priority function and once it observes a data item it has to commit to a decision, which is irrevocable. The algorithm begins with no knowledge of the contents of these items, but it can order them. In each step, the first item in the ordering is revealed and the algorithm makes a decision about it. This decision-making process restricts the algorithm to maintain a *single partial solution* during its computation. The pBT model generalizes the priority framework by allowing the algorithm to maintain a *tree of partial solutions*, while the pFBT model generalized the pBT model by allowing in addition many priority functions to be explored in parallel. Our new model called *prioritized Branching Programs* (pBP model), like the pBT model, maintains multiple partial solutions, but also allows memoization which seems essential to the concept of dynamic programming. Consider the following intuitive definition ([19]): ”Dynamic-programming algorithms typically take advantage of overlapping subproblems by solving each subproblem once and then storing the solution in a table where it can be looked up when needed....There is a variation of dynamic programming that offers the efficiency of the usual dynamic-programming approach while maintaining a top-down strategy. The idea is to *memoize*". The Prioritized Branching Programs (pBP) algorithms combine the power of **branching** with the power of **memoization**. Branching allows multiple partial solutions to be maintained, while **merging** allows different branches of the computation to memoize the solution to common subproblems for later reuse (so in a sense $\mathbf{pBP} = \mathbf{BRANCH} + \mathbf{MERGE}$). Any discrete optimization problem can be solved by a pBP algorithm, although possibly not an efficient one. The computation of a pBP algorithm consists of three phases. On a given input instance the algorithm generates a directed acyclic graph top-down successively as it sees more and more of the input. It then traverses the DAG bottom-up to obtain the value of the best solution it computed, and then finds the actual solution with one more top-down traversal. The number of states in the computational DAG (size) is closely related to the quality of the solution an algorithm finds. In particular, if we allow an exponential size we can solve any optimization problem.

5.1 The pBP Model

We begin with formal definitions and then illustrate them with examples. We will define infinite problems Π as a collection of finite problems $\{\Pi_n | n \in \mathbb{N}\}$, where n is a size parameter. When n is understood, we will drop the subscripts. Algorithms will also be non-uniform, in that the size parameter is known to the algorithm, and we will not enforce any connection between the algorithms for different size parameters n.

For each n the problem Π_n has the following priority model:

1. A universe of possible data items, $\mathcal{D}$. There is a special element $end \notin D$ which is a marker to halt the computation before all data items are seen. So in a sense end signals the absence of unseen items.

2. A collection of ***valid instances***, where each valid instance I is viewed as a set of data items, $I \subseteq D$. Each valid instance contains end.

3. A set $\Sigma = \Sigma_n$ of ***decision options*** for each data item.

4. An objective function $F = F_n$ which takes an input of the form $(d_1, \sigma_1), \ldots, (d_k, \sigma_k)$ with $d_i \in \mathcal{D} - \{end\}$ and $\sigma_i \in \Sigma$ and returns a real number, infinity, or negative infinity. (Without loss of generality, we can assume that the objective is to maximize F; we can model minimization by maximizing $-F$. We also usually assume $\{d_1, \ldots, d_k\}$ is a valid instance; however, we can give F an arbitrary value such as 0 if not. We can model search and decision problems by picking F to be a Boolean function which has value 1 on feasible outputs.)

Given an instance of the search/optimization problem $I \subset \mathcal{D}^k$ the problem is to find a solution, which is an assignment of an option $\sigma_i \in \Sigma$ to each data item $d_i \in I - \{end\}$. So a *solution* for $I = \{d_1, \ldots, d_k, end\}$ is a set of the form $\{(d_i, \sigma_i) | i = 1, \ldots, k\}$, where each $\sigma_i \in \Sigma\}$, and we wish to, given I, find a solution such that $F((d_1, \sigma_1), (d_2, \sigma_2), \ldots, (d_k, \sigma_k))$ is maximized. The additional item end has no choices associated with it, and represents terminating the algorithm when all edges have been labeled.

5.1.1 Defi nition of a General pBP Algorithm

In the following, since we allow algorithms to be non-uniformly tuned to the size parameter, we omit this parameter as a subscript.

For any Π, a pBP algorithm specifies $\mathcal{S}$, a set of *computation states* (possibly infinite) and $\mathcal{T} \subset \mathcal{S}$, a set of *terminal states*. There is a special empty state $S_0 \in \mathcal{S}$, which is the initial (start) state of any pBP algorithm. (Note that $\mathcal{S}$ and $\mathcal{T}$ are problem-specific, but not instance-specific. Intuitively, one can think of each state $s \in \mathcal{S}$ as representing a "sub-problem" to be solved recursively, with the terminal states as the "base cases" of the recursion. The sub-problem might not be determined by s alone, but also by the unseen or unremembered parts of the input; e.g., S_0 always represents "The complete instance, whatever it is". Alternatively, like in a memoized dynamic programming algorithm, one can think of states as encoding the class of partial solutions to the problem that cause the state to be reached.) Let $\mathcal{M}(R)$ be the set of all monotone functions from $\mathbb{R}\cup\{\infty, -\infty\}$ to $\mathbb{R}\cup\{\infty, -\infty\}$. Let $\mathcal{O}(\mathcal{D})$ be the set of all total orderings of $\mathcal{D}$.

A pBP algorithm $\mathcal{A}$ for a given optimization problem $\Pi = (\mathcal{D}, \Sigma, F)$ is defined by specifying the set of states and three components for each state. In addition, it can also specify $\sigma_{def} \in \Sigma$, a default option. Let $s \in \mathcal{S}$ be any state, then the algorithm defines:

1. A priority ordering,

$$\pi_s \in \mathcal{O}(\mathcal{D} \cup \{end\}),$$

This is used to determine which data item is branched on in this state.

2. A state transition function

$$g_s : \mathcal{D} \times \Sigma \mapsto \mathcal{S} \times \mathcal{M}(R) \cup \{\perp\}.$$

If d, σ maps to s', f, we think of this as defining a directed edge (s, s') labeled by (d, σ, f). We insist that the graph induced on $\mathcal{S}$ by $\{g_s\}$, which we call $\mathcal{DAG}_{\mathcal{A}}$, be a (multi)dag. As we shall see later, this requirement is usually ensured by imposing a natural layered structure on the graph. We can interpret such an edge intuitively as: if, recursively, we find a value v solution for the sub-problem at s', we can obtain a solution to the sub-problem at s of value $f(v)$ by appending (d, σ). Also, any transition where $d = end$ must go to some $s' \in \mathcal{T}$.

3. For $s \in \mathcal{T}$, the algorithm defines a value

$$v_s \in \mathbb{R} \cup \{\infty, -\infty\}.$$

Intuitively, this represents "The value returned in the base case s".

The computation of a pBP algorithm on a given instance traces out a subgraph of the above graph. For a pBP algorithm $\mathcal{A}$, we define the computation of $\mathcal{A}$ on instance

$I \subset D$ as follows. A pBP computation has three stages: build the (multi)dag, output the value of the best solution, and output the solution itself.

Stage I Build the pBP DAG. The algorithm $\mathcal{A}$ builds $\mathcal{DAG}_{\mathcal{A}}(I)$ top down. Let $\mathcal{S}_1 = \{S_0\}$ and assume that at step $i \in 1, 2, 3...$ we have a set of frontier states $\mathcal{S}_i$. For each node $s \in \mathcal{S}_i - \mathcal{T}$, do the following: let $d \in I$ be the first item in I according to the ordering π_s. For each $\sigma \in \Sigma$, check if $g_s(d, \sigma) = \perp$; if not, say it equals (s', f). Then, put s' in $\mathcal{S}_{i+1}$ and include both s' and the edge (s, s') labeled by (d, σ, f) in $\mathcal{DAG}_{\mathcal{A}}(I)$. It is the algorithm's responsibility to ensure that for some finite i, $\mathcal{S}_i$ is empty and that all sinks in the graph are in $\mathcal{T}$. Note that the only source in the graph is S_0.

Stage II Compute the value of the *best* solution.

The second stage begins after the $\mathcal{DAG}_{\mathcal{A}}(I)$ is fully defined. It determines the value of the optimal solution. This stage traverses the $\mathcal{DAG}_{\mathcal{A}}(I)$ bottom up and successively computes values for each state. The value of the start state S_0 will be the value of the algorithm's solution. First, consider any sink s in $\mathcal{DAG}_{\mathcal{A}}(I)$. Since $s \in \mathcal{T}$, the algorithm defines a value v_s. We assign this value to s. Now consider any s in $\mathcal{DAG}_{\mathcal{A}}(I)$ that does not yet have a value but such that all of its children do. Let the outdegree of s be k and let $s_1, \ldots, s_k$ be its (not necessarily distinct) children. They have assigned values $val(s_1), \ldots, val(s_k)$. Finally, assume that the edge (s, s_i), $i \in [k]$, is labeled by the monotone function f_i. Now we compute $val(s)$ as

$$\max\{f_1(val(s_1)), \ldots, f_k(val(s_k))\}.$$

When computing the value of each s, we also remember which of its outgoing edges contributed the maximum value by marking that edge. If there is a tie, it is broken arbitrarily; that is, the algorithm has no control over which of the maximum-value edges is chosen. Hence there will be one marked edge out of every non-sink in $\mathcal{DAG}_{\mathcal{A}}(I)$.

Stage III Recover the *best* solution.

The third stage recovers the actual solution by traversing $\mathcal{DAG}_{\mathcal{A}}$ top down by following the marked directed path from S_0 until a leaf state is reached. This path is well-defined and gives, for each of its edges, an assignment of a decision to an item. This partial assignment is then extended to a complete assignment by assigning any unlabeled item the default label, σ_{def}. The algorithm must ensure that this assignment is consistent (in that the same data item is not assigned different options), and that when the problem's optimization function $F_{\mathcal{P}_n}$ is applied to this assignment, it yields the same value that the algorithm reported in Stage II.

It is clear that the pBP model subsumes the pBT model. How does this model capture some notion of dynamic programming? Our model resembles the recursive-memoization implementation of dynamic programming. The nodes of the DAG represent sub-solutions only to those subproblems which we definitely need and the set of states reachable in the DP dag represent exactly those subproblems that are needed during the computation.

Just as in the pBT case, a pBP algorithm can solve any optimization problem by considering the items in some fixed order and exploring each possible assignment of decisions to those items on a separate path. So, similarly to the pBT case, the complexity measure of interest will be the number of states used in any execution of the algorithm. More formally, define

$$size_{\mathcal{A}_n} = \max\{|\mathcal{DAG}_{\mathcal{A}}(I)| \quad : \quad I \subset D_n,\ I \text{ is a valid instance of } \mathcal{P}_n\},$$

where $|G|$ denotes the number of states in G. Call a family of pBP algorithms $\{\mathcal{A}_n\}$ *polynomial* if there exists a polynomial p such that $size_{\mathcal{A}_n} \leq p(n)$ for all n.

As was the case for priority, pBT, and pFBT algorithms the functions used in this model have only information-theoretic restriction and no computational limits are imposed.

5.1.2 Submodels of the pBP Model

The pBP model defined above is quite general. Here we will consider several natural ways to refine the model. These refinements will help us to classify more precisely various dynamic programming algorithms.

The first four restrictions deal with how many times an algorithm may view the same data item on a single path and what it may do with that data item. Any valid algorithm in the model must return a consistent path, but we may want a stronger consistency requirement.

Definition 26. *(Read-Once) Consider any path in the computation DAG from the root state to a terminal state. Suppose each data item appears at most once on this path, then this path satisfies the Read-Once (RO) property.*

Definition 27. *(Path Consistency) Call a path from the root state to a terminal state of the DAG* **consistent** *if for each data item $d \in \mathcal{D}$ there are not two edges along the path labeled $(d, \sigma, ...)$ and $(d, \sigma', ...)$, respectively, where $\sigma \neq \sigma'$.*

Definition 28. *(Syntactic consistency) We call a pBP algorithm $\mathcal{A}$ syntactically consistent if, for every instance I, every path from the root node S_0 to a terminal state in $\mathcal{DAG}_{\mathcal{A}}(I)$ is consistent.*

Definition 29. *(Semantic consistency) We call a pBP algorithm $\mathcal{A}$ semantically consistent if, for every valid instance I, the optimal path from the root node S_0 to a terminal state in $\mathcal{DAG}_{\mathcal{A}}(I)$ is consistent.*

Definition 30. *(Syntactic RO property) We call a pBP algorithm $\mathcal{A}$ syntactically RO if, for every instance I, every path from the root node S_0 to a terminal state in $\mathcal{DAG}_{\mathcal{A}}(I)$ is RO.*

Definition 31. *(Semantic RO property) We call a pBP algorithm $\mathcal{A}$ semantically RO if, for every valid instance I, the optimal path from the root node S_0 to a terminal state in $\mathcal{DAG}_{\mathcal{A}}(I)$ is RO.*

Definition 32. *(Honest pBP algorithm) For the path found in phase 3, our pBP definition guarantees that the data items and decisions on the edge labels along the path code a solution which has a value equal to $f_1(f_2(\dots(f_k(v))\dots))$, where f_i is the function labeling the i'th edge in the path, and v is the value assigned to the terminal state the path ends in. Intuitively, this should also be true for non-optimal solutions. Call an algorithm* honest *if the same is true for any path from the start state to a terminal state.*

The next figure shows natural containments of the subclasses defined above. Let SynCons, SemCons, SynRO and SemRO denote the class of problems that can be solved by polynomial pBP algorithms that have the syntactic consistency, semantic consistency, syntactic read-once and semantic read-once properties, respectively. SemCons is the most powerful because it can trivially simulate the rest. The weakest class of algorithms is the class of SynRO.

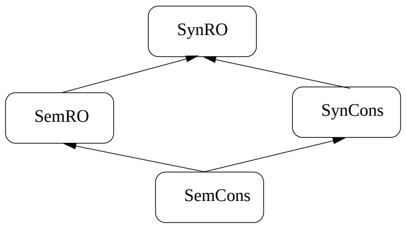

Figure 5.1: Submodel Lattice

We do not know whether the containments are proper.

We can also restrict the variety of orderings that the states use. The following three variants also appeared in the case of the pBT model. Before we describe them, however, one structural point is in order: given a pBP algorithm $\mathcal{A}_n$ for a problem $\mathcal{P}_n$, we can create an algorithm $\mathcal{A}'_n$ such that $\mathcal{D}AG_{\mathcal{A}'_n}$ is leveled where $size_{\mathcal{A}'_n} \le n \cdot size_{\mathcal{A}_n}$ if A_n is semantic read-once or $size_{\mathcal{A}'_n} \le (\max_I \ depth(\mathcal{D}AG_{\mathcal{A}_n}(I))) \cdot size_{\mathcal{A}_n} \le (size_{\mathcal{A}_n})^2$ in general. To do this, let d be the depth of $\mathcal{D}AG_{A_n}$. The new state space will be $\mathcal{S} \times [d]$, where the levels are $\mathcal{S} \times \{i\}$ for each i, and we will make each edge increase exactly one level. Hence, we will often assume that $\mathcal{DAG}_{\mathcal{A}}$ is levelled.

1. Fully Adaptive (order) pBP algorithms.

   Each state can have an entirely arbitrary ordering.

2. Adaptive (order) pBP algorithms.

   Consider the DAG $\mathcal{DAG}_{\mathcal{A}}(I)$ defined by the algorithm $\mathcal{A}$. Here we require that for each instance I, all states at the same level use the same ordering. Hence, in any computation, all paths of the same length from the root will include the same data items. Note that such algorithms are either syntactic read-once or are not read-once at all.

3. Fixed (order) pBP algorithms.

   All states at the same level have the same ordering and that ordering is the same as the previous level's ordering except that the item viewed at the previous level is moved to the end of the ordering, i.e., the data items after "end" in the ordering are precisely the previously viewed data items, and the other data items are in the same order as in the start state. In essence, all states use the same ordering, but technically that ordering must be updated in each level so that the same input item isn't viewed repeatedly forever. Such algorithms are syntactic read-once.

Again, define Fixed pBP, Adaptive pBP and Fully Adaptive pBP, respectively, to be the classes of problems that can be solved by polynomial pBP algorithms with the corresponding property. It is clear that Fixed pBP $\subseteq$ Adaptive pBP $\subseteq$ Fully Adaptive pBP. While we will see strong evidence that this hierarchy is strict, it remains an open question.

5.2 Examples

1. Longest Increasing Subsequence problem

   Consider the following pBP formulation for the longest increasing subsequence. The instance is an array of integers A of length n, coded as follows. A data item is a pair (a, i), where $a \in \mathbb{Z}$ and $i \in \{1, \ldots, n\}$ is the position in the array A, where a appears. A valid instance is a set of n data items in which each i from 1 to n occurs exactly once. For short we will use array index notation and will refer to the data item as $a[i]$. The set of decisions is $\Sigma = \{0, 1\}$, where 0 means that the current number is not chosen to be part of the LIS, and 1 means it is chosen. The computation states are all pairs (B, i), where B is a sequence of integers of length at most n and $0 \leq i \leq |B|$ is a natural number. The intended meaning is that B is the prefix of the input A viewed so far and i is the index of the rightmost element of that prefix chosen to be in the LIS. A terminal state is a state which has observed the full input, therefore is identified with (B, i), where $|B| = n$ and $0 \leq i \leq n$.

- The algorithm will use a fixed ordering on the items. That (initial) ordering puts all items with index 1 first, followed by all items with index 2, etc. Within each index partition, the items are ordered arbitrarily.
- Consider a state $s = (B, i)$, and a new item (a, j). If $j \neq |B| + 1$, then the instance is invalid and we may as well terminate. Otherwise, $g_s((a, j), 0) = ((B \circ a, i), f(x) = x)$. If $i = 0$ or $a \geq B[i]$, then $g_s((a, j), 1) = ((B \circ a, j), f(x) = x + 1)$; otherwise $g_s((a, j), 1) = \perp$.
- All terminal nodes will have value 0.

Below we give an example of the DAG built by the algorithm for the input $5, 1, 2, 3$: The forward edges are labeled with the data item and the decision made by the algorithm on each data item. The edges are labeled with $x + 1$ if the decision was to accept the data item and x otherwise. The best solution has length 3 and the bold backward arrows indicate the path of the winning solution. Following the bold path from the root to the leaf we can recover the solution as $1; 2; 3$. Each level will have at most n states in any computation, so the size of the above algorithm is at most n^2.

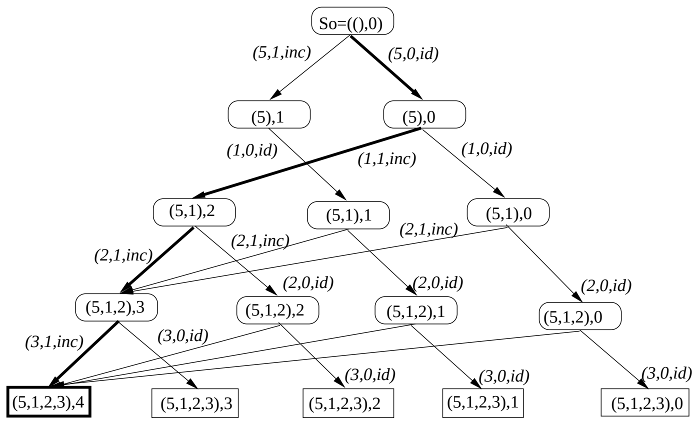

Figure 5.2: pBP DAG for LIS on instance $5, 1, 2, 3$

Basically the same solutions can be implemented in fixed order pBT with the same size, similarly as Example 3.1 from Chapter 3. Both problems view data items in fixed predetermined order, hence each branch of the tree knows whether the subsolution along it is to be subsumed or is dominated by another branch.

2. Single Source Shortest path in graphs with negative weights

We consider a simplified version of the shortest paths problem, where we are interested in the path starting at a specified source, s, of smallest total length (to any destination) in a weighted directed graph that contains no negative-weight cycles. Because the solution to the problem is a subset of edges which form a path, it is natural to use the *edge model*. Here each data item is an edge, represented as the names of the two end points and the weight (u, v, ℓ), where the names of nodes are the numbers in $[n]$. (Note that each graph yields a set of data items, but only those subsets of data items that code graphs with no negative cycles are valid.) The set of decision options is $\Sigma = \{1, 0\}$, meaning accepted and rejected, respectively. The number of vertices, n, and the name of the source, s are part of the problem definition and hence are known to the algorithm. We will implement a version of the well-known Bellman-Ford algorithm in the pBP model. Note that in the absence of negative weights, a priority algorithm can solve the problem by implementing Dijkstra's algorithm ([20]).

- The default label is 0; all edges not explicitly accepted along the final output path will be considered rejected.
- A computation state is encoded as the name of the currently reached vertex, u, the last vertex we rejected going to from u, v (with $v = 0$ meaning no edges have yet been rejected), and an upper bound, k, on the length (number of nodes) on the path from s to the current vertex, which is always less then or equal to $n-1$.

  A terminal states are a special "no more neighbors" state or one where the value of k is $n-1$.

  The start state is $(s, 0, 0)$.

  From the definition it is clear that the cardinality of the set of states is $O(n^3)$, because there are n nodes and the number of possible lengths is $n-1$.
- Every terminal node $(u, v, k = n-1)$ and "no more neighbors" has value 0.
- Consider a computation state $a = (u, v, k)$, where $k < n-1$. The order π_a will first put all items of the form (u, v', ℓ) where $n \geq v' > v$ in order of increasing v' and put all other items after "end". Data items with the same v' are ordered arbitrarily.
- Again, consider state a and assume $(u, v', \ell), v' > v$ is the data item viewed. Set $g_a((u, v', \ell), 0) = ((u, v', k), f(x) = x)$, and set $g_a((u, v', \ell), 1) = ((v', 0, k+1), f(x) = x + \ell)$. Otherwise, the data item viewed is "end", and we go to the "no more neighbors" state and terminate, with $f(x) = 0$ for the transition.

We showed that the LIS example can efficiently be implemented by a pBT. Note that the computation will be semantically read-once, but not syntactically read-once. It seems difficult to explore paths in a graph using a small number of computation states without allowing for viewing the same edge more than once on a given computation path because the states cannot encode which nodes have already been visited. It also seems necessary to use full adaptivity.

Next we illustrate the computation of the pBP algorithm for SSSP problem with the following example. Consider the graph on Figure 5.3. We want to find the

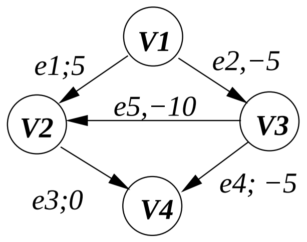

Figure 5.3: Example SSSP instance

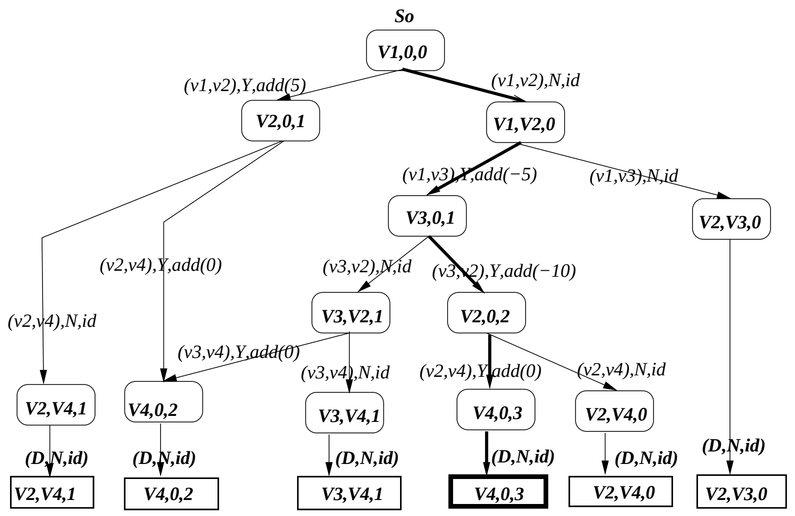

Figure 5.4: pBP DAG for the instance graph on Figure 5.3

shortest path from s to any other vertex in the instance. Each edge has a label and weight. We want to find the shortest path from s to any vertex in G. For this instance there are three paths from s to t each of length 2. The path e_1, e_3 has total weight 5, while e_2, e_4 has weight -10, and e_2, e_5, e_3 has length -15.

3. TSP

The fastest known exact algorithm for general TSP is due to Held and Karp. The idea is as follows. Let the set of vertices be $[n]$, the start node of the tour be 1, the end node of the tour be n. For every vertex $i \in [n]$ and every subset $U \subset [n] \backslash \{i, n\}$, $OPT_{U,i}$ is the shortest tour beginning at i, finishing at n, visiting all the nodes in U exactly once, and not visiting any nodes outside $U \cup \{i, n\}$. Intuitively, what makes TSP susceptible to dynamic programming is the fact that

$$OPT_{U,i} = \min_{j \in U} \{OPT_{U-\{j\},j} + w(i, j)\}.$$

To put this algorithm in our model, we consider the problem in the edge model, where each data item is an edge of the graph with a weight. We explore paths originating at node 1 just as we did in the previous example, except now states will keep track not just of the current node and how many steps it took to get there, but of the complement of the set of nodes visited so far. For this reason, the pBP algorithm will be syntactic read-once. More precisely, just as in the recurrence above, a state will be encoded by a tuple (U, i), where $U \subset [n]$ and $i \in [n]$. Technically, there will be a little overhead, just as in the previous example, so that, if we are currently at vertex i, we can explore all edges leaving i. There will be $O(n^2 2^n)$ states. The terminal states will be those that have current node n; those with empty subset U represent successful TSPs, while those with nonempty U represent failed traversals. The remaining details are quite analogous to the shortest path example.

- Each computation state has two components. The first is a subset of the vertices of the graph, and the second is a vertex name. This corresponds directly to the recurrence above. In addition, there will be one special state called FAIL, which we will use if we detect there is no TSP. Hence for a graph of order n the number of computation states is $O(n2^n)$
- The set of terminal states will be those that have as a second component

the target of the tour n, plus the state FAIL.

- The ordering function initially would consider edges incident to vertex n. If we view the DAG level by level, at level i we will have subsets of size $n-i$. The ordering function at state (U,i) will explore all edges incident to i.
- State generation function. Say we are at state (PI,u) and have considered and accepted edge (u,v) then the state $g_{(PI,u),(u,v),accepted} = PI \cup \{u\}, v$.

The value function, the backward labeling function are the same as in the previous example.

5.3 Simulations of pBPs by pBTs

We show that, under certain natural conditions, fixed and adaptive pBT algorithms can simulate fixed and adaptive pBP algorithms, respectively, without an increase in size. In the next section, we show that this is not the case for fully-adaptive pBP.

First, a subtle point: our definition of pBT algorithms requires that, for a pBT algorithm $\mathcal{A}$, $\mathcal{DAG}_{\mathcal{A}}$ must be a tree. It is not hard to see that, given the weaker condition that $\mathcal{DAG}_{\mathcal{A}}(I)$ is a tree for every instance I, we can create an algorithm $\mathcal{A}'$ that satisfies the stronger condition without increasing the complexity. More generally, if two paths merge in $\mathcal{DAG}_{\mathcal{A}}$ but there is no instance I such that both of those paths appear in $\mathcal{DAG}_{\mathcal{A}}(I)$, then we may as well make two copies of the state where the merge occurred. The point is that increasing the size of the state space $\mathcal{S}$ does not hurt the complexity of the algorithm if there is no instance which uses both copies of the original state.

Lemma 45. *Let $\mathcal{A}$ be a fixed (respectively, adaptive) order pBP algorithm for optimization problem $\mathcal{P}$. If the monotone functions labeling the edges of $\mathcal{DAG}_{\mathcal{A}}$ are all linear functions of the form $x+c_e$, where c_e is a constant that depends on the edge e, then there is a fixed (respectively, adaptive) order pBT algorithm $\mathcal{B}$ for $\mathcal{P}$ such that, $size_{\mathcal{B}} \leq size_{\mathcal{A}}$.*

Proof. $\mathcal{B}$ simulates $\mathcal{DAG}_A$, but remembers the following information: the partial instance PI, the partial solution PS, the state in $\mathcal{DAG}_A$ reached, and the sum of the weights c_e along the transitions on its current path. At depth t, it views the same input as $\mathcal{DAG}_A$ at this state, but before simulating a transition, it uses PI, which is the same along every other path, to simulate all other $t+1$ step paths of $\mathcal{DAG}_A$; if any of them reach the same state with a smaller sum of transition constants, or if a lexicographically prior partial solution gives the same sum and the same state, then that branch is pruned. If it reaches a terminal state, then it assigns all unassigned data items the default value, and terminates with value equal to the objective function at the (now total) solution.

Since $\mathcal{B}$'s states remember PI and PS, they form a tree. If $\mathcal{B}$ on I simulated two paths that arrived at the same state, either one has a smaller sum of transition constants, or both have the same sum, and one has a lexicographically prior partial solution. Therefore, one of the two will be pruned. Thus, there is at most one unpruned path in $\mathcal{B}$ per reachable computation state in $\mathcal{DAG}_A(I)$, so the width is at most the complexity of $\mathcal{DAG}_A$. Finally, consider an algorithm that, when two transitions give a state s the same value, breaks ties according to the lexicographical ordering on the label. Then in the third phase of the algorithm, if from s_0 we reach node v, the chosen path will always have the highest sum of transition constants among possible paths from s_0 to v (otherwise, whatever value v is assigned, a higher value is assigned to s_0 along the other path) and among such paths, it will be the lexicographically first. Thus, the optimal solution's path, which is found in phase 3 of $\mathcal{DAG}_A$, is not pruned in $\mathcal{B}$, so in returning the best of the solutions found at its branches, $\mathcal{B}$ will return an optimal solution. $\square$

As a corollary of this theorem, we see that the LIS and the LCS problems can be solved by a polynomial width fixed order pBT algorithm.

The following lemmas show another condition under which pBT algorithms can simulate FIXED or ADAPTIVE pBP algorithms for search problems.

Given a maximization $\mathcal{P}$, let $\mathcal{P}^v$ denote the same problem except that $\mathcal{F}_{\mathcal{P}^v}$ returns 1 on those solutions where $\mathcal{F}_P$ is at least v, and 0 otherwise.

Lemma 46. *Let $\mathcal{A}$ be an honest fixed (respectively, adaptive) order pBP algorithm for optimization problem $\mathcal{P}$. For any $v = v(n)$, consider the corresponding search problem $\mathcal{P}^v$. Then there is a fixed (respectively, adaptive) order pBT algorithm $\mathcal{B}$ for this search problem such that, $size_{\mathcal{B}} \leq size_{\mathcal{A}}$.*

Proof. $\mathcal{B}$ simulates $\mathcal{A}$, remembering the partial instance PI, and the partial solution PS, as well as the current state in $\mathcal{DAG}_A$, and a value v_t given by: $v_0 = v$ and $v_t = f_t^{-1}(v_{t-1})$, where f_t is the monotone function labeling the transition at time t. As before, when $\mathcal{B}$ simulates the transition, if there is a path reaching the same state of $\mathcal{DAG}_A$ and yielding a smaller value of v_t or if there is a lexicographically prior such a path giving an equal value for v_t, the path is pruned. When we get to a terminal node, we give it value 1 if $\mathcal{A}$ gives it value at least v_t and 0 otherwise. Hence, at most one path that reaches any given state of $\mathcal{DAG}_A$ is not pruned, so the total size for $\mathcal{B}$ is at most that of $\mathcal{A}$.

Assume s_0 is given value 1 in $\mathcal{B}$. Then phase 3 returns a path in $\mathcal{B}$ $s_0, s_1, \ldots, s_t$ with s_t a terminal, where all nodes along the path are given value 1. Each transition in $\mathcal{B}$ corresponds to a transition in $\mathcal{A}$, so we can look at the series of functions $f_1, \ldots, f_t$ labeling the edges in this path in $\mathcal{A}$, and the series of values $v_0, \ldots, v_t$ defined $v_0 = v, v_i = f_i^{-1}(v_{i-1})$. Then since s_t is a terminal given value 1, in $\mathcal{A}$, s_t is assigned a value $\geq v_t$. Assume that s_{i+1} is assigned a value in $\mathcal{A}$ which is $\geq v_{i+1}$. Since the value of s_i is the maximum of a number of terms, one of which is $f_i(v(s_{i+1}))$ and f_i is monotone, s_i is assigned a value in $\mathcal{A}$ at least $f_i(v_{i+1}) = f_i(f_i^{-1}(v_i)) \geq v_i$ by definition of inverse. Thus, $f_1(f_2(\ldots(v(s_t))\ldots)) \geq v$, which, by the honesty condition implies that the edges on the path code a solution of value at least v. So if $\mathcal{B}$ claims there is a solution, it successfully solves the search problem.

Then assume there is a solution of value $\geq v$. Let $s_0, \ldots, s_t$ be the path returned by $\mathcal{A}$ in phase 3, coding an optimal solution (and hence one of value $\geq v$). Define the series of values $v_0, ..v_t$ by $v_0 = v, v_i = f_i^{-1}(v_{i-1})$, where f_i labels the transition. Since $\mathcal{B}$ only aborts a path to s_i when it finds another path where the composition of the inverse functions on v, w_i, along that path is as small, $\mathcal{B}$ contains paths to each state of $\mathcal{A}$ with search value w_i equal to the smallest such composition of any path to s_i. Thus, it will have a path to s_t of label at most v_t, and will thus give s_t value 1. Since the value of the root s_0 in $\mathcal{B}$ is the or of the values of the terminal states, this means that B returns a 1 in phase 2. By the previous paragraph, this means it finds a solution of value at least v in phase 3. $\square$

Lemma 47. *Let $\mathcal{A}$ be a pBP algorithm for a problem $\mathcal{P}$ and let $V \subset \mathbb{R}$ be a finite set such that, for every instance I and every state s in $\mathcal{DAG}_{\mathcal{A}}(I)$, the value of s computed in phase II is contained in V. Let w be the maximum value in V. There exists a pBP algorithm $\mathcal{B}$ for $\mathcal{P}^w$ where $size_{\mathcal{B}} \leq |V| size_{\mathcal{A}}$ which uses the identity function as its only monotone function. Furthermore, if $\mathcal{A}$ is fixed (respectively, adaptive) order, then $\mathcal{B}$ will also be fixed (respectively, adaptive) order.*

Proof. Let $\mathcal{S}$ be the state space of $\mathcal{A}$. The state space of $\mathcal{B}$ will be $((\mathcal{S} \backslash \{S_0\}) \times V) \cup \{S_0\}$. If t is a terminal state for $\mathcal{A}$ that returns value $val(t)$, then for all $v \in V$, (t, v) will be a terminal state for $\mathcal{B}$ that returns 1 if $v \leq val(t)$ and 0 otherwise. In general, if there is an edge (s, s') in $\mathcal{DAG}_{\mathcal{A}}$ labeled by (d, σ, f), then, for $v, v' \in V$, there will be an edge $((s, v), (s', v'))$ labeled by (d, σ, id) if $v \leq f(v')$. The root state of $\mathcal{B}$ will be S_0. If there is an edge (S_0, s') in $\mathcal{DAG}_{\mathcal{A}}$ labeled by (d, σ, f), then, for $v' \in V$, there will be an edge $(S_0, (s', v'))$ labeled by (d, σ, id) if $w \leq f(v')$. If a state (s, v) in $\mathcal{B}$ has no children and s is not a terminal state in $\mathcal{A}$, then make (s, v) a terminal state in $\mathcal{B}$ with value 0.

Given any instance I, it is clear by induction on the height of a state s in $\mathcal{D}AG_{\mathcal{A}}(I)$ that the state (s, v) will achieve value 1 on instance I in $\mathcal{B}$ if and only if state s achieved value at least v on instance I in $\mathcal{A}$. $\mathcal{B}$ will choose as its solution an arbitrary path from S_0 to a sink in $\mathcal{D}AG_{\mathcal{B}}(I)$ labelled $(d_1, \sigma_1), \ldots, (d_k, \sigma_k)$ such that every state along this path achieves value 1 (if there is such a path). This path corresponds to a path in $\mathcal{A}$ that contributed value w to the state S_0. Since every such path in $\mathcal{A}$ must constitute a valid solution of value w (note this is not necessarily true in a non-honest algorithm if w is not the max value), so must this path. $\square$

Note that, in the previous lemma, if $\mathcal{A}$ was fixed or adaptive order, then $\mathcal{B}$ satisfies the assumptions of Lemma 45. Hence there is a pBT algorithm with the same size computing $\mathcal{P}^w$ in this case.

We would like to point out that most natural implementations of dynamic programming algorithms as pBPs seem to satisfy the conditions of Lemmas 45 and 46 (certainly those in Section 5.1 do). This is strong evidence that the separation between fixed order and adaptive order pBTs given in [1] also holds for pBPs. In Chapter 4 we showed that pBT algorithms require exponential width to solve shortest paths in graphs with negative weights while pBP can solve the problem using $O(n^3)$ states, thus separating the models.

5.4 Notes

Manuscript [17] "A Stronger Model for Dynamic Programming Algorithms" by Josh Buresh-Oppenheim, Sashka Davis, and Russell Impagliazzo, contains the pBP model and the simulation results presented in this chapter. In addition, in [17], it was shown that pBP algorithms require $2^{(\Omega n^{1/9})}$ states to find a perfect matching in bipartite graphs. Flow algorithms can find perfect matching in bipartite graphs in polynomial time, thus exponentially separating the power of flow algorithms from pBP algorithms.

Chapter 6

Open Questions

Figure 6.1 shows the current formal models and how they relate to the existing basic algorithmic paradigms. An edge from model A to model B is present if every algorithm in model B can be simulated (usually at no cost) by an algorithm in model A. To show a proper containment we need to exhibit a problem for which we can obtain a lower bound on the resources for all algorithms in class B, of say $\Omega(n)$, and to exhibit an algorithm in class A which uses $o(n)$ resources. If a separation is achieved then the arrow is labeled by $\subsetneq$ together with the problem which separates the models.

Currently we have four formal models: Priority algorithms, prioritized Branching Tree algorithms (pBT), prioritized Free Branching Tree algorithms (pFBT), and prioritized Branching Programs (pBP). Priority algorithms were introduced by [14] as a formal model of greedy algorithms for scheduling problems. In this dissertation we extended the Priority model to a model of greedy algorithms for arbitrary problem domain. The pBT model of [1] captures the power of backtracking algorithms and many of the known dynamic programming algorithms. [1] showed that the pBT model is exponentially stronger than the Priority model. In this dissertation we extended the pBT model by allowing free branching. We showed that pBT algorithms required $2^{\Omega(n)}$ width to solve a family of hard 7-SAT instances. We further proved that pFBT algorithms required $2^{\Omega(\sqrt{n})}$ width to solve the same family of hard 7-SAT instances. Jeff Edmonds

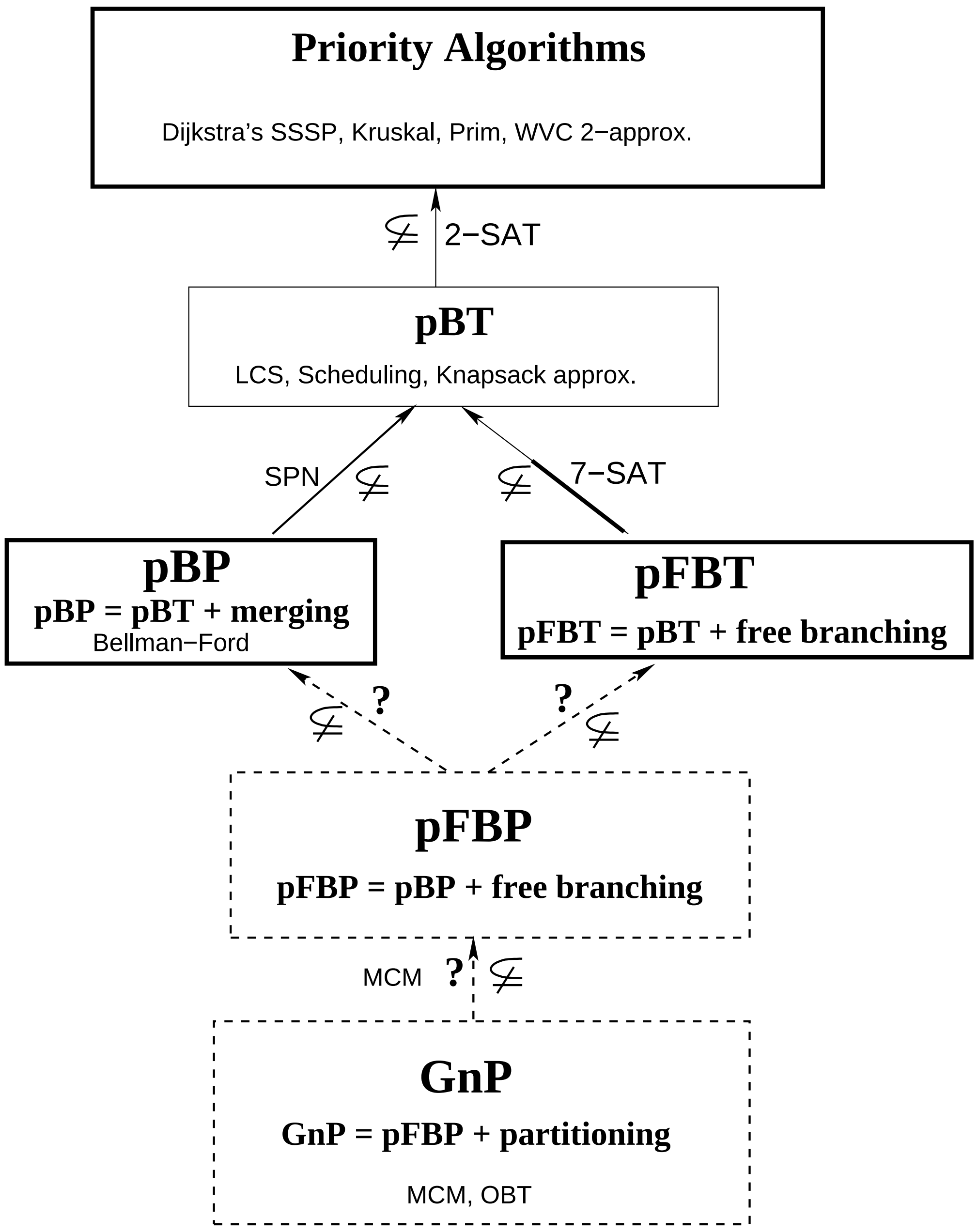
Priority Algorithms
Dijkstra's SSSP, Kruskal, Prim, WVC 2−approx.
⊈
2−SAT
pBT
LCS, Scheduling, Knapsack approx.
SPN
⊈
⊈
7−SAT
pBP
pBP = pBT + merging
Bellman−Ford
pFBT
pFBT = pBT + free branching
⊈
?
?
⊈
pFBP
pFBP = pBP + free branching
MCM
?
⊈
GnP
GnP = pFBP + partitioning
MCM, OBT

Figure 6.1: Lattice of the algorithmic models. SPN stands for the problem of finding shortest paths in graphs with negative weights; LCS stands for the longest common subsequence problem; MCM stands for matrix chain multiplication problem; OBT stands for the optimal binary tree problem.

developed a width $2^{O(n \log n)}$ pFBT algorithm for the same family of hard instances. This establishes an exponential separation between pBT and pFBT algorithms. The classical dynamic programming algorithm of Bellman-Ford for solving shortest path in graphs with negative weights but no negative weight cycles could not be seen to fit any of the existing models (Priority, pBT, pFBT) so we proposed a new model pBP, which is capable of solving the shortest path problem in graphs with negative weights and no negative weight cycles using $O(n^3)$ states. We further showed that any pBT algorithm requires width $2^{\Omega(n^{1/9})}$ to solve the shortest paths in graphs with negative weights, thus separating the pBT and the pBP models. The open questions roughly fall into four categories.

6.1 Inside the Existing Models

Although many non-trivial lower and upper bounds have been obtained for the four known formal models, the work towards understanding their true power and limitations is all but done. The Priority model seems a flexible and useful tool for understanding the limitations of the simple greedy algorithms. However, some "intuitively greedy" algorithms fall outside our model. One such natural extension will be to consider a model where the algorithm is given access to "global information", or to redefine the notion of *local information* associated with a data item. For example consider graph problems, suppose the type of data item encodes not just the names of the neighbors, but also neighbors of the neighbors of a node as well, assuming the problem is viewed in the node model. Would that additional information, given to the algorithm during the decision-making process, increase the power of the Priority algorithms? What kind of lower bounds can we prove for those models?

The pFBT algorithms were shown to be exponentially more powerful that the pBT algorithms. What kind of upper and lower bounds can we obtain for pFBT for other optimization problems, say independent set, subset sum, maximum clique? What kind of lower bounds on the approximation ratio by such algorithms can we prove for TSP?

There are two “standard” approaches to apply dynamic programming technique and solve hard problems approximately: *trimming-the-state-space* technique (introduced by Ibarra and Kim [29], see also [51]) and *rounding-the-input* technique (introduced by [47], see also [51]). Can we develop lower bound techniques for the pBP and the pFBP models and use them to show interesting lower bounds on the approximation ratio for the standard dynamic programming approaches to approximation for various optimization problems?

6.2 Are the Containments Proper?

pFBT algorithms extended the pBT model by allowing free branching, similarly the pBP model could be extended by allowing free branching and it is not difficult to see that a pFBP algorithm could simulate the Bellman-Ford algorithm using $O(n^2)$ states. Free branching was proven to give additional power to pFBT algorithms so we ask the analogous question. Are pFBP algorithms more powerful than pBP algorithms or does there exist a non-trivial simulation of pFBP algorithms by pBP algorithms?

$$(\text{pFBP} = \text{pBP}) \ \text{ or } \ (\text{pBP} \subsetneqq \text{pFBP})?$$

The pBP model extended the pBT model by allowing *merging*, and we showed that merging is essential and it makes the class of pBP algorithms exponentially more powerful than the pBT algorithms. Does merging make pFBP more powerful than pFBT algorithms or not?

$$(\text{pFBT} = \text{pFBP}) \ \text{ or } \ (\text{pFBT} \subsetneqq \text{pFBP})?$$

6.3 Beyond pFBP

While many dynamic programming algorithms we have encountered fit into the pBP model, some do not. For example, the dynamic programming algorithm for building optimal binary tree (OBT) and the algorithm for the matrix chain multiplication

(MCM) are not seen to fit in any of the existing models. Both algorithms seem to utilize two paradigms the divide-and-conquer approach together with dynamic programming. The pBP model does capture the notion of memoization but the divide and conquer employs partitioning of the problem into smaller subproblems and solving those smaller subproblems independently. Therefore if the pFBP model can be extended to allow for partitioning of the instance then, such a model will be able to model dynamic programming algorithms utilizing divide and conquer strategies as well. The last box on Figure 6.1 represents our candidate for such a model. GnP which stands for *Guess* and *Partition*, GnP $=$ DnC $+$ DP, will be a model for dynamic programming and divide and conquer paradigms. If the model could guess a partitioning point on the input and then use a pFBP to solve the subproblem, while memoizing common subproblems then such a model would simulate the dynamic algorithms for MCM and OBT problems.

6.4 Beyond the Basic Algorithmic Paradigms

Greedy, backtracking and dynamic programming algorithms deliver exact solutions and elegant approximation schemes for variety of optimization problems. However, the most powerful techniques we have are the linear programming relaxations and semi-definite programming relaxations. The basic algorithmic paradigms have a common property that they all have a *local access to the input instance.* For example, the graph is inspected one edge at a time, regardless of how many decision options would be explored; in the context of scheduling problems we consider one job at a time and make scheduling decision about it without considering the remaining instance. The same does not hold true for say flow algorithms, LP relaxations or SDP relaxations. In our models if the algorithm knows the entire instance then it can solve any problem. Hence, it is not clear how the techniques we have developed here would help to analyze those advanced approaches to optimization which work on the solution space. Formalizing and analyzing those advanced approaches to optimization will give a clear and precise answer of the weaknesses of our techniques, and perhaps it would be a staring point for development of new innovative approaches to algorithm design.

Appendix A

Priority Formalization for Adaptive Contract

Recall the formal definition of ADAPTIVE priority algorithms. An ADAPTIVE priority algorithm proceeds in rounds. At the beginning of round t the data items not yet observed are sorted according to a priority function π_t. Then the π_t-first data item is considered and an irrevocable decision is made about it. Here we translate the algorithm Adaptive Contract from Section 2.4.3 in this framework.

The priority model for the Steiner Tree problem is as follows: The input to the algorithm is the edge set of the graph. Each edge e is a 5-tuple (v_1, t_1, v_2, t_2, w), where v_1, v_2 are the names of the end points, t_1, t_2 are the types, $t_1, t_2 \in \{\mathbf{r}, \mathbf{s}\}$, r for required, and s for Steiner, and a weight $w \in [1; 2]$. The decisions the algorithm can make are $\Sigma = \{accept, reject\}$.

Algorithm **Adaptive Contract**

Input: Sequence of edges I: $\forall e \in (v_1, t_1, v_2, t_2, w), w \in [1; 2]$

Output: Edges accepted form a Steiner tree.

1. Initialization:

- Iteration counter: $t \leftarrow 1$;
- Counter for the number of required nodes in the instance: $N_R \leftarrow -\infty$;
- Indicator for the number of Steiner nodes present in the instance. Will be set to: 0 if the instance has no Steiner nodes; 1 if there is only one Steiner node; a value greater than or equal to 2 if there are 2 or more Steiner nodes in the instance: $N_S \leftarrow -\infty$;
- Set of Steiner nodes: $Steiner \leftarrow \emptyset$;
- Set of Required nodes: $Req \leftarrow \emptyset$;
- For each Steiner node $u \in Steiner$, the algorithm will maintain a dynamic set R_u, containing the names of required nodes to which u is connected via lightweight edges, hence $|R_u|$ represents the lightweight degree of u;
- Variable representing a contracted node: $c \leftarrow \emptyset$;
- Variable which keeps the lightweight degree of the Steiner node when the instance has exactly one Steiner node: $Degree \leftarrow 0$;
- Flag set by the algorithm after the names and number of Steiner nodes in the instance is learned. $Learning \leftarrow$ **YES**. When the value of the flag becomes **NO**, then the algorithm has learned the precise number of Steiner nodes N_S and $Steiner$ contains the names of all Steiner nodes in the instance;
- $S, Deg(S), R_S$ will eventually point to the Steiner node of maximum lightweight degree, and the number of lightweight edges of that node, and the set of required nodes to which S is connected via lightweight edges, respectively. $Max, Deg(Max)$ are temporary variables which also point to the Steiner node of maximum lightweight degree.

  $Deg(Max) \leftarrow -\infty$; $Deg(S) \leftarrow -\infty$;

2. $while(I \neq \emptyset)$

(a) if ($t = 1$)

- Sort edges in the instance according to

$$\pi_1(v_i, t_i, v_j, t_j, w) = \begin{cases} 0 & \text{if } t_i = t_j = \mathbf{s} \\ 3 - w & \text{if } t_i = \mathbf{s} \text{ and } t_j = \mathbf{r} \\ 2 + w & \text{if } t_i = \mathbf{r} \text{ and } t_j = \mathbf{r} \end{cases}$$

- Let $e = (v_i, t_i, v_j, t_j, w)$ be the π_1-first edge
- Make a decision about e

  A. if ($\pi_1(e) = 0$) then $\mathit{reject}(e)$;

  $N_S \leftarrow 2$; $\mathit{Steiner} \leftarrow \mathit{Steiner} \cup \{v_i, v_j\}$

  B. if ($\pi_1(e) \in [1, 2]$) then $\mathit{reject}(e)$;

  $N_S \leftarrow 1$.

  C. if ($\pi_1(e) \in [3, 4]$) then $\mathit{accept}(e)$;

  $N_S \leftarrow 0$.

- $t \leftarrow t + 1$

(b) if ($t \geq 2$ and $N_S = 0$) then continue with Kruskal's algorithm:

- Sort edges according to

$$\pi_t(v_i, t_i, v_j, t_j, w) = w$$

- Let e be the π_t-first edge in this order
- $\mathit{accept}(e)$, if the solution remains a forest, otherwise $\mathit{reject}(e)$
- $t \leftarrow t + 1$

(c) if ($N_S = 1$) then

  i. if ($t = 2$) then

  - Sort edges according to

$$\pi_2(v_i, t_i, v_j, t_j, w) = \begin{cases} w & \text{if } t_i = \mathbf{r} \text{ and } t_j = \mathbf{r} \\ 2 + w & \text{if } t_i = \mathbf{r} \text{ and } t_j = \mathbf{s} \end{cases}$$

```
        • Let $e = (v_i, t_i, v_j, t_j, w)$ be $\pi_2$-first edge
        • Make decision about $e$
          A. if ($\pi_2(e) \in [1; 2]$) then $accept(e)$
                 $c \leftarrow \{r_i, r_j\}$ is 1-st contracted node (counts as a single
             required node)
                 $N_R \leftarrow 1$
          B. if ($\pi_2(e) \in [3, 4]$) then $reject(e)$ and halt
             The instance has one Steiner and one required node.
        • $t \leftarrow t + 1$
    ii. if ($t \geq 3$) then
        • Sort edges according to
```

$$\pi_t(v_i, t_i, v_j, t_j, w) = \begin{cases} 3 - w & \text{if (1) holds true} \\ 5 - w & \text{if } (t_i = \mathbf{r}) \wedge (t_j = \mathbf{s}) \ \wedge (w > 1.4) \\ 3 + w & \text{if (2) holds true} \\ 5 + w & \text{if (3) holds true} \\ 7 + w & \text{if } (t_i = \mathbf{r}) \wedge (t_j = \mathbf{r}) \end{cases}$$

$$(1) = \Big((t_i = \mathbf{r}) \wedge (t_j = \mathbf{r}) \wedge (v_i \in c) \wedge (v_j \notin \mathit{Req})\Big);$$
$$(2) = \Big((t_i = \mathbf{r}) \wedge (t_j = \mathbf{s}) \wedge (w \leq 1.4) \wedge \mathit{Degree} \geq 5\Big);$$
$$(3) = \Big((t_i = \mathbf{r}) \wedge (t_j = \mathbf{s}) \wedge (w \leq 1.4) \wedge \mathit{Degree} < 5\Big);$$

```
        • Let $e = (v_i, t_i, v_j, t_j, w)$ be $\pi_t$-first edge
        • Make a decision about $e$
          A. if ($\pi_t(e) \in [1, 2]$) then $reject(e)$
             Compute $G_1 = contract(G, c)$ and learn names and number of
             required nodes
                 $Req \leftarrow Req \cup \{v_j\}$;$N_R \leftarrow N_R + 1$; $Degree \leftarrow N_R$
```

B. if ($\pi_t(e) \in [3, 3.6)$) then $reject(e)$

e is heavyweight edge, update the lightweight degree of the Steiner node, with respect to the contract operation[1] performed in step 2(c)

$$Degree \leftarrow Degree - 1$$

C. if ($\pi_t(e) \in [4, 4.4]$) then $accept(e)$

e is lightweight edge and the Steiner node has lightweight degree ≥ 5

D. if ($\pi_t(e) \in [6, 7]$) then $reject(e)$

the lightweight degree of the Steiner node is less than 5

E. if ($\pi_t(e) \in [8, 9]$) then $accept(e)$ if it does not create a cycle, otherwise $reject(e)$

Here we use the Kruskal's algorithm to build a minimum cost tree spanning the required nodes.

- $t \leftarrow t + 1$

(d) if ($N_S \geq 2$ and $Learning$ = **YES**) then "count" the number of Steiner nodes in the instance

- Sort edges according to

$$\pi_t(v_i, t_i, v_j, t_j, w) = \begin{cases} 0 & \text{if } t_i = t_j = \mathbf{s} \\ w & \text{if } t_i = t_j = \mathbf{r} \\ 2 + w & \text{if } (t_i = \mathbf{s}) \wedge (t_j = \mathbf{r}) \end{cases}$$

- Let $e = (v_i, t_i, v_j, t_j, w)$ be π_t-first edge
- Make a decision about e

[1] If e is an edge between the Steiner node, call it S, and a required node not contracted by $c = \{r_i, r_j\}$ in step 2(c), then the algorithm will simply decrement the degree as noted. When e is an edge between S and one of the two required nodes previously contracted, then the algorithm does not immediately decrement $Degree$. Only when both edges (S, r_i) and (S, r_j) are heavyweight, then $Degree$ is decremented by 1, to be consistent with $G_1 = contract(G, c)$.

```
        A. if ($\pi_t(e) = 0$) then $reject(e)$
           $e$ is of type $(s, s)$, learn names and number of Steiner nodes
           – $Steiner \leftarrow Steiner \cup \{v_i, v_j\}$
           – $N_S \leftarrow N_S + 2$
        B. if $\pi_t(e) \in [3, 4]$ then $reject(e)$ and *halt*
           $e$ is of type $(r, s)$ hence the instance has only one required node
        C. if ($\pi_t(e) \in [1, 2]$) then $accept(e)$
           Contract the two required nodes into a single required node:
           – $c_1 \leftarrow \{r_i, r_j\}$
           – $N_R \leftarrow 1$
           – $Req \leftarrow Req \cup \{c_1\}$
           – $R_u \leftarrow \{c_1\}, \forall u \in Steiner$
           – $Learning \leftarrow \mathbf{NO}$
      • $t \leftarrow t + 1$
(e) if ($N_S \geq 2$ and $Learning = \mathbf{NO}$) then
      • Sort edges according to:
```

$$\pi_t(v_i, t_i, v_j, t_j, w) = \begin{cases} 3 - w & \text{if } (4) \text{ holds true} \\ 5 - w & \text{if } (5) \text{ holds true} \\ 7 - w & \text{if } (6) \text{ holds true} \\ 9 - w & \text{if } (7) \text{ holds true} \\ 11 - w & \text{if } (t_i = \mathbf{r}) \wedge (t_j = \mathbf{s}) \\ 10 + w & \text{if } (t_i = \mathbf{r}) \wedge (t_j = \mathbf{r}) \end{cases}$$

$$(4) = \Big((t_i = \mathbf{r}) \wedge (t_j = \mathbf{r}) \wedge (v_i \in c_1) \wedge (v_j \notin Req)\Big)$$
$$(5) = \Big((t_i = \mathbf{r}) \wedge (t_j = \mathbf{s}) \wedge (w > 1.4)\Big)$$
$$(6) = \Big((t_i = \mathbf{r}) \wedge (t_j = \mathbf{s}) \wedge (v_j = Max) \wedge (Deg(Max) \geq 5)\Big)$$
$$(7) = \Big((t_i = \mathbf{r}) \wedge (t_j = \mathbf{s}) \wedge (v_j = S) \wedge (Deg(S) > 0)\Big)$$

- Let $e = (v_i, t_i, v_j, t_j, w)$ be π_t-first edge.
- Make a decision about e:

A. if ($\pi_t(e) \in [1, 2]$) then $reject(e)$

Compute $G_1 = contract(G, c_1)$; learn names and number of required nodes

For each Steiner node u initialize the set R_u to be all required nodes

- $Req \leftarrow Req \cup \{v_j\}$;$N_R \leftarrow N_R + 1$
- $R_u \leftarrow Req, \forall u \in Steiner$

B. if ($\pi_t(e) \in [3, 4]$) then $reject(e)$

e is heavyweight edge; Update the sets R_v

- Decrease lightweight degree of v_j, respecting the contract operation performed in step 2(d) (See footnote 1 for details.)

  $R_{v_j} \leftarrow R_{v_j} \setminus \{v_j\}$
- Identify the Steiner node of highest lightweight degree at the moment

  $Max \leftarrow \text{argmax}_{u \in Steiner} |R_u|$

  $Deg(Max) \leftarrow |R_{Max}|$

C. if ($\pi_t(e) \in [5, 6]$) then $accept(e)$

Initialize S to be the Steiner node of the highest degree

- $S \leftarrow Max;\quad Deg(S) \leftarrow Deg(Max)$
- Uninitialize Max: $Max \leftarrow \emptyset$, $Deg(Max) \leftarrow -\infty$
- Update the lightweight degrees of the remaining Steiner nodes

  For all $u \in Steiner - \{S\}$ such that $|R_u \cap R_S| \geq 2$

```
    while($|R_u \cap R_S| \geq 2$)
        Pick $w \in (R_u \cap R_s)$
        $R_u \leftarrow R_u \setminus \{w\}$
```

```
            end-do
        D. if ($\pi_t(e) \in [7, 8]$) then $accept(e)$
           $e$ is incident to $S$; accept all edges between $S$ and nodes in $R_S$
           – Update the lightweight degree of $S$: $Deg(S) \leftarrow Deg(S) - 1$
           – If all the lightweight edges of $S$ have been accepted then find a
             Steiner node whose lightweight degree is maximum
             if ($Deg(S) = 0$) then
                 $Steiner \leftarrow Steiner - \{S\}$
                 $Max \leftarrow \mathrm{argmax}_{u \in Steiner}$; $Deg(Max) \leftarrow |R_{Max}|$
        E. if ($\pi_t(e) \in [9, 10]$) then $reject(r)$
           $e$ is a lightweight edge, but the lightweight degree of $v_j$ is less than
           $5$
        F. if ($\pi_t(e) \in [11, 12]$) then $accept(e)$ if $e$ does not form a cycle with
           the edges already accepted, otherwise $reject(e)$
           (Build a minimum cost spanning tree on required and contracted
           nodes.)
      • $t \leftarrow t + 1$
  $end(while)$
End Adaptive Contraction
```

Appendix B

Concentration Inequalities and Some Simple Bounds

B.1 Chernoff Style Bounds and Chebyshev's Inequality

Proposition 48. *[48]. Let X_i be i.i.d. Bernoulli variables with $\Pr[X_i = 1] = p, \forall i \in [1, n], n > 1$. Let $X = \sum_i X_i$ and $E(X) = \mu = np$ then*

1. *Lower tail. For any $\delta \in (0, 1]$:*

$$\Pr[X < (1-\delta)E(X)] < e^{-\frac{\mu\delta^2}{2}}.$$

2. *Upper tail. For any $\delta > 0$:*

$$\Pr[X > (1+\delta)E(X)] < \left(\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right)^{\mu}.$$

3. *Upper tail (simplified). For any $\delta > 2e - 1$:*

$$\Pr[X > (1+\delta)E(X)] < 2^{-(1+\delta)\mu}.$$

4. *Upper tail (simplified). For any $\delta \leq 2e - 1$:*

$$\Pr[X > (1+\delta)E(X)] < e^{-\frac{\mu\delta^2}{4}}.$$

Chebyshev's inequality states that if X is a random variable with finite variance σ^2 and let $\gamma > 0$ then

$$\Pr[|X - \mathrm{E}(X)| > \gamma\sigma] < \frac{1}{\gamma^2}.$$

B.2 Martingale and Submatringale Concentration Inequalities

The inequalities and definitions in this sections are from [46, 28, 4, 37, 36].

Definition 33. *A sequence of random variables $(Z_i)_{i=1}^{\infty}$ is a martingale with respect to another sequence $(Y_i)_{i=0}^{\infty}$ if $\mathrm{E}[Z_i | Y_0, \ldots Y_{i-1}] = Z_{i-1}$.*

Definition 34. *A sequence of random variables (Z_i) is a submartingale with respect to another sequence (Y_i) if $\mathrm{E}[Z_i | Y_0, \ldots Y_{i-1}] \geq Z_{i-1}$.*

Definition 35. *A sequence of random variables (Z_i) is a supermartingale with respect to another sequence (Y_i) if $\mathrm{E}[Z_i | Y_0, \ldots Y_{i-1}] \leq Z_{i-1}$.*

B.2.1 Properties of Conditional Expectation

Definition 36. *(Tower property of conditional expectation)*
Let $\mathcal{F}, \mathcal{G}$ be two σ-fields such that $\mathcal{G} \subseteq F$ and X is a random variable, measurable with respect to $\mathcal{F}, \mathcal{G}$, then the following two properties hold.

$$\mathrm{E}\left[\mathrm{E}[X|\mathcal{F}]|\mathcal{G}\right] = \mathrm{E}[X|\mathcal{G}] \tag{B.1}$$

$$\mathrm{E}\left[\mathrm{E}[X|\mathcal{G}]|\mathcal{F}\right] = \mathrm{E}[X|\mathcal{G}] \tag{B.2}$$

Equations (B.1) and (B.2) are called the tower property of conditional expectation because if you order the σ-fields in a tower from larger to smaller, then the smaller field always wins.

B.2.2 Doob Martingales

Let $(X)_{i=1}^{n}$ be a sequence of random variables. Let F be a random variable, which depends on $(X)_{i=0}^{n}$ (Usually F is a fuction of $X_0, X_1, \ldots, X_n$), such that $\mathrm{E}[F] < \infty$. A Doob martingale of F with respect to $(X)_{i=0}^{n}$ is defined as $Z_i = \mathrm{E}\left[F | X_0, \ldots, X_i\right]$.

$$\begin{aligned} Z_0 = &\ \mathrm{E}[F], \\ Z_1 = &\ \mathrm{E}[F|X_0, X_1], \\ \vdots & \\ Z_n = &\ \mathrm{E}[F|X_0, X_1, \ldots, X_n]. \end{aligned}$$

Obviously the sequence $Z_{i=0}^{n}$ is a martingale because of the tower property of conditional expectation (a.k.a. "the smaller field wins.").

1) $\mathrm{E}[Z_i] < \infty$.

2)
$$\begin{aligned} \mathrm{E}[Z_i|X_1, \ldots, X_{i-1}] &= \mathrm{E}[\mathrm{E}[Y|X_1, \ldots, X_i]|X_1, \ldots, X_{i-1}] \\ &= \mathrm{E}[Y|X_1, \ldots, X_{i-1}] = Z_{i-1}. \end{aligned}$$

The Doob martingale of F gives us better estimates of the random variable F, as more variables X_i become available. Initially we know only the expectation $\mathrm{E}[F]$, at the end when all X_i become available we know $\mathrm{E}[F|X_0, \ldots, X_n]$ whichi is the actual value (a deterministic quantity).

Suppose $(Z)_{i=0}^{n}$ is a martingale sequence with respect to the sequence $(X)_{i=1}^{n}$, then a martingale difference sequence is

$$Y_i = Z_i - Z_{i-1}.$$

If the martingale difference sequence is bounded then the Azuma-Hoeffding inequality gives the following concentration result.

Theorem 49 (Azuma). *Let (Z_i) be a martingale with respect to a sequence $(X)_{i=0}^{n}$ (or itself) and let Y_i be a martingale difference sequence defined as $Y_i = Z_i - Z_{i-1}$. If*

$|Y_i| \leq c_i, \forall i$ *then*

$$\left.\begin{array}{l}\Pr[Z_n \geq Z_0 + \lambda] \\ \Pr[Z_n \leq Z_0 - \lambda]\end{array}\right\} \leq \exp\left(-\frac{\lambda^2}{2\sum_{i=1}^n c_i^2}\right)$$

We show the proof of the upper tail.

Proof. [36] We use the standard technique by introducing additional variable t whose value we later optimize and exponentiate the two sides and apply the Markov's inequality.

$$\Pr[X_k - X_0 \geq \lambda] = \Pr[t(X_k - X_0) \geq t\lambda] = \\ \Pr[e^{t(X_k - X_0)} \geq e^{t\lambda}] \leq \frac{\mathrm{E}\left[e^{t(X_k - X_0)}\right]}{e^{t\lambda}}$$

Define $Y_i = X_i - X_{i-1}$, hence $|Y_i| \leq c_i$. Furthermore, since (X_i) is a martingale we have

$$\begin{aligned}\mathrm{E}[Y_i|X_0,\ldots,X_{i-1}] \quad &= \mathrm{E}[X_i - X_{i-1}|X_0,\ldots,X_{i-1}] \\ &= \mathrm{E}[X_i|X_0,\ldots,X_{i-1}] - \mathrm{E}[X_{i-1}|X_0,\ldots,X_{i-1}] \\ &= \mathrm{E}[X_i|X_0,\ldots,X_{i-1}] - X_{i-1} = X_{i-1} - X_{i-1} = 0.\end{aligned}$$

We want to bound $\mathrm{E}[e^{tY_i}|X_0,\ldots,X_{i-1}]$ next. Because e^x is a convex[1] function then

$$\begin{aligned}e^{tY_i} \quad &= e^{t(-c_i(\frac{1-Y_i/c_i}{2}))+c_i(\frac{1+Y_i/c_i}{2})} \\ &\leq (\frac{1-Y_i/c_i}{2}))e^{-tc_i} + (\frac{1+Y_i/c_i}{2})e^{tc_i} \\ &= \frac{e^{-tc_i}+e^{tc_i}}{2} + \frac{Y_i}{2}(e^{tc_i} - e^{-tc_i})\end{aligned}$$

[1]If $f(x)$ is convex then $f(\alpha x_x + (1-\alpha)x_2) \leq \alpha f(x) + (1-\alpha)f(x_2)$, for any $\alpha \in (0,1)$.

$$\begin{aligned}
\mathrm{E}[e^{tY_i}|X_0,\dots X_{i-1}] \quad & \leq \mathrm{E}\left[\frac{e^{-tc_i}+e^{tc_i}}{2}+\frac{Y_i}{2}(e^{tc_i}-e^{-tc_i})|X_0,\dots X_{i-1}\right] \\
& = \mathrm{E}[\left[\frac{e^{-tc_i}+e^{tc_i}}{2} \mid X_0,\dots X_{i-1}\right] \\
& + \mathrm{E}[\left[\frac{Y_i}{2}(e^{tc_i}-e^{-tc_i}) \mid X_0,\dots X_{i-1}\right] \\
& = \frac{e^{-tc_i}+e^{tc_i}}{2}+\frac{(e^{cc_i}+e^{-tc_i})}{2}\cdot \mathrm{E}[Y_i \mid X_0,\dots X_{i-1}] \\
& = \frac{e^{-tc_i}+e^{tc_i}}{2}+\frac{(e^{tc_i}+e^{-tc_i})}{2}\cdot 0 \\
& \leq \frac{e^{-tc_i}+e^{tc_i}}{2}.
\end{aligned}$$

Now by the Maclaurin expansion of e^x we have that $e^x = \sum_{i=1}^{\infty} \frac{x^i}{i!}$, hence

$$\begin{aligned}
e^x + e^{-x} \quad & = 1 + x + \frac{x^2}{2}\cdots + 1 - x + \frac{x^2}{2} + \cdots \leq 1 + x + \frac{x^2}{2} + 1 - x + \frac{x^2}{2} \\
& = 2(1+\frac{x^2}{2}) \leq 2e^{\frac{x^2}{2}}.
\end{aligned}$$

Hence

$$\mathrm{E}[e^{tY_i} \mid X_0,\cdots,X_{i-1}] \leq \frac{1}{2}(e^{-tc_i}+e^{tc_i}) \leq e^{\frac{(tc_i)^2}{2}}. \tag{B.3}$$

We are ready to conclude the proof.

$$\begin{aligned}
\mathrm{E}\left[e^{t(X_k-X_0)}\right] = \quad & \mathrm{E}\left[\Pi_{i=1}^{k} e^{tY_i}\right] \\
& = \mathrm{E}\left[e^{tY_i} \mid X_0,\dots,X_{i-1}\right]\cdot \mathrm{E}\left[\Pi_{i=1}^{k-1} e^{tY_i}\right] \\
& = \mathrm{E}\left[e^{tY_i} \mid X_0,\dots,X_{i-1}\right]\cdot \mathrm{E}\left[e^{tY_i-1} \mid X_0,\dots,X_{i-2}\right]\cdots \mathrm{E}\left[e^{tY_1}\right] \\
& = e^{((tc_1)^2+\cdots+(tc_k)^2)/2}.
\end{aligned}$$

Finally

$$\Pr[X_k - X_0 > \lambda] \leq \frac{\mathrm{E}[e^{t(X_k-X_0)}]}{e^{t\lambda}} = e^{\frac{t^2}{2}(\sum_{i=1}^{k} c_i^2)-t\lambda} = e^{-\frac{\lambda^2}{(2\sum_{i=1}^{k} c_i^2)}}.$$

For $t = \frac{\lambda}{\sum c_i^2}$. □

B.2.3 Submartingales

The following concentration result about bounded difference submartingale is from [46].

Theorem 50. *[46] Let (Z_i) be a submartingale with $Z_0 = 0$ and $X_i = Z_i - Z_{i-1} \in [0, 1]$, for all i. If $E[X_i \mid X_1, \ldots, X_{i-1}] \geq \mu_i$ and let $\mu = \sum_i \mu_i$ then*

$$\Pr[Z_n < (1-\delta)\mu] < e^{-\frac{\delta^2 \mu}{2}},$$

for all $\delta \in (0, 1), x > 1$.

Proof. The proof is similar as above.

$$\Pr[Z_n \leq (1-\delta)\mu] = \Pr[e^{-tZ_n} \geq e^{-t(1-\delta)\mu}] \leq \frac{\mathrm{E}\left[e^{-tZ_n}\right]}{e^{-t(1-\delta)\mu}}$$

Note that we know the conditional expectation of X_i's but not of e^{tX_i}, which we need. By the convexity of e^y, for any $y \in [0, 1]$ we have that $e^{\alpha y} = e^{(1-y)0+y\alpha} \leq (1-y)e^0 + ye^\alpha = 1 - y + ye^\alpha$. Similarly for any random variable $X \in [0, 1]$ we can lower bound

$$\mathrm{E}[e^{-tX}] \leq \mathrm{E}[(1-X) + Xe^{-t}] = 1 + \mathrm{E}[X](e^{-t} - 1).$$

Now we continue with $\mathrm{E}[e^{-tZ_n}]$.

$$\begin{aligned}
\mathrm{E}\left[e^{-tZ_n}\right] &= \mathrm{E}\left[e^{-t\sum_{k=1}^n X_k}\right] = \mathrm{E}\left[\Pi_{k=1}^n e^{-tX_k}\right] \\
&= \mathrm{E}\left[\left(\Pi_{k=1}^{n-1} e^{-tX_k}\right)\right] \cdot \mathrm{E}\left[e^{-tX_n} \mid X_1 \ldots X_{n-1}\right] \\
&\leq \mathrm{E}\left[\Pi_{k=1}^{n-1} e^{-tX_k}\right] \cdot \mathrm{E}\left[1 + (1 + e^{-t})X_n \mid X_1 \ldots X_{n-1}\right] \\
&= \mathrm{E}\left[\Pi_{k=1}^{n-1} e^{-tX_k}\right] \cdot \big(1 + (e^{-t} - 1)\big)\, \mathrm{E}\left[X_n \mid X_1 \ldots X_{n-1}\right] \\
&= \mathrm{E}\left[\Pi_{k=1}^{n-1} e^{-tX_k}\right] \cdot \big(1 + (e^{-t} - 1)\mathrm{E}\left[X_n \mid X_1 \ldots X_{n-1}\right]\big) \\
&= \mathrm{E}\left[\Pi_{k=1}^{n-1} e^{-tX_k}\right] \cdot \big(1 + (e^{-t} - 1)\mu_n\big) \\
&\vdots \\
&= \Pi_{k=1}^n \left(1 + (e^{-t} - 1)\mu_k\right) < \Pi_{k=1}^n \left(e^{(e^{-t}-1)\mu_k}\right) = e^{(e^{-t}-1)\sum_{k=1}^n \mu_k} \\
&= e^{(e^{-t}-1)\mu}
\end{aligned}$$

$$\Pr[Z_n \leq (1-\delta)\mu] < \frac{e^{(e^{-t}-1)\mu}}{e^{-t(1-\delta)\mu}}$$

Choose $t = -\ln(1-\delta)$

$$\Pr[Z_n \leq (1-\delta)\mu] < \left(\frac{e^{-\delta}}{(1-\delta)^{(1-\delta)}}\right)^{\mu}$$

The Maclaurin expansion of $\ln(1+x)$ gives us $\ln(1-\delta) \geq -\delta + \frac{\delta^2}{2}$, and because $(1-\delta) < 1$ then $(1-\delta)^{(1-\delta)} \geq e^{-\delta+\frac{\delta^2}{2}}$. Now we conclude the proof:

$$\Pr[Z_n \leq (1-\delta)\mu] < e^{-\frac{\delta^2\mu}{2}}.$$

□

Bibliography

[1] Alekhnovich, M., Borodin, A., Buresh-Oppenheim, J., Impagliazzo, R., Magen, A., and Pitassi, T., 2005: Toward a model for backtracking and dynamic programming. In *IEEE Conference on Computational Complexity*, 308–322.

[2] Alekhnovich, M., Hirsch, E. A., and Itsykson, D., 2004: Exponential lower bounds for the running time of DPLL algorithms on satisfiable formulas. In *ICALP 2004, Turku, Finland.*, 84–96.

[3] Alekhnovich, M., and Razborov, A. A., 2001: Lower bounds for polynomial calculus: Non-binomial case. In *FOCS*, 190–199.

[4] Alon, N., and Spencer, J., 1992: *The Probabilistic Method*. John Wiley. ISBN 0-471-53588-5.

[5] Angelopoulos, S., 2003: Randomized priority algorithms. In *WAOA, Budapest, Hungary, September 16-18, 2003*, 27–40.

[6] Angelopoulos, S., 2004: Order-preserving transformations and greedy-like algorithms. In *WAOA*, 197–210.

[7] Angelopoulos, S., and Borodin, A., 2004: On the power of priority algorithms for facility location and set cover. *Algorithmica*, **40(4)**, pp.271–291.

[8] Arora, S., Bollobás, B., and Lovász, L., 2002: Proving integrality gaps without knowing the linear program. In *FOCS*, 313–322.

[9] Bern, M. W., and Plassmann, P. E., 1989: The Steiner problem with edge lengths 1 and 2. *Inf. Process. Lett.*, **32**(4), 171–176.

[10] Bollobas, B., 2001: *Random Graphs*. Cambridge University Press. ISBN 0-521-80920-5.

[11] Borodin, A., Boyar, J., and Larsen, K. S., 2004: Priority algorithms for graph optimization problems. In *WAOA*, 126–139.

[12] Borodin, A., Cashman, D., and Magen, A., 2005: How well can primal-dual and local-ratio algorithms perform?. In *ICALP*, 943–955.

[13] Borodin, A., and El-Yaniv, R., 1998: *Online Computation and Competitive Analysis*. Cambridge University Press.

[14] Borodin, A., Nielsen, M. N., and Rackoff, C., 2003: (Incremental) Priority Algorithms. *Algorithmica*, **37(4)**, pp.295–326.

[15] Brightwell, G., and Winkler, P., 1991: Counting linear extensions is #p-complete. In *STOC, New Orleans, Louisiana, USA*, 175–181. ACM.

[16] Buresh-Oppenheim, J., Davis, S., and Impagliazzo, R., 2008: A general model for backtracking and dynamic programming algorithms. *manuscript*.

[17] Buresh-Oppenheim, J., Davis, S., and Impagliazzo, R., 2008: A stronger model for dynamic programming algorithms. *manuscript*.

[18] Clementi, A. E. F., and Trevisan, L., 1999: Improved non-approximability results for minimum vertex cover with density constraints. *Theor. Comput. Sci.*, **225**(1-2), 113–128.

[19] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C., 2001: *Introduction to Algorithms, Second Edition*. The MIT Press and McGraw-Hill Book Company. ISBN 0-262-03293-7, 0-07-013151-1.

[20] Davis, S., and Impagliazzo, R., 2004: Models of greedy algorithms for graph problems. In *SODA, New Orleans, Louisiana, USA*, 381–390. SIAM.

[21] Davis, S., and Impagliazzo, R., 2007: Models of greedy algorithms for graph problems. *Algorithmica*. doi:0178-4617,Springer,November,2007.

[22] Dinur, I., and Safra, S., 2005: On the hardness of approximating minimum vertex-cover. *Annals of Mathematics*, **162**(1), 439–485.

[23] Fiat, A., and Woeginger, G., Septemer 1998: *Online Algorithms: The State of the Art*. Springer.

[24] Frieze, A., and Yukich, J., 2002: *Probabilistic analysis of the Traveling Salesman problem, in: G. Gutin and A.P. Punnen (Eds.), The Traveling Salesman Problem and its Variations*. Kluwer Acadeic Publisher, Dordrecht. ISBN 0-521-80920-5.

[25] Halldórsson, M. M., and Yoshihara, K., 1995: Greedy approximations of independent sets in low degree graphs. In *ISAAC*, 152–161.

[26] Halperin, E., 2002: Improved approximation algorithms for the vertex cover problem in graphs and hypergraphs. *SIAM J. Comput.*, **31**(5), 1608–1623.

[27] Håstad, J., 2001: Some optimal inapproximability results. *J. ACM*, **48**(4), 798–859.

[28] Hoeffding, W., 1963: Probability inequalities for sums of boundded random variables. *J. of Amer. Statist. Assoc.*, **58**(301), 13–30.

[29] Ibarra, O. H., Kim, S. M., and Rosier, L. E., 1985: Some characterizations of multihead finite automata. *Information and Control*, **67**(1-3), 114–125.

[30] Johnson, D. S., 1974: Approximation algorithms for combinatorial problems. *J. Comput. Syst. Sci.*, **9**(3), 256–278.

[31] Kleinberg, J., and Tardos, E., 2005: *Algorithm Design*. Addisn Wesley.

[32] Knuth, D. E., 1997: *Stable Marriage and its Relation to Other Combinatorial Problems*. American Mathematical Society.

[33] Kou, L. T., Markowsky, G., and Berman, L., 1981: A fast algorithm for steiner trees. *Acta Inf.*, **15**, 141–145.

[34] Matousek, J., and Vondrak, J., 2004: *The Probabilistic Method, Lecture Notes*. Manuscript.

[35] McDiarmid, C. J. H., 1983: General first passage percolation. *Advances in Applied Probability*, **15**(1), 149–161.

[36] Mitzenmacher, M., and Upfal, E., 2005: *Probability and Computing.* Cambridge University Press. ISBN 0-521-83540-2.

[37] Motwani, R., and Raghavan, P., 1995: *Randomized Algorithms.* Cambridge University Press. ISBN 0-521-47465-5.

[38] Neapolitan, R. E., and Naimipour, K., 1997: *Foundations of Algorithms*. Jones & Bartlett Publishing.

[39] Papakonstantinou, P. A., 2006: Hierarchies for classes of priority algorithms for job scheduling. *Theor. Comput. Sci.*, **352**(1-3), 181–189.

[40] Paturi, R., Pudlák, P., and Zane, F., 1999: Satisfiability coding lemma. *Chicago J. Theor. Comput. Sci.*

[41] Plesnìk, J., 1981: A bound for Steiner tree problem in graphs. *Math. Slovaca*, **31**, 155–163.

[42] Rayward-Smith, V. J., 1983: The computation of nearly minimal Steiner trees in graphs. *Internat J. Math. Educ. Sci. Tech.*, **14**, 15–23.

[43] Robins, G., and Zelikovsky, A., 2005: Tighter bounds for graph steiner tree approximation. *SIAM J. Discrete Math.*, **19**(1), 122–134.

[44] Robson, J. M., 1986: Algorithms for maximum independent sets. *J. Algorithms*, **7**(3), 425–440.

[45] Ross, S., 1976: *A First Cource in Probability*. Macmillan Publishing Co., Inc.

[46] Russell, A., Saks, M. E., and Zuckerman, D., 2002: Lower bounds for leader election and collective coin-flipping in the perfect information model. *SIAM J. Comput.*, **31**(6), 1645–1662.

[47] Sahni, S., 1976: Algorithms for scheduling independent tasks. *J. ACM*, **23**(1), 116–127.

[48] Vazirani, V. V., 2001: *Approximation Algorithms*. Springer.

[49] Waxman, B. M., and Imase, M., 1988: Worst-case performance of Rayward-Smith's Steiner tree heuristic. *Inf. Process. Lett.*, **29**(6), 283–287.

[50] Winter, P., 1987: Steiner problem in networks: a survey. *Networks*, **17**, 129–167.

[51] Woeginger, G. J., 1999: When does a dynamic prgramming formulation guarantee the existance of fptas? In *SODA*, 820–829.

[52] Woeginger, G. J., 2001: Exact algorithms for np-hard problems: A survey. In *Combinatorial Optimization*, 185–208.