Electronic Colloquium on Computational Complexity, Report No. 120 (2005)

Models of Greedy Algorithms for Graph Problems

Sashka Davis*
University of California, San Diego
sdavis@cs.ucsd.edu

Russell Impagliazzo†
University of California, San Diego
russell@cs.ucsd.edu

October 13, 2005

Abstract

Borodin, Nielsen, and Rackoff ([BNR02]) gave a model of greedy-like algorithms for scheduling problems and [AB02] extended their work to facility location and set cover problems. We generalize their notion to include other optimization problems, and apply the generalized framework to graph problems. Our goal is to define an abstract model that captures the intrinsic power and limitations of greedy algorithms for various graph optimization problems. We prove bounds on the approximation ratio achievable by such algorithms for basic graph problems such as shortest path, vertex cover, and others. Shortest path is an example of a problem where no algorithm in the FIXED priority model can achieve any approximation ratio (even one dependent on the graph size), but for which the well-known Dijkstra's algorithm shows that an ADAPTIVE priority algorithm can be optimal. We also prove that the approximation ratio for vertex cover achievable by ADAPTIVE priority algorithms is exactly 2. Here, a new lower bound matches the known upper bounds ([Joh74]).

1 Introduction

There is a huge variety of known algorithms for a huge variety of computational problems. However, a surprisingly large fraction of the known efficient algorithms fit into relatively few basic design paradigms: e.g., divide-and-conquer, dynamic programming, greedy algorithms, linear programming relaxation, and hill-climbing. While algorithm designers have long had a good intuitive feel for these paradigms, there is still little research formalizing these paradigms and so little is known about their relative computational power. For example, intuitively, greedy algorithms are faster but less powerful than dynamic programming algorithms; how could we make such a statement formal?

To begin to address such questions, Borodin, Nielson and Rackoff recently introduced a formal framework for the greedy algorithm paradigm ([BNR02]). Their framework, which they called *priority algorithms*, was originally given only for scheduling problems. However, Angelopoulos and Borodin [AB02] extended the priority algorithm framework to facility location and set cover problems. In this paper, we present a more abstract definition of a priority algorithm that can be applied to a variety of problem domains. In particular, we apply this model to formalize greedy algorithms for graph problems.

Even using the standard techniques only, the space of possible algorithms for a problem is large. This profusion of competing algorithmic approaches has led to a growing interest in formalizing

*Research supported by NSF grant CCR-0098197.

†Research supported by NSF grant CCR-0098197. Some work done while at the Institute for Advanced Study, supported by the State of New Jersey.

ISSN 1433-8092

and analyzing algorithmic methods, rather than individual algorithms. For example, much of the interest in recent proof complexity has been in studying the relative strength of different approaches to satisfiability algorithms. Extending traditional integrality gap approaches to proving lower bounds for linear programming relaxation algorithms based on specific relaxations, Arora, et. al. [ABL74], have shown lower bounds on very general forms of relaxations. While not as technically difficult as the above work, our work contributes to a more general understanding of the relative power of algorithmic methods. We believe that the priority models described here can be taken as a starting point in formally defining and analyzing more powerful computational paradigms, such as backtracking and dynamic programming.

1.1 Motivation

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 plausible greedy algorithms for a problem, and one's first choice is often not the best algorithm.

There are also distinctions that can be made between greedy algorithms. For example, 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.

The priority model allows one to formally address all of these uses of greedy algorithms and issues in greedy algorithm design. 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 and the separation of FIXED and ADAPTIVE priority algorithms in Section 3.1).
2. Show that sometimes greedy approximation algorithms need to be counter-intuitive. This can be done by defining a sub-model of "sensible" priority algorithms, and showing that this subclass is weaker than the general priority model. (See the definition of memoryless algorithms in Section 4).
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.

1.2 Related Work

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. For this reason, the techniques used to prove bounds for priority algorithms often are borrowed from the extensive literature on on-line algorithms (See [BEY98] for a good overview).

However, where an on-line algorithm sees the parts of its input 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. [BNR02] considered two variants: *FIXED priority* algorithms where this order is independent of the instance and constant throughout the algorithm, and *ADAPTIVE priority* where the algorithm can change the order of future parts based on the part of the instance that it has seen. 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, and *MEMORYLESS* adaptive priority algorithms, which must base decisions only on the set of previously accepted data items.

[BNR02] 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, and minimum makespan). They showed a separation between the class of ADAPTIVE priority algorithms and FIXED priority algorithms, by proving lower bound of 3 on the performance of FIXED priority algorithms on the 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 [BNR02] 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 [BNR02] 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 performance of ADAPTIVE priority algorithms, 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 proved that no ADAPTIVE priority algorithm can achieve an approximation ratio better than $\ln n - \ln\ln n + \Theta(1)$. This bound is tight because the greedy set cover heuristic, classified as an ADAPTIVE priority algorithm, achieves the bound. [AB02] also considered the unrestricted facility location problem, and proved a tight bound of $\Omega(\log n)$ on the performance of any ADAPTIVE priority algorithm, which is matched by the known greedy heuristic for the problem. For the uniform metric facility location problem, they were able to show a tight bound of 3 on the approximation ratio achieved by fixed priority algorithms.

Boyar and Larsen ([BL03]) 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 where a data item encodes only the degree of the vertex, excluding the names of the vertices adjacent to it, and proved lower bounds matching known upper bound results in those models.

1.3 Our Results

In this paper, we extend their model to combinatorial optimization problems of other domains. In particular, we look at models for graph problems, and evaluate the performance of priority algorithms for some classical graph problems. Our main contributions are:

Abstract priority model We present an abstract definition of priority algorithm that applies

to a variety of problem domains. Previous work defined a new model for each domain, e.g, scheduling algorithms ([BNR02]) or facility location problems ([AB02]). We define two instantiations of our abstract model for graph problems, the node model and the edge model.

Characterization of models We show how to characterize the power of three of the models (FIXED, ADAPTIVE, and MEMORYLESS) in terms of combinatorial games between a Solver and an Adversary. This characterization holds for the abstract definitions of priority algorithms, and so is problem-independent.

Shortest Paths We show a strong separation between FIXED and ADAPTIVE priority algorithms. We consider the problem of finding the shortest path in graph G from node s to t. Dijkstra's algorithm is seen to be an ADAPTIVE priority algorithm in this model, for the case when edge weights are positive. In the case of positive edge weights, we show that no FIXED priority algorithm can achieve any approximation ratio, even a ratio that is a function of the graph size. Secondly, for graphs with negative edge weights (but no negative cycles), we show that no ADAPTIVE priority algorithm can achieve any approximation ratio.

Steiner Trees We consider the Steiner Tree problem for metric spaces. Here, a standard FIXED priority algorithm achieves an approximation ratio of 2. We show that no ADAPTIVE priority algorithm can achieve an approximation ratio better than 1.18, even for the special case when every positive distance is between 1 and 2. We give an improved algorithm for this special case, in the ADAPTIVE priority model.

Weighted Vertex Cover We considered the Weighted Vertex Cover problem in the node model. For the weighted vertex cover problem, the standard 2-approximation algorithm fits into the ADAPTIVE priority model. Moreover, we show that no ADAPTIVE priority algorithm can achieve a better approximation ratio. Thus, the known algorithm is optimal within the class of priority algorithms.

Independent Set We consider the independent set problem for graphs of small degree, in the node model. For degree-3 graphs, the standard greedy algorithm achieves an approximation ratio of $\frac{5}{3}$, [HY95]. This algorithm fits in the ADAPTIVE priority model. We show that no ADAPTIVE priority algorithm can achieve an approximation ratio better than $\frac{3}{2}$.

Memoryless Algorithms We define a formal model of memoryless algorithms and prove a separation between the class of ADAPTIVE priority algorithms (with memory) and memoryless adaptive priority algorithms.

1.4 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 ([NN97]) 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 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:

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.
5. Never reverses a choice once made (i.e., decisions are irrevocable).
6. 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. Many of the lower bound techniques here, in [BNR02], and in [AB02] 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 algorithms, respectively. Dijkstra's single source shortest path algorithm also can be seen to fit the ADAPTIVE priority model. The known greedy approximation [Joh74] for the Weighted Vertex Cover problem (WVC) can be classified as an ADAPTIVE priority algorithm. The greedy approximation for the independent set problem [HY95] also fits our model. In [AB02] it was shown that 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. 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 metaphorically "greedy" but do not fit the above description[1].

[1] 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; however, this will be motivation to try to extend the model in future work.

1.5 Priority models

To make the above definition 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 exact 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 denotes the type of a data item; thus an instance is a set of items of type Γ, $I \subseteq \Gamma^2$. 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 $\Sigma = \{1, \ldots, k\}$, and Γ should correspond to the *information available about a node* when the algorithm has to color it. This 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), and a list of neighbors. 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 (as appropriate to the problem), 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, [BNR02] 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, $\Gamma = \{(a, d, t, w) | a < d, t \geq d - a, w \geq 0\}$.

In fact, we can put pretty much any search or optimization problem in the above framework. Let $D(I)$ determine ***decision points*** for an instance I so that solutions can be described as

[2] 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.

[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 $y \in AdjList(x)$.

arrays indexed by $D(I)$, $S \in \Sigma^{D(I)}$. Assume we give the algorithm access to the information $\mathit{LocalInfo}(d, I)$ when making decision $d \in D(I)$. 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 *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 [BNR02], we distinguish between algorithms that order their data points at the start, and those that reorder them at each iteration. A FIXED priority algorithm orders the decisions at the start, and proceeds according to that order. The format for a FIXED priority algorithm is as follows:

FIXED PRIORITY ALGORITHM
Input: instance $I \subseteq \Gamma$, $I = \{\gamma_1, \ldots, \gamma_d\}$.
Output: solution $S = \{(\gamma_i, \sigma_i) | i = 1, \ldots, d\}$.
- Determine a criterion for ordering the decisions, based on the data items[4]: $\pi : \Gamma \to \mathbb{R}^+ \cup \{\infty\}$
- Order I according to $\pi(\gamma_i)$, from smallest to largest

Repeat

- Go through the data items γ_i in order
- In step t, observe the t'th data item according to π, let that be γ_{i_t}. 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 π). Update the partial solution: $S \leftarrow S \cup \{(\gamma_{i_t}, \sigma_{i_t})\}$
- Go on to the next data item

Until (decisions are made for all data items)
Output $S = \{(\gamma_i, \sigma_i) | 1 \leq i \leq d\}$.

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

ADAPTIVE PRIORITY ALGORITHM
Input: instance $I \subseteq \Gamma$, $I = \{\gamma_1, \ldots, \gamma_d\}$.
Output: solution vector $S = \{(\gamma_i, \sigma_i) | 1 \leq i \leq d\}$
- Initialize the set of unseen data points U to I, an empty partial instance PI, an empty partial solution S, and a counter t to 1.

Repeat

- Based only on the previously observed data items PI, determine an ordering function $\pi_t : \Gamma \to \mathbb{R}^+ \cup \{\infty\}$
- Order $\gamma \in U$ according to $\pi_t(\gamma)$
- Observe the first unseen data item $\gamma_t \in U$, and add it to the partial instance, $PI \leftarrow PI \cup \{\gamma_t\}$.
- Based only on PI, make an irrevocable decision σ_t and add (γ_t, σ_t) to the partial solution S
- Remove the processed data point γ_t from U, increment t

[4]We could instead use a more general notion, where π is a total ordering of Γ. 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.

Until (decisions are made for all data items, $U = \emptyset$)
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.

[BNR02] 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.*" 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. An interesting problem for future research is to define an analogous notion of GREEDY priority algorithm for graph problems.

We formalize the notion of MEMORYLESS algorithms in Section 4 and show a separation between the class of memoryless algorithms and adaptive priority algorithms.

2 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.

Let Π be a maximization problem, with objective function μ, and 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 ratio $\rho \geq 1$ between two players a Solver and an Adversary is as follows:

1. Initialize an empty partial instance PI and 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 $PI \notin T$, S is not a valid solution, or if PS is a valid solution and $\rho \geq \frac{\mu(PS)}{\mu(S)}$ for maximization problem, or $\rho \geq \frac{\mu(S)}{\mu(PS)}$ for minimization problem.

Lemma 1 *There is a winning strategy for the Solver in the above 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.*

Proof:

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. The priority algorithm has to define an ordering on data items, without looking at the instance, and then has to make a decision for the data item with highest priority value.

Assume there is an ADAPTIVE priority algorithm achieving approximation ratio 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 of 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 is a solution PS. Since the algorithm achieves an approximation ratio ρ, $\frac{\mu(PS)}{\mu(S)} \geq \rho$.

For the converse, assume there is a strategy for the Solver that achieves payoff ρ. We describe an ADAPTIVE priority algorithm that ensures approximation ratio ρ.

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 is guaranteed payoff at least ρ, $\frac{\mu(S)}{\mu(S')} \geq \rho$, so our algorithm achieves ratio 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 $\Gamma_1 \subseteq \Gamma$.

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 above game if and only if there is a FIXED priority algorithm that achieves an approximation ratio ρ on every instance of Π in T.*

3 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 Section 4 resemble derivation of integrality gaps for given LP formulation, in that easy instances are used to establish bounds on the approximation ratio (See [Vaz01] for examples). Our results are used to evaluate priority algorithms as a model for greedy heuristics, rather than to establish hardness of approximation results for the particular problem.

3.1 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 1 (ShortPath 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 ShortPath problem in the edge model. 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 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 ShortPath 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.

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 ShortPath problem with any approximation ratio ρ.*

Proof sketch: 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 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 itself be simple. 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

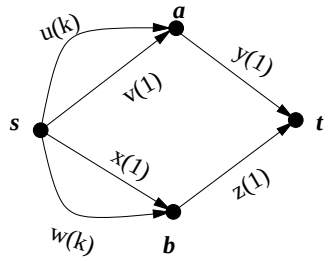

Figure 1: Adversary selects $\Gamma_1 = \{x, y, z, u, v, w\}$.

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

□

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 algorithms, 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 ShortPath 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. Although the set Γ_1 defines a with a negative weight cycle, the final instances T are subgraphs of the graph shown in Figure 2, with a path from s to t, and no negative weight cycle. The set of decisions options is $\Sigma = \{accepted, rejected\}$. We will refer to the edge from v to u of weight $-k$ as $d(-k)$, for short. The Adversary and the Solver play of the combinatorial game defined in Section 2. The Adversary observes the first data item selected and the decision committed by the Solver and uses the following strategy.

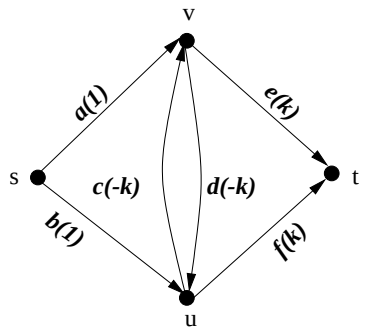

Figure 2: The set of edges initially selected by the Adversary $\Gamma_1 = \{a, b, c, d, e, f\}$.

1. The first data item chosen by the Solver is $a(1)$.
   - If the Solver decides to accept a, then then the Adversary presents the instance $I = \{a(1), b(1), c(-k), e(k), f(k)\}$ 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 is awarded an approximation ratio $\rho = \frac{\mu(S_{Sol})}{\mu(S_{Adv})} = \frac{k+1}{1}$.
   - If the Solver decides to reject a, then the Adversary presents the instance $I = \{a(1), e(k)\}$ and a solution $S_{Adv} = \{a, e\}$, while the Solver would fail to construct a path.
2. The first data item chosen by the Solver is $c(-k)$.
   - If the Solver decides to accept c, then the Adversary presents the instance $I = \{a(1), c(-k), e(k)\}$ and a solution $S_{Adv} = \{a, e\}$. The Solver would fail to construct a tree.
   - If the Solver decides to reject c, then the Adversary presents the instance $I = \{a(1), b(1), c(-k), e(k)\}$ and a $S_{Adv} = \{b, c, e\}$ of cost 1. The only solution left for the Solver is $S_{Sol} = \{a, e\}$ of cost $k + 1$ and is awarded an approximation ratio $\rho = \frac{\mu(S_{Sol})}{\mu(S_{Adv})} = \frac{k+1}{1}$.
3. The first data item chosen by the Solver is $e(1)$.
   - If the Solver decides to accept e, then the Adversary presents the instance $I = \{b(1), e(1), f(1)\}$ and a solution $S_{Adv} = \{a, f\}$, while the Solver would fail to construct a tree.
   - If the Solver decides to reject e, then the Adversary presents the instance $I = \{a(1), e(1)\}$ and a solution $S_{Adv} = \{a, e\}$, while the Solver failed to construct a tree.

The cases when the Solver considers edges $b(1), d(-k)$, or $f(k)$ first, have the same analysis as the cases already discussed and will be omitted. The Adversary can set k arbitrarily large, thus he can force any approximation ratio. ▌

This result shows a separation between priority algorithms and dynamic programming algorithms for shortest path problems; a similar separation was shown by [BNR02] for interval scheduling on a single machine with arbitrary profits.

3.2 The weighted vertex cover problem

We examine the performance of ADAPTIVE priority algorithms on the Weighted Vertex Cover problem (WVC) next. WVC is **NP**-hard problem and no polynomial time algorithm can solve it

exactly, or even with approximation ratio $1+\epsilon$, for some $\epsilon > 0$ unless $P = NP$, [Vaz01]. The well known 2-approximation algorithm [Joh74], fits our ADAPTIVE priority model, and we show that no ADAPTIVE priority algorithm can achieve an approximation ratio better than 2.

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 Section 2. 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 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 possible for all nodes on the same side to 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 $n^2 + n - 1$. 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 containing all of the other side B, giving a total weight of $2n-1$. The Adversary takes 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 ([BL03]) consider the unweighted version and prove a lower bound of $\frac{4}{3}$ for any priority algorithm.

3.3 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. Each edge $e \in E$ has a positive weight $w(e)$. The problem is to find a minimum cost tree, spanning the required vertices which may contain any number of Steiner nodes. We are interested in the metric version, where the edge weights obey the triangle inequality. The standard 2-approximation algorithm for the Steiner tree problem discovered independently by [KMB81] and [Ple81] 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 [BP89] proved that *the average distance heuristic* ([MI88], [RS83]), is a $\frac{4}{3}$-approximation. The *average distance heuristic*, however, does not seem to fit our priority model.

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.

Theorem 7 *No ADAPTIVE priority algorithm in the edge model can achieve an approximation ratio better than* $\frac{13}{11}$ *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 (required or Steiner), and $w(v_1, v_2)$ is the weight of the edge. For short, we will refer to a data item as $w(r, s) = p$, meaning this is an edge between a required and a Steiner node of weight p. The set of options is $\Sigma = \{accepted, rejected\}$; S_{Sol}, S_{Adv} are the respective solutions chosen by the Solver and the Adversary.

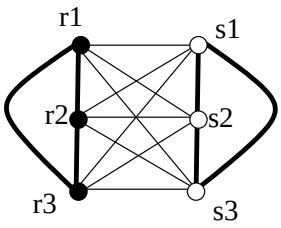

Figure 3: K_6 with three required and three Steiner nodes

The set of instances selected by the Adversary are graphs shown on Figure 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 are represented with a single data item. The set of instances T is a finite set of graphs K_6, shown on figure shown on Figure 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 bellow.

- Suppose the first edge selected by the Solver is e, $w(s_j, r_i) = 1$.
  - Case 1: If the Solver decides to accept e, then the Adversary makes s_j as far away from the remaining required nodes as possible, by leaving in Γ_2 edges between s_j and the other required nodes of weight $(2-\epsilon)$ only. The Adversary chooses another Steiner node, say s_k, and removes from Γ_2 edges between s_k and all required nodes of weight $2-\epsilon$, i.e., only edges of weight 1 are left. The Adversary selects a Steiner tree $S_{Adv} = \{(s_k, r_1)(s_k, r_2)(s_k, r_3)\}$. The choices for the Solver are either $S_{Sol} = \{(s_j, r_i)(s_k, r_1)(s_k, r_2)(s_k, r_3)\}$ of cost 4, or $S_{Sol} = \{(s_j, r_1)(s_j, r_2)(s_j, r_3)\}$ of cost $1+2(2-\epsilon)$ and she is awarded an approximation ratio $\rho = \frac{\mu(S_{Sol})}{\mu(S_{Adv})} = \frac{4}{3}$ or $\rho = \frac{\mu(S_{Sol})}{\mu(S_{Adv})} = \frac{5-2\epsilon}{3}$.
  - Case 2: If the Solver decides to reject e, then the Adversary does the opposite. He makes this Steiner node close to the remaining remaining required nodes, by restricting Γ_2 to edges between s_j and the remaining required nodes of weight 1 only. The Adversary further restricts Γ_2 by removing edges between the other two Steiner nodes and the required nodes of weight 1. The Adversary chooses $S_{Adv} = \{(s_j, r_1)(s_j, r_2)(s_j, r_3)\}$ of cost 3 while the best the Solver can do is to select a spanning tree of cost 4 and the Solver is awarded the approximation ratio $\rho = \frac{\mu(S_{Sol})}{\mu(S_{Adv})} = \frac{4}{3}$. (The possible solutions for the Solver are either two edges of weight 1 and edge of type $w(r, r) = 2$, or two edges of type $w(r_i, r_j) = 2$).
- The first edge selected by the Solver is e $w(r_i, s_j) = 2-\epsilon$.
  - Case 3: If the Solver decides to accept e, then the Adversary restricts Γ_2 to edges between s_j and the remaining required nodes of weight $(2-\epsilon)$ only. The Adversary then 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 $S_{Adv} = \{(s_k, r_1)(s_k, r_2)(s_k, r_3)\}$ of cost 3, while the best the Solver can do is select a Steiner tree of cost $3+2-\epsilon = 5-\epsilon$. The Solver is awarded the approximation ratio $\rho = \frac{C_{Sol}}{C_{Adv}} = \frac{5-\epsilon}{3}$.
  - Case 4: If the Solver decides to reject e, the Adversary restricts Γ_2 to edges from s_j to the remaining required nodes of weight 1; Γ_2 is further restricted to edges between the remaining required and Steiner nodes of weight $(2-\epsilon)$ only. The Adversary outputs a solution $S_{Adv} = \{(s_j, r_1)(s_j, r_2)(s_j, r_3)\}$ 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 is awarded an approximation ratio $\rho = \frac{\mu(S_{Sol})}{\mu(S_{Adv})} = \frac{4}{4-\epsilon}$ or $\rho = \frac{\mu(S_{Sol})}{\mu(S_{Adv})} = \frac{3(2-\epsilon)}{4-\epsilon}$, respectively.
- The first edge chosen by the Solver is e $w(r_1, r_j) = 2$.
  - Case 5: If the Solver decides, to accept e, then the Adversary restricts Γ_2 to edges between Steiner and required nodes of weight 1, only. The Adversary selects a Steiner tree $S_{Adv} = \{(s_1, r_1)(s_1, r_2)(s_1, r_3)\}$ of cost 3. The best the Solver can do is select a Steiner tree of cost 4. The Solver is awarder an approximation ratio $\rho = \frac{\mu(S_{Sol})}{\mu(S_{Adv})} = \frac{4}{3}$.

– If the Solver decides to reject e, then the Adversary awaits the next choice and behaves as in Case 1,2,3, 4, or 5, respectively, depending on the edge chosen by the Solver.

- If the Solver chooses an edge of type $w(s_i, s_j) = 1$, or $w(s_i, s_j) = (2 - \epsilon)$ and decides to add it to the tree. The Solver will loose the game and make the approximation ratio worse. The Adversary can restrict Γ_2 such that the edges remaining in Γ_2 make those two Steiner nodes far away from the required nodes. The Adversary will choose a tree of cost 3, while the Solver's Steiner tree would be of cost at least 4. If the Solver rejects all such edges, 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 awarded is infinity.

Considering the cases above, the best approximation ratio awarded to the Solver is $\rho = \frac{3(2-\epsilon)}{4-\epsilon}$. The Adversary chooses $\epsilon = 2/3$ and $\rho = \frac{3(2-\epsilon)}{4-\epsilon} = \frac{13}{11} = 1.18$. ▌

Next we present an algorithm achieving an approximation ratio of 1.75 for the MST problem with edge weights restricted to the interval $[1, 2]$. From here on, when we refer to a metric graph G, we also assume the restriction to the weights of the edges to be in the interval $[1, 2]$. The input to the algorithm are the edges of the graph and each edge is represented as the triple: names of the two endpoints and its weight. The algorithm runs in two stages. In the first stage we and 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. Formally, 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 $v_1, \ldots, v_k$ connected by a single acyclic connected component with edges from E. $E' = E - E_0 - E_1 \cup E_2$, 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. Following is a high level description of the algorithm.

Algorithm Contract

1. Contract required nodes.

   An edge between Steiner and required node is called lightweight 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 descending 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 the required nodes connected to the Steiner node 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.

2. Span a 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.

end

First we argue that the algorithm builds a tree. This follows from the fact that the algorithm contracts required nodes into one single node, in which the required nodes are connected by a path, and terminates when a single node remains in the instance.

Theorem 8 *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.

1. Analysis of contract phase.

   Let G be the instance graph. We want to convert 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 9 *For any metric graph* $G = (V = R \cup S, E)$*, with weight function on edges* $w : E \to [1, 2]$*, and any two nodes* $r_1, r_2 \in R$*, and any Steiner tree* τ *there exists a Steiner tree* τ_1 *in* $G_1 = contract(G, r_1, r_2)$ *such that* $Cost(\tau) \geq Cost(\tau_1) + 1$.

   Proof: Since we consider metric graphs with edge weights $[1, 2]$ 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 deleting such an edge creates a spanning tree in the contracted graph G_1. Therefore $Cost(\tau) \geq Cost(\tau_1) + 1$. ∎

   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 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 $G_1 = contract(G, \{r_{1,1}, \ldots, r_{1,t}\})$. Then we apply Lemma 9 $(t-1)$ 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 $Cost(\mathcal{A}_C)$ 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 $Cost(\mathcal{A}_C) = \sum_{i=1}^{k} Cost(\mathcal{A}_i) = \frac{7}{4}(Cost(\tau) - Cost(\tau_k))$, where τ_k is an optimal spanning tree in the contracted graph G_k.

2. During the second phase the algorithm builds a minimum cost spanning tree on the 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 as edges of type (s,s), (r,s), and (r,r), respectively.

Lemma 10 *Let $G=(S\cup R,E)$ be any metric graph with a weight function on edges $w:E\rightarrow[1,2]$, so that the lightweight degree of any Steiner node $s\in S$ is smaller 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: 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 and are restricted to 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.

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. 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 $|R_1|=k$. The combined cost of the edges $\mathcal{A}_1$, needed to connect the nodes in R_1 by a path is $Cost(\mathcal{A}_1)\leq 2(k-1)$. 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 k edges in the cut (S_1,R_1). If $k\leq 4m$ then the cost of edges of type (r,s) is greater then or equal to k, because each required node is connected to a Steiner node with an edge of weight at least 1. Otherwise, the cost is greater than $4m+1.4(k-4m)$, 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. The two sets of edges $\mathcal{D}_1$ and $\mathcal{A}_1$ are disjoint, because $\mathcal{A}_1$ are edges between two required nodes only, while $\mathcal{D}_1$ are edges of type (s,s) and (r,s). We consider the following cases next:

(a) Suppose $k\leq 4m$, then $m\geq 0.25k$ and $Cost(\mathcal{D}_1)\geq m-1+k\geq 1.25k-1$.

(b) Suppose $k>4m$ then $m<0.25k$ and $Cost(\mathcal{D}_1)\geq m-1+4m+1.4(k-4m)=1.4k-0.6m-1\geq 1.4k-0.15k-1\geq 1.25k-1$.

$$Cost(\mathcal{A}_1)\leq 2k-2, Cost(\mathcal{D}_1)\geq 1.25k-1\geq\frac{5}{8}(2k-2)=\frac{5}{8}Cost(\mathcal{A}_1),$$

$$Cost(\mathcal{D}_1)\geq\frac{5}{8}Cost(\mathcal{A}_1).$$

Similarly, $Cost(\mathcal{D}_i)\geq\frac{5}{8}Cost(\mathcal{A}_i),\ \forall i=1,\ldots,n$, where n is the number of connected components of Steiner nodes in G. 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

$$Cost(\tau) = Cost(\tau') + \sum_{i=1}^{n} Cost(\mathcal{D}_i) \geq Cost(\tau') + \frac{5}{8} \sum_{i=1}^{n} Cost(\mathcal{A}_i) \geq \frac{5}{8} \left(Cost(\tau') + \sum_{i=1}^{n} Cost(\mathcal{A}_i)\right).$$

∎

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 the Lemma 10. 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 10, and τ_k is the optimal tree τ. Applying the lemma 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 8. The cost incurred by the algorithm is $Cost(\mathcal{A}) = Cost(\mathcal{A}_C) + Cost(\tau_{A_2}) \leq \frac{7}{4}(Cost(T) - Cost(\tau_k)) + \frac{7}{4} Cost(\tau_k) = \frac{7}{4} Cost(T)$.

∎

Does the algorithm Contract fit the model of adaptive priority algorithms? The first phase must compute the lightweight degree of each Steiner node, 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. Bellow follows the description of the Adaptive Contract algorithm, achieving an approximation ratio of 1.8.

I. First we learn whether the number of Steiner nodes is 0, 1, or greater than 1. 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 we see such an edge, we reject it. Note that, the algorithm Contract does not use such edges and therefore those edges can be safely rejected. If there is such an edge we learn that the number of Steiner nodes is greater or equal to 2, and also remember their names. Then we go to step II.3 bellow.
- $\pi(e) = 3 - w(e)$, if e is an edge between a Steiner and a required node. If we see such an edge we reject the edge (note that e would be the heaviest weight edge). If the first data item has a priority value in the interval $[1, 2]$, then the number of Steiner nodes in the instance is exactly 1, and we go to step II.2 bellow.
- $\pi(e) = 2 + w(e)$, if e is an edge between two required nodes. If the first data item is of this type, then we accept the edge. 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. Then we continue in stage II.1 bellow.

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:

1. In this case, 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, s. In this case, we rejected the heaviest edge $e = (s, r_i)$ in stage I.

   (a) First we accept the lowest weight edge between r_i and a required node $e = (r_i, r_j)$, contracting them into a single required node, call it c_1 (Note that, this is the first step of Kruskal algorithm). We need to do this, 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 we reject the heaviest edge between r_i or r_j and an unseen required node, until all required nodes are seen. In doing this we compute the contracted graph $G_1 = \mathit{contract}(G, \{r_i, r_j\})$ and learn 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 the Steiner node. The degree is the number of required nodes 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 edges of type (r, s) with weight smaller or equal to 1.4, where $r \in R - \{r_1, r_2\} \cup \{c\}$ 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 nodes via a minimum cost spanning tree, using Kruskal[5] algorithm.

3. Suppose the instance has two or more Steiner nodes. First we learn the number and the names of Steiner and required nodes. Then we would 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 we continue with the algorithm Contract. Details follow bellow.

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

   (b) We learn the names and the number of the required nodes as in 2(a), by accepting the smallest weight edge between two required nodes and then computing the contracted graph.

   (c) We reject the heavyweight edges (weight strictly greater than 1.4) of type (r, s). For each Steiner node we dynamically compute the set of required nodes connected to it. Initially each such set is set to 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. At the end the lightweight degree of each Steiner node is simply the size of that set.

[5]Note that, the algorithm builds a minimum cost spanning tree using the Kruskal algorithm on any instance with three or four required nodes.

(d) At this point all remaining edges of type (r, s) will have weight smaller 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 will be added to the solution and the connected required nodes are contracted to a single 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} set.

(e) When no Steiner node has degree bigger than four all remaining (lightweight) 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 11 *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. The cost of such a tree is at most $2(r-1)$. If the optimal Steiner tree does not use Steiner nodes then 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 8 $Cost(T_{A_1}) \leq 1.75Cost(T_1)$. Combining the last two inequalities we have that $Cost(T_A) \leq 2 + 1.75Cost(T_1) = 1.75(Cost(T_1) + 1) + \frac{1}{4}$. By Lemma 9, $Cost(T) \geq Cost(T_1) + 1$, thus $Cost(T_A) \leq 1.75Cost(T) + \frac{1}{4}$. 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 $Cost(T_A) \leq 1.8Cost(T)$.

∎

3.4 Maximum Independent Set Problem

We study the performance of ADAPTIVE priority algorithms for the MIS problem and prove a lower bound of $\frac{2}{3}$ on the approximation ration for the MIS problem on graphs with maximum degree 3.

Theorem 12 *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 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 G_2 and G_3 have 6 vertices, with degrees 2 and 3 and cannot be distinguished by the Solver 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 $S_{Adv} = \{A, D, E\}$. The approximation ratio awarded is $\rho = \frac{3}{2}$.

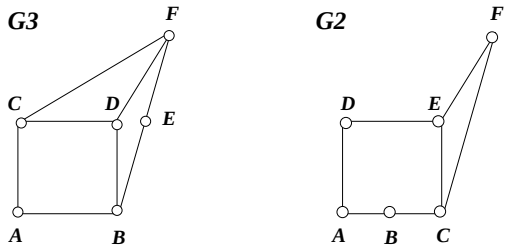

Figure 4: Nemesis graphs chosen by the Adversary

   - 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 possibilities solutions for the Solver are $\{C, D\}$, $\{A, E\}$, or $\{A, F\}$, while the Adversary selects $\{A, D, E\}$, and the approximation ratio is $\rho = \frac{3}{2}$.

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 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 $\{B, D, F\}$. The approximation ratio awarded to the Solver is $\rho = \frac{3}{2}$.
   - If the Solver decided to reject the the node of degree 2, then the Adversary presents an isomorphic copy of graph G_3 in Figure 4, in which the node chosen by the Solver is A. The possibilities solutions for the Solver are $\{B, C\}$,$\{B, F\}$,$\{C, E\}$, or $\{D, E\}$ of size 2, while the Adversary chooses $\{A, D, E\}$. The approximation ratio awarded to the Solver is $\rho = \frac{3}{2}$.

∎

4 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 with such priority model include scheduling problems, some graph optimization problems (vertex cover, Steiner trees, maximum clique, etc.), and also facility location and set cover problems. Memoryless priority algorithms have a restriction on what part of the instance 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. 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 π_{t+1}
- If $(\sigma_t = reject)$ then
  - Forget γ_t, i.e., delete it from current state.
- Remove the processed data item γ_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 a 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.

4.1 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 is as follows:

Game (Solver, Adversary)

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. If no such I exists then the Solver is awarded $\rho = 1$.

5. The Solver presents a valid solution S_{sol} for I such that $M \subseteq S_{sol}$.

6. The Solver is awarded the ratio $\rho = \frac{\mu(S_{sol})}{\mu(S_{adv})}$.

Lemma 13 *Let Γ_1 be any finite set of data items. There is a strategy for the Solver in the game defined above that achieves a payoff 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:

1. Suppose the Solver has a strategy for the game which guarantees to achieve a payoff 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 $\Gamma_{i+2} = \Gamma_{i+1} - \{\gamma : (\gamma \in \Gamma_{i+1}) \vee (\pi_{i+2}(\gamma) < \pi_{i+2}(\gamma_{i+2}))\}$. The memoryless algorithm removed from Γ_{i+1} all data items with π_{i+2} values smaller than $\pi_{i+2}(\gamma_{i+2})$ and 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 strategy that guarantees a payoff ρ the approximation ratio secured by the algorithm is at least ρ.

2. 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 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 looses the game and the Solver is awarded $\rho = 1$. 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 ratio ρ on any instance then the Solver will be awarded payoff ρ.

∎

Corollary 14 *If there is a strategy for the Adversary in the game defined above that guarantees a payoff ρ then there is no memoryless adaptive priority algorithm that achieves an approximation ratio better than ρ.*

Theorem 15 *There is an optimal strategy for the Solver that has the following property: once the Solver rejects one data item, she never accepts any later data items.*

Proof: To prove the theorem it suffices to show how any strategy for the Solver achieving payoff ρ, in which accepting and rejecting decisions are not separated into distinct phases can be converted to a strategy achieving the same payoff, in which the Solver never accepts after she has rejected a data item. Intuitively, the Solver gains nothing after rejecting a data item, because her profit remains the same as before considering the data item. Furthermore, since the Solver has forgotten it, 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 rejecting a data item in the model of memoryless algorithms could be beneficial to the Adversary but not to the Solver.

Say S be any strategy for the Solver, and π_i are the ordering functions used by the Solver during the rounds 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 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 $\pi_t : \Gamma_t \to \mathbb{R}^+$ from S;
  compute the first $\gamma_t \in \Gamma_t$ according to π_t, such that $\sigma_{\gamma_t} = accept$ under S.
- $R \leftarrow R \cup \{\gamma \mid (\gamma \in \Gamma) \vee (\pi(\gamma) < \pi(\gamma_t)) \vee (\sigma_\gamma = reject)\}$
- Set the ordering function and decision for S' as follows:
  $\pi'(\gamma) = \pi(\gamma)_{|\Gamma_t - R}$
  $\pi'(\gamma) = \infty, \ \forall \gamma \in R$
  $\sigma'_{\gamma_t} = accept$
- $\Gamma_{t+1} \leftarrow \Gamma_t - R$
  $t \leftarrow t + 1$

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 awarded to the Solver is the same regardless which strategy she plays. 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. ▌

4.2 Separation of between the class of adaptive priority algorithms and memoryless adaptive priority algorithms

We show that the class of adaptive priority algorithms are more powerful than memoryless adaptive priority algorithms. We consider the weighed independent set problem on cycles (WIS-2)[6], where the weight of the nodes is 1 or k, only, for some non-negative integer k. We show that memoryless adaptive priority algorithms cannot achieve an approximation ratio better than 2. Then we will show an adaptive priority algorithm 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 16 *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 5 and all isomorphic copies. 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.

- 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 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 $k \to \infty$, $\rho \to 2$.

[6]Note, that the Independent set problem in cycles with arbitrary weights can be solved exactly in polynomial time but our goal is to separate the power of two classes of algorithms.

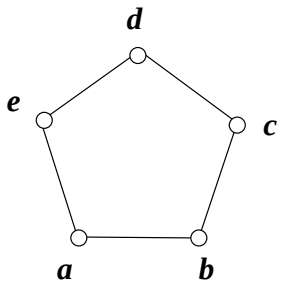

Figure 5: The nemesis graph for MIS problem.

- 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 $\rho = \frac{S_{Adv}}{S_{Sol}} \geq \frac{2k}{2} = k > 2$.

If the Solver never accepts a data item then the Adversary will select any non-empty independent set, and the Solver will be awarded an approximation ratio infinity. ▌

To show the separation between the models we present an adaptive priority algorithm 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 or the neighbor of a, that

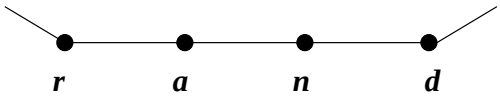

Figure 6: Fraction of the cycle considered.

hasn't been seen yet. Let the name of that node be n, see Figure 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 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 6, then d becomes the new r node. The main part of the algorithm is the following loop. We use $N(x)$ to denote the names of the neighbors of x in the instance.

repeat

- $n \leftarrow N(a) - \{r\}$
- if $weight(a) = k$ then $accept(a)$;
  $reject(n)$; $r \leftarrow n$
- if $weight(a) = 1$ then
  - if $weight(n) = k$ then $accept(n)$;
    $d \leftarrow N(n) - \{d\}$; $reject(a)$; $reject(d)$; $r \leftarrow d$
  - if $weight(n) = 1$ then $accept(a)$;
    $reject(n)$; $r \leftarrow n$
- a is set to point to the neighbor of r, not yet considered: $a \leftarrow N(r) - \{a, n\}$

until (decisions for each node of the cycle are made)

Bellow we give a formalization of the description as an adaptive priority algorithm. Notation used: $w(\gamma)$ 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** $(I = M)$

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

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

- $a = \min_{\gamma \in I, \gamma \notin M}\{\pi(\gamma)\}$. 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** $\sigma_a = accept$
    $S \leftarrow S \cup \{a\}$
  - $R = \{\gamma \mid \gamma \in adj(a) \setminus M\}$; $\forall \gamma \in R$: $\sigma_\gamma = reject$
    $\gamma_{last} = adj(a) \setminus adj(\gamma_r)$; $\gamma_r \leftarrow \gamma_{last}$

– $M \leftarrow M \cup \{a\} \cup \{adj(a) \setminus M\}$

Output S;
End.

Theorem 17 *ADAPTIVE WIS is 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 $ALG_i = wt(\text{ADAPTIVE WIS}(S_i))$ 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 $OPT_i = wt(S_i)$ 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. First we show that during Step 3, $\forall i \in \{1, 2, \ldots, n\}$ $\frac{OPT_i - ALG_i}{OPT_i} \leq \frac{1}{k+1}$. Suppose we can bound from above the ratio

$$\frac{OPT_i - ALG_i}{OPT_i} \leq x, \ \forall i \in \{1, 2, \ldots, n\}$$

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

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

$$1 - \frac{ALG_i}{OPT_i} \leq x$$

$$(1 - x)\, OPT_i \leq ALG_i$$

$$(1 - x) \sum_{i=1}^{n} OPT_i \leq \sum_{i=1}^{n} ALG_i = ALG(I)$$

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

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

Proof: 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 loose anything (the algorithm always takes that element).

$$ALG_n = OPT_n,\ \frac{OPT_n - ALG_n}{OPT_n} = 0,$$

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 $OPT_i \leq k$.

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

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 $OPT_i \leq k+1$:

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

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 $OPT_i \leq 1$:

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

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

We claim this case never happens. Every valid instance of the problem is a cycle and 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 5 then after the removal of the node with the smallest weight, there will be nodes with degree 1 and 2 left, thus there will be nodes with π values $1, 2$, or 3. We assume the number of nodes in the instance is smaller than five. On the next evaluation of π the neighbors of the node just removed, will have priorities 1, or 3 depending on their weight. The algorithm performs optimally on cycles of size 2 and 3, where the independent 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, $\{(a,1),(b,1),(c,1),(d,k)\}$, 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. ▌ Now finish the proof of Theorem 17 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 with 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:

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

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

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

Thus

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

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

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

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

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

As $k \to \infty$, $\quad \frac{ALG(I)}{OPT(I)} \to 1$.

∎

5 Open questions and future directions

Priority algorithms are a formal framework for greedy algorithms, and many greedy algorithms fit this model. We were able to show lower bounds for large class of algorithms for various graph optimization problems, which shows the weakness of the technique.

However, several questions remain unanswered in our current model for priority algorithms. Can we obtain an improved lower bound for the general Metric Steiner tree problem, with unrestricted edge weights? Our current upper and lower bounds do not match, and we would like to know whether we can close the gap. What lower bounds can we obtain for priority algorithms for the Maximum Independent Set problems for graphs of arbitrary degrees and for the Weighted Independent Set problem and other graph optimization problems?

Memoryless priority algorithms were proved less powerful for graph optimization problems, yet most of the known approximation algorithms are classified as memoryless algorithms. Can we design improved approximation schemes using memory?

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. Thus in future work we might want to consider extensions of the model, so that the extended model captures a larger class of algorithms. One such natural extension will be to consider a model where the algorithm is given access to "global information", such as the length of the instance. For graph problems this will be the number of edges or vertices in the graph. What lower bounds can we prove for priority algorithms in this extended model? [BNR02] and [BL03] showed that some of their lower bounds for scheduling problems do hold for this extended model and they considered this as evidence of robustness of the model.

Another extension of priority algorithms for graph problems is to redefine the notion of *local information* associated with a data item. For example, 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? A different direction would be to introduce randomization in the model. Our current lower bounds hold only for deterministic algorithms.

Greedy algorithms are simple and efficient. However, dynamic programming algorithms and backtracking algorithms are more powerful[7]. We would like to define similar general frameworks that would capture the defining characteristics of those powerful classes of algorithms. Can we design similar formal models and frameworks for proving lower bounds? Can we establish both negative results on their performance but also identify the strength of the technique and the problems on which it performs well? If we build formal models for the known efficient algorithm design paradigms (greedy, dynamic programming, hill-climbing, etc.) then negative results will show the limits of the known (natural) approaches to optimization.

Acknowledgments:
The authors are thankful to Allan Borodin, Joan Boyar, Jeff Edmonds, Valentin Kabanets, Kim Larsen, Vadim Lyubashevsky, and Morten Nielsen.

References

[AB02] Spyros Angelopoulos and Allan Borodin. On the power of priority algorithms for facility location and set cover. *5th International Workshop on Approximation Algorithms for Combinatorial Optimization*, September 2002.

[ABL74] Sanjeev Arora, Bela Bollobas, and Laszlo Lovasz. Proving integrality gaps without knowing the linear program. *Journal Of Computer And System Sciences*, 9:256–278, 1974.

[BEY98] Allan Borodin and Ran El-Yaniv. *Online Computation and Competitive Analysis.* Cambridge University Press, 1998.

[BL03] J. Boyar and K. Larsen. Preliminary results on priority algorithms for graph problems. *Manuscript*, 2003.

[BNR02] Allan Borodin, Morten N. Nielsen, and Charles Rackoff. (Incremental) Priority Algorithms. *Thirteen Annual ACM-SIAM Symposium on Discrete Algorithms*, January 2002.

[7] We proved that the Single Source Shortest Path problem on graphs with negative weight edges cannot be solved by any priority algorithm.

[BP89] Marshall Bern and Paul Plassmann. The Steiner problem with edge lengths 1 and 2. *Information Processing Letters*, 32:171–176, 1989.

[HY95] Magnus Halldorsson and Kiyohito Yoshihara. Greedy approximations of independent sets in low degree graphs. *Sixth International Symposium on Algorithms and Computation*, December 1995.

[Joh74] David S. Johnson. Algorithms for combinatorial problems. *Journal Of Computer And System Sciences*, 9:256–278, 1974.

[KMB81] L. Kou, G. Markowsky, and L. Berman. A fast algorithm for Steiner trees. *Acta Informatica*, 15:141–145, 1981.

[MI88] B. M. Maxman and M. Imase. Worst case performance of rayward-smith's Steiner tree heuristic. *Information Processing Letters*, 29:283–287, 1988.

[NN97] Richard E. Neapolitan and Kumarss Naimipour. *Foundations of Algorithms.* Jones & Bartlett Publishing, 1997.

[Ple81] J. Plesnìk. A bound for Steiner tree problem in graphs. *Math. Slovaca*, 31:155–163, 1981.

[RS83] V. J. Rayward-Smith. The computation of nearly minimal Steiner trees in graphs. *Internat J. Math. Educ. Sci. Tech.*, 14:15–23, 1983.

[Vaz01] Vijay V. Vazirani. *Approximation Algorithms.* Springer, 2001.

[Win87] P. Winter. Steiner problem in networks: a survey. *Networks*, 17:129–167, 1987.

A Priority Formalization for Adaptive Contract

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 weight $w \in [1; 2]$.

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: A Steiner tree T

1. $while(I = \emptyset)$

   (a) 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 π-first edge.

   i. If $\pi_1(e) = 0$ then $reject(e)$; Number of Steiner nodes $N_S \leftarrow 2$.
   ii. If $\pi_1(e) \in [1, 2]$ then $reject(e)$; Number of Steiner nodes $N_S \leftarrow 1$.
   iii. If $\pi_1(e) \in [3, 4]$ then $accept(e)$; Number of Steiner nodes $N_S \leftarrow 0$.

(b) If the number of Steiner nodes N_S is zero then we continue with Kruskal's algorithm. The edges of the instance are sorted according to:

$$\pi_t(v_i, t_i, v_j, t_j, w) = w$$

The π_t-first edge in this order is added to the solution, as long as the solution remains a forest.

(c) If the number of Steiner nodes N_S is 1 then:

- 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 π_2-first edge.

i. if $\pi_2(e) \in [1;2]$ then $accept(e)$; $c \leftarrow \{r_i, r_j\}$ is the new contracted node, which counts as a single required node; $N_R \leftarrow 1$; Create a set for the names of required nodes $Seen_R \leftarrow \emptyset$.

ii. if $\pi_2(e) \in [3,4]$ then $reject(e)$ and halt because the instance has one Steiner and one required nodes.

$t \leftarrow t+1$.

- if $t \geq 3$ then sort edges according to:

$$\pi_t(v_i, t_i, v_j, t_j, w) = \begin{cases} 3 - w & \text{if } (t_i = \mathbf{r}) \wedge (t_j = \mathbf{r}) \wedge (v_i \in c) \wedge (v_j \notin Seen_R) \\ 5 - w & \text{if } (t_i = \mathbf{r}) \wedge (t_j = \mathbf{s}) \ \wedge (w > 1.4) \\ 3 + w & \text{if } (t_i = \mathbf{r}) \wedge (t_j = \mathbf{s}) \ \wedge (w \leq 1.4) \wedge Degree(x) \geq 5 \\ 5 + w & \text{if } (t_i = \mathbf{r}) \wedge (t_j = \mathbf{s}) \ \wedge (w \leq 1.4) \wedge Degree(x) < 5 \\ 7 + w & \text{if } (t_i = \mathbf{r}) \wedge (t_j = \mathbf{r}) \end{cases}$$

Let $e = (v_i, t_i, v_j, t_j, w)$ be π_t-first edge.

i. if $\pi_t(e) \in [1,2]$ then $reject(e)$; $Seen_R \leftarrow Seen_R \cup \{v_j\}$, here we compute the $G_1 = contract(G, c)$; $N_R \leftarrow N_R + 1$; $Degree(s) \leftarrow N_R$.

ii. if $\pi_t(e) \in [3, 3.6)$ then $reject(e)$; $Degree(s) \leftarrow Degree(s) - 1$, here we compute the lightweight degree of the Steiner node.

iii. if $\pi_t(e) \in [4.4, 5]$ then $accept(e)$, here we contract the Steiner node with the required nodes incident to it via lightweight edges $S_{contract} = S_{contract} \cup \{v_i\}$.

iv. if $\pi_t(e) \in [6, 7]$ then $reject(e)$.

v. if $\pi_t(e) \in [8, 9]$ then $accept(e)$ as long as it does not create a cycle, otherwise reject it (Here we use the Kruskal's algorithm to build a minimum cost tree spanning the required nodes).

(d) If the number of Steiner nodes $N_S \geq 2$ then 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.

- if $\pi_t(e) = 0$ then $reject(e)$, because e is of type (s, s), here we learn names and numbers of Steiner nodes: $Steiner \leftarrow Steiner \cup \{v_i, v_j\}$; $N_S \leftarrow N_S + 2$.

- if $\pi_t(e) \in [3,4]$ then $reject(e)$ and Halt, because e is of type (r,s) and the instance has only one required node.
- if $\pi_t(e) \in [1,2]$ then $accept(e)$. We contract the two required nodes into a single required node. $c_1 \leftarrow \{r,r\}$; $N_R \leftarrow N_R + 1$; and $Seen_R \leftarrow \emptyset$. We switch to a new priority function by setting a flag $Phase \leftarrow 2$.

(e) If $N_S \geq 2$ and $Phase = 2$ then sort edges according to:

$$\pi_t(v_i,t_i,v_j,t_j,w) = \begin{cases} 3-w & \text{if } (t_i=\mathbf{r}) \wedge (t_j=\mathbf{r}) \wedge (v_i \in c_1) \wedge (v_j \notin Seen_R) \\ 5-w & \text{if } (t_i=\mathbf{r}) \wedge (t_j=\mathbf{s}) \wedge (w > 1.4) \\ 7-w & \text{if } (t_i=\mathbf{r}) \wedge (t_j=\mathbf{s}) \wedge (v_j = MaxD) \wedge (D_{MaxD} \geq 5) \\ 9-w & \text{if } (t_i=\mathbf{r}) \wedge (t_j=\mathbf{s}) \wedge (v_j = S) \wedge (D_S > 0) \\ 9-w & \text{if } (t_i=\mathbf{r}) \wedge (t_j=\mathbf{s}) \wedge (D_{v_j} = \max_{u \in Steiner} D_u) \wedge (D_{v_j} \geq 5) \\ 11-w & \text{if } (t_i=\mathbf{r}) \wedge (t_j=\mathbf{s}) \\ 10+w & \text{if } t_i=\mathbf{r}; t_j=\mathbf{r}) \end{cases}$$

Let $e = (v_i,t_i,v_j,t_j,w)$ be π_t-first edge.

- if $\pi_t(e) \in [1,2]$ then $reject(e)$, here we compute $G_1 = contract(G,c_1)$; $Seen_R \leftarrow Seen_R \cup \{v_j\}$; $N_R \leftarrow N_R + 1$; For every Steiner node $u \in Steiner$ let R_u be the set of the required nodes to which u is connected to, $R_u \leftarrow Seen_R \cup \{c_1\}$;
- if $\pi_t(e) \in [3,4]$ then $reject(e)$; We reject a heavyweight edge and adjust the lightweight degree of the Steiner node v_j $R_{v_j} \leftarrow R_{v_j} - \{v_j\}$. Let $MaxD$ points to the Steiner node of highest lightweight degree at the moment: $MaxD \leftarrow s$, and $s \in Steiner$ and $R_s = \max_{u \in Steiner} |R_u|$.
- if $\pi_t(e) \in [5,6]$ then $accept(e)$. We initialize S to be the Steiner node of maximum degree and will add all lightweight edges incident to it to the current solution. $S \leftarrow MaxD$; $R_S \leftarrow R_S \cup \{v_j\}$. $D_S \leftarrow D_{MaxD}$; $D_{MaxD} \leftarrow 0$.
- if $\pi_t(e) \in [7,8]$ then $accept(e)$, here we accept all lightweight edges incident to the Steiner node S and update the lightweight degree of S, $D_S \leftarrow D_S - 1$. We also update the lightweight degrees of the remaining Steiner nodes:
  For all $u \in Steiner - \{S\}$ such that $|R_u \cap R_S| \geq 2$
  do $R_u \leftarrow R_u - (R_u \cap R_s) \cup \{t_v\}$ **end-do**.
  If we have added all the lightweight edges of S we need to find the Steiner node whose lightweight degree is maximum: if $D_S = 0$ then Remove S from the remaining set of Steiner nodes $Seen_S \leftarrow Seen_S - \{S\}$. $MaxD \leftarrow s$, and $s \in Steiner$ and $R_s = \max_{u \in Steiner} |R_u|$.
- if $\pi_t(e) \in [9,10]$ then $reject(r)$.
- if $\pi_t(e) \in [11,12]$ then $accept(e)$, if e does not form a cycle with the already added edges, else $reject(r)$ (build a minimum cost spanning tree on required and contracted nodes).

end(while)

End **Adaptive Contraction**

ECCC ISSN 1433-8092
http://www.eccc.uni-trier.de/eccc
ftp://ftp.eccc.uni-trier.de/pub/eccc
ftpmail@ftp.eccc.uni-trier.de, subject 'help eccc'