

Priority Algorithms

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

June 2, 2003

Abstract

This paper reviews the recent development of the formal framework of priority algorithms for scheduling problems [BNR02]; the extension of the model for facility location and set cover [AB02], and graph problems [DI02]. We pose some open questions and directions for future research.

1 Introduction

Greedy algorithms are a natural approach to optimization. They are simple and efficient. However, a precise formal model of greedy algorithms had not appeared in the literature prior to the framework of priority algorithms developed in [BNR02]. The definition of the formal model of priority algorithms is important because it allows for analysis of the power and limitations of a large class of algorithms. The study of lower bounds is important because it establishes a negative result about the performance of a large class of algorithms. When the lower bound is tight, the algorithm achieving the desired ratio is optimal for the given class. Lower bounds are useful for the design of an algorithm, as they guide possible directions for improvements, and serve as an evaluation of the power of the model. The formal framework of priority algorithms allows us to evaluate a very large class of algorithms known in the literature as greedy, and derive non-trivial lower bounds on the performance of these algorithms for a large domain of problems.

This paper is organized in five sections. First we present the definition of priority algorithms as it appeared in [BNR02] and summarize their results. In Section 2 we review online algorithms and competitive analysis and their relationship to deriving lower bounds on the performance of priority algorithms. In Section 3 we present the first extension of the priority algorithm framework for the facility location problem and the set cover. In Section 4 we present our work generalizing the priority algorithm framework to any problem domain, and applying it to graph optimization problems. And lastly, we pose some open questions and future research directions, in Section 5.

1.1 Greedy and greedy-like algorithms for scheduling problems

The scheduling problem has been studied for a long time and many of the proposed algorithms, exact and approximation, are greedy. Are these algorithms the best we can

hope for or are there still greedy algorithms that can improve the performance of the existing ones? To answer this question [BNR02] introduced two formal models of greedy algorithms, adaptive and fixed priority, which capture the defining characteristics of greedy algorithms. Their model only applied to scheduling problems, but was extending in [AB02] and [DI02] to include almost any combinatorial optimization algorithm. In scheduling problems, the algorithm is given a list of jobs and must output a solution, which is a schedule of some of the jobs on a set of machines. Later, we will precisely define what a scheduling problem is, but for now we would like to focus on the structure of a greedy algorithm for a “general scheduling problem” rather than on the specific parameters and objective functions classifying the different scheduling problems. [BNR02] observed that most (not all) of the greedy algorithms for scheduling problems shared the following common features:

- “**one input at a time**”. Meaning, that the algorithm considers a single job and has to make a decision of whether to schedule the job or not. The algorithm does not look at the whole instance prior to making a decision.
- the decision of the algorithm is “**irrevocable**”, that is, the algorithm cannot change its mind later, after observing larger portion of the input.
- the order in which jobs are considered is determined by a **priority function**, which orders not just the set of the jobs in the input but all possible jobs.

Based on how often the algorithm gets to re-order the inputs [BNR02] defined **fixed** and **adaptive** priority algorithms.

FIXED PRIORITY ALGORITHM

(input: a set of jobs S)

Ordering: Determine, without looking at S , a total ordering of all possible jobs

while not empty (S)

$next :=$ index of job in S that comes first in the order

Decision: Decide if and how to schedule job J_{next} , and remove J_{next} from S

ADAPTIVE PRIORITY ALGORITHM

(input: a set of jobs S)

while not empty (S)

Ordering: Determine, without looking at S , a total ordering of all possible jobs

$next :=$ index of job in S that comes first in the order

Decision: Decide if and how to schedule job J_{next} , and remove J_{next} from S

The previous two models differ in the ability of the algorithm to reorder data inputs. The adaptive priority algorithms have the flexibility to reorder data inputs and can simulate the simple fixed priority algorithms. Based on how the decision is made, [BNR02] defined greedy and not-necessarily greedy algorithms. They called a greedy algorithm one “*which makes the decision with the goal of optimizing the objective function as if the input observed is the last.*” That is greedy algorithms locally optimize the objective function. Priority algorithms not-necessarily greedy do not have the restriction on how to make a decision. For example, a greedy scheduling rule will always schedule a job, if a machine is available. A not-necessarily greedy algorithm may decide to not schedule such a job.

A natural question to ask is whether adaptive priority algorithms are more powerful than fixed priority algorithms, also is the greedy rule a restriction? [BNR02] proved

that the class of adaptive priority algorithms is more powerful than the fixed priority algorithms. They did not show a proof of separation between greedy and not-necessarily greedy algorithms.

Next we will discuss what a scheduling problem is, and some of the results [BNR02] were able to prove. A scheduling problem is defined as a *machine configuration*, and an input list of jobs. The algorithm is given a finite set of machines, each machine has a description of its computation power. The machines could be identical or, some machines can be parallel. The input to the algorithms is a *list of jobs*. Each job is described by a four tuple $J = (r, d, p, w)$, where r is the release time, d is the deadline, p is processing time, and w is the weight or the profit that the algorithm claims if it chooses to schedule the job. A scheduling algorithm with a given machine configuration must produce an assignment of jobs to machines satisfying a set of constraints. [BNR02] considered non-preemptive scheduling, where once a job is scheduled to a machine it must complete its execution. The quality of the solution is determined by the value of the solution for the objective function specified for the problem. For example, for the profit maximization problem the algorithm's goal is to schedule jobs so that the combined profit of the jobs scheduled is maximized. For the minimum makespan problem the goal is to produce a schedule so that the completion time of the jobs scheduled on any machine is minimized. Although [BNR02] considered a variety of job scheduling problems, maximization and minimization problems, we will present some of their results for the most simple instance of job scheduling with profit maximization, namely interval scheduling on set of identical machines. Interval scheduling is a restricted version of the scheduling problem described above, where $p = d - r$ holds for each job in the sequence. The objective function is to maximize the profit, which is the total length of the intervals scheduled.

Next we summarize some of the interval scheduling results proved in [BNR02]. Proofs of some of the results will be deferred to the next section where we present competitive analysis of online algorithms and define a similar framework for proving lower bounds for priority algorithms.

1. [BNR02] studied the performance of fixed priority algorithms. First they considered interval scheduling for a single machine configuration. They showed that the Longest Processing Time heuristics achieves an approximation ratio of 3. LPT heuristics orders the set of all possible intervals according to their processing time once in non-increasing order prior to making any decisions. Then it looks at each interval and greedily tries to schedule it, if possible. [BNR02] were able to prove that no priority (fixed or adaptive) can achieve an approximation ratio better than 3. This is a good result which shows that a simple greedy heuristic performs optimally for both classes of algorithms fixed and adaptive priority, for a single machine configuration.

When considering multiple machine machine configurations, [BNR02] showed that LPT again achieves an approximation ratio of 3 and also they proved that no fixed priority greedy algorithm can achieve an approximation ratio better than 3. This result shows that the LPT heuristic is optimal in the class of fixed priority greedy algorithms for a multiple machine configuration.

2. Are adaptive priority algorithms more powerful than the fixed priority algorithms? They are certainly more complex, since the algorithm reorders the remaining intervals in the instance prior to making a decision. The answer is yes, adaptation helps the algorithm. Recall that no fixed priority algorithm can have an approximation

ratio better than 3 for interval scheduling with proportional profit on a multiple machine configuration (the bound was tight because LPT achieved it). [BNR02] proved that there exists an adaptive priority algorithm, CHAIN-2, which achieves an approximation ratio of 2 on a two machine configuration. They later generalized the algorithm for an even number of machines.

The lower bound proved for this class of algorithms, however, is not tight. CHAIN-2 was shown to be a 2-approximation, and the lower bound on the performance of adaptive priority algorithms is $1 + \frac{\sqrt{17}-3}{2}$. Can we close the gap? Is the lower bound not good enough or there is another algorithm not yet discovered? These are questions that remain unanswered.

3. Interval scheduling for arbitrary profits is a difficult problem and both fixed and adaptive perform poorly. The weight (profit) of each interval in the sequence is no longer proportional to its length. Let Δ and δ be the maximum and the minimum profit per unit of all intervals in the instance, respectively, i.e., $\delta_i = \frac{w_i}{p_i}$, $\Delta = \frac{\max_i \delta_i}{\min_i \delta_i}$ and $\delta = \min_i \delta_i$.

[BNR02] showed that no deterministic priority greedy algorithm can achieve a constant approximation ratio, establishing a lower bound $\Omega(\Delta)$ on the performance of greedy deterministic priority algorithms. However, if Δ and δ are known to the algorithm then randomization helps. [BNR02] presented a randomized fixed priority not-necessarily greedy achieving an approximation ratio $O(\log n)$.

This result is significant because it shows that randomization helps. Thus one direction for extending the current framework would be to build a formal model for randomized priority algorithms.

1.2 Evidence of priority algorithms framework robustness

We would like to know whether the current lower bounds would hold for various extensions of the model. A natural extension to the model would be to give the algorithm access to global information. This extended model would capture an even larger set of the known greedy algorithms, and lower bounds in this model would be very important. For example, suppose the algorithm knows the length of the instance. [BNR02] proved that their lower bound of 3 on the approximation ratio for the interval scheduling problem with proportional profit holds for this extended model.

Another interesting extension is to let the algorithm see two inputs (rather than one) or any fixed number of jobs at a time. They proved a lower bound of 2 on the approximation ratio of any fixed priority not-necessarily greedy algorithms that can see two jobs at a time but must schedule one of them.

2 Online computation, competitive analysis, and priority algorithms

In this section we introduce online algorithms and competitive analysis. The framework for deriving lower bounds for priority algorithms is borrowed from competitive analysis of online algorithms. We will give two examples of deriving lower bounds for deterministic and randomized paging algorithms and then give an example of deriving a lower bound on the approximation ratio achieved by fixed priority algorithms for interval scheduling with proportional profit.

What is an online algorithm? [BEY98] defined an online algorithm as follows: “*In online computation, an algorithm must produce a sequence of decisions that will have an impact on the final quality of its overall performance. Each of these decisions must be made based on past events without secure information about the future. Such an algorithm is called an online algorithm.*” Priority algorithms resemble online algorithms. Both algorithms do not see the whole instance, rather they observe the input one item at a time. Both algorithms must make an irrevocable decision about a data item, based on the partial input seen so far. The differences between online algorithms and priority algorithms is in the order in which the algorithms see the input. Priority algorithms can use arbitrarily complex functions to order the data items. In the case of online algorithms, the Adversary or other constraints define the order.

2.1 Competitive ratio

The competitive ratio is a worse-case complexity measurement of the performance of the online algorithm as it compares the quality of the solution output by the algorithm to that of the optimal offline algorithm. Suppose we are given a cost minimization problem. An online algorithm A is c -competitive if there exists a constant α , which does not depend on the length of the instance, such that for all valid instances I the following holds: $A(I) \leq c \cdot OPT(I) + \alpha$, where $A(I)$ is the cost of the online algorithm A on instance I , and $OPT(I)$ is the optimal offline cost on I . The competitive ratio is defined as the infimum over all constants c such that the algorithm A is c -competitive.

Competitive analysis of online algorithms is viewed as a two player zero-sum game. The players are the Online Algorithm and the Adversary. Zero-sum games best capture the antagonistic relationship between the Online Algorithm and the Adversary. The Online Algorithm seeks to minimize the competitive ratio and the Adversary seeks to maximize it. In what follows we will refer to the Online Algorithm player as the Algorithm for short.

Consider the following paging problem. We are given a cache with capacity k pages, and a slow memory with capacity $N = k + 1$ pages. An input to the algorithm is a sequence of page requests, say I , where each page is numbered $1, 2, \dots, N$. The request sequence of pages I must be served. If a page requested is in the cache, i.e., the request is a hit, then the cost of servicing the request is 0. If the request is a miss then a page from the cache must be evicted and the requested page must be swapped in. The cost of servicing a miss is 1. The problem is to design a demand paging, page replacement online algorithm and minimize the total number of pages evicted on any sequence of N pages.

We analyze the performance of the class of deterministic algorithms first, and show that there is a strategy for the Adversary achieving a payoff of $\frac{\text{Algorithm}(I)}{\text{Adversary}(I)} = k$. Here we have a cost minimization problem and the Online player seeks to minimize the cost of the solution, thus to minimize the competitive ratio. The class of algorithms we consider are deterministic, which means that Algorithm player has chosen a pure strategy from the set of all possible strategies prior the beginning of the game. The pure strategy chosen is known to the Adversary. The Adversary possesses unrestricted power and can request any page so that the cost of processing the request for Algorithm is maximized. One round of the game consists of a request issued by the Adversary and response by the Algorithm, indicating how the request will be served. That is, Algorithm must show which page will be evicted if the request is a miss, otherwise Algorithm does nothing (we are considering performance of demand paging algorithms, only). A winning strategy for

the Adversary is to always request the page that Algorithm has evicted at the previous round of the game. The Adversary can choose to end the game at any time. At the end, the Adversary processes the request sequence offline, and the ratio of the online cost to the offline cost is awarded to Algorithm player.

The strategy described above causes the Algorithm to fault on any page request. Let the length of the instance I is n then the cost incurred by the Algorithm is $A(I) = n$. The Adversary serves the entire sequence offline as follows. The cache has size k thus prior to serving the next k requests the Adversary compares his cache with the sequence and if necessary will swap in the page that that will be requested but is not in his cache. Thus on a sequence of k requests adversary misses at most 1. Thus for a sequence of n requests the cost for the adversary is at most $\frac{n}{k}$. The desired ratio $\frac{Algorithm(I)}{Adversary(I)} = k$ is achieved. Thus we conclude that k is a lower bound on the competitive ratio of any deterministic online paging algorithm.

2.2 A lower bound for fixed priority greedy algorithms

Proving a lower bound on the performance of priority algorithms is also viewed as a request-response zero-sum game. Suppose we have a profit maximization problem then an algorithm A is a c approximation if $c \cdot A(I) \geq OPT(I)$ holds for any valid instance I , where $A(I)$ is the profit gained by the algorithm A , and $OPT(I)$ is the best possible profit for the instance I .

To prove a bound on the performance of any fixed priority algorithm, we define a request-response zero-sum game between two players: the Algorithm and the Adversary. The Adversary is no longer as powerful as before, and the reason is that priority algorithms define an order on the data items and this ordering restricts the sets of instances that the Adversary can present. In abstract, the game between the Algorithm and the Adversary aims to build a nemesis instance which exhibits the worse performance of the Algorithm player. Initially, the Adversary selects a finite set of data items. The Algorithm without looking at the set determines an ordering on the data items. Then the game proceeds in rounds. At each round the Algorithm looks at the highest priority data item and makes a decision. The Adversary may choose to remove data items not yet seen by Algorithm, depending on the the strategy, or may choose to end the game. The payoff to the Algorithm is $\frac{Algorithm(I)}{Adversary(I)}$, where $Algorithm(I)$ and $Adversary(I)$ are the qualities of the solutions output by the Algorithm and Adversary, respectively.

We illustrate the technique with an example. We will show a lower bound on the performance of any priority algorithm for the interval scheduling on a single machine with proportional profit problem.

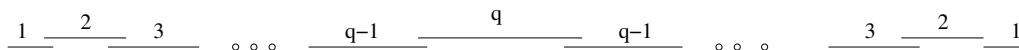


Figure 1: The large sequence used by Adversary.

The game between Algorithm and Adversary is as follows: Initially the Adversary selects $4(2q - 1)$ intervals in total, which form two sequences - large and short. The large sequence contains $2q - 1$ intervals of lengths $1, 2, \dots, q$, as shown on Figure 1. Each long interval intersects the two adjacent long intervals by ϵ , except the leftmost and the rightmost intervals of length 1, which overlap only one long interval. In addition, each long interval is intersected by three short intervals. Say the length of the long interval is

p , then the three short intervals for this long interval have length $\frac{p-2\epsilon}{3}$. Thus the total number of intervals is $3(2q-1) + (2q-1) = 4(2q-1)$.

The Algorithm assigns priorities to the set of all possible intervals. Say i is the interval with the highest priority. In this particular case the Adversary has a strategy for the the game and will win after the Algorithm plays his first move:

1. Suppose Algorithm rejects the interval. The Adversary then removes all the remaining intervals and schedules i . The Algorithm gained profit 0, while the Adversary gained profit proportional to the length of i . The payoff to the Algorithm is $\frac{Algorithm}{Adversary} = \frac{0}{|i|} = 0$ (The approximation ratio is ∞).
2. Suppose the Algorithm schedules i . The Adversary's strategy then depends on the length of the interval i :
 - (a) If i is a short interval with length $\frac{p-2\epsilon}{3}$ the Adversary removes all but the long interval intersecting i and schedules the long interval. The Algorithm's profit is $Algorithm = \frac{p-2\epsilon}{3}$. The Adversary's profit is $Adversary = p$. The payoff to the Algorithm is $\frac{Algorithm}{Adversary} = \frac{p-2\epsilon}{3p}$ (the approximation ratio is $\frac{3p}{p-2\epsilon}$).
 - (b) If i has length 1 the Adversary removes all but the three short intervals contained by the long interval, and the long interval of length 2 that intersects with i . The Algorithm's profit is $Alg = 1$. The Adversary's profit is $3 \cdot (\frac{1-2\epsilon}{3}) + 2 = 3 - 2\epsilon$. The payoff to the algorithm is $\frac{Algorithm}{Adversary} = \frac{1}{3-2\epsilon}$ (the approximation ratio is $3 - 2\epsilon$).
 - (c) If i has length j such that $1 < j < q$ the Algorithm's profit is $Alg = j$. The Adversary removes all but the intersecting intervals and gains profit $OPT = j - 1 + j + 1 + 3\frac{j-2\epsilon}{3} = 3j - 2\epsilon$. The payoff to the Algorithm is $\frac{Algorithm}{Adversary} = \frac{j}{3j-2\epsilon}$ (the approximation ratio is $\frac{3j-2\epsilon}{j}$).
 - (d) At last if the length of i is q Algorithm gains profit q . The Adversary removes all but the intervals intersecting i and gains profit $= 2j - 2 + 3\frac{q-2\epsilon}{3} = 3q - 2 - 2\epsilon$. The payoff to the Algorithm is $\frac{Algorithm}{Adversary} = \frac{q}{3q-2-2\epsilon}$ (the approximation ratio is $\frac{3q-2-2\epsilon}{q}$).

By making q arbitrarily large and ϵ arbitrarily small, the Adversary ensures that the approximation ratio achieved by any priority algorithm is arbitrarily close to 3.

In general, to prove lower bounds on performance of priority algorithms we define a zero-sum game between the Algorithm and the Adversary. The existence of a priority algorithm with a guaranteed approximation ratio, can be used by the Algorithm as a strategy in the game against the Adversary. On the other hand a strategy for the Adversary establishes a lower bound. The moves allowed to the Algorithm and the Adversary players are modeled after the structure of the corresponding class of algorithm. For fixed priority algorithms the game has structure such that Algorithm orders the set of data items once. For the remaining rounds, t moves of the Algorithm are decisions for data items, and moves of the Adversary are modifications of the remaining unseen set of instances. When we want to establish a lower bound on the performance of adaptive priority algorithms the game is slightly different. During each round the Algorithm a data item with the highest priority and makes decision on it. The Adversary then observes the move, and based on that information restricts the set of data items according to the strategy. The Adversary then decides whether to continue the game or not. This combinatorial game will be illustrated with examples in Section 3 and 4.2.

2.3 Lower bound on the performance of randomized paging algorithms

Next we illustrate the usage of Yao's technique for proving lower bounds on the performance of randomized algorithms. For cost minimization problems Yao's principle states that a lower bound on the performance of the best randomized algorithm can be obtained by evaluating the performance of the best deterministic algorithm with respect to probability distribution of inputs selected by the Adversary.

We consider the paging problem. Let the size of the cache be k and the number of pages be $N = k + 1$. Denote the competitive ratio of any randomized paging algorithm as c_R . Our goal is to show that no randomized algorithm can achieve an approximation ratio better than H_k , i.e., $c_R \geq H_k$. To prove a lower bound of H_k on the performance of the best randomized paging algorithm, it suffices to choose a probability distribution on requests sequences, and to prove that the ratio of the expected online cost of the best deterministic algorithm to the expected optimal offline cost is greater than H_k .

1. We must select a probability distribution on inputs. Let \mathcal{P} be the probability distribution where: $next_request \in_u \{1, 2, \dots, k + 1\}$. That is any page could equally likely be chosen as the next request.
2. We evaluate the expected online cost of the best deterministic algorithm. For any deterministic algorithm and any request of the sequence $\Pr[\text{request } i \text{ is a miss}] = \frac{1}{k+1}$. This holds for the best deterministic algorithm, call it A , as well. Suppose the length of the request sequence I is n . Then $\mathbf{Exp}_{\mathcal{P}}[A(I)] = \frac{n}{k+1}$.
3. We evaluate the expected offline cost. Define a phase as the longest sequence of k distinct page requests. Then a phase ends before the $k + 1$ -st distinct request. The optimal offline algorithm can service the k pages in a phase incurring only one mis. Thus we need to estimate the number of phases. Let $X(n)$ be a random variable denoting the number of phases in a sequence of n requests. $\mathbf{Exp}_{\mathcal{P}}[OPT(I)] = \mathbf{Exp}_{\mathcal{P}}[OPT(n)] \leq \mathbf{Exp}[X(n) + 1]$. For each phase i we let Y_i be a random variable denoting the number of requests in i . Note that Y_i s are independent and identically distributed random variables, due to the choice of the distribution on inputs \mathcal{P} .

$$\lim_{n \rightarrow \infty} \frac{n}{\mathbf{Exp}[X(n)]} = \mathbf{Exp}[Y_i]$$

By the coupon collectors problem $\mathbf{Exp}[Y_i] = (k + 1)H_k$

$$\lim_{n \rightarrow \infty} \frac{\mathbf{Exp}_{\mathcal{P}}[A(I)]}{\mathbf{Exp}_{\mathcal{P}}[OPT(I)]} = \frac{\frac{n}{k+1}}{\frac{n}{\mathbf{Exp}[Y_i]}} = \frac{\mathbf{Exp}[Y_i]}{k + 1} = H_k$$

We proved that any randomized algorithm cannot achieve an approximation ratio better than H_k for the paging problem.

3 Priority algorithms for facility location and set cover

The first extension of the priority algorithms framework work appeared in [AB02]. Angelopoulos and Borodin defined a model of priority algorithms for the unrestricted facility

location and the set cover problems. Their goal was to define a model of priority algorithms for those problems, to prove lower bounds on performance of adaptive and fixed priority algorithms, and to evaluate the performance of the existing algorithms in the literature that can be classified as priority algorithms.

[AB02] were able to prove the following tight bounds on performance of adaptive and fixed priority algorithms for unrestricted facility location and set cover.

1. Performance of adaptive priority algorithms for set cover problem.

Slavik showed that the greedy set cover is a $\ln n - \ln \ln n + \Theta(1)$ approximation for the set cover problem. Borodin and Angelopoulos proved that no adaptive priority algorithm can achieve an approximation ratio better than $\ln n - \ln \ln n + \Theta(1)$. The greedy set cover heuristics belongs to the class of adaptive priority algorithms and thus is an optimal algorithm for this class.

2. Performance of priority algorithm for the facility location problem.

- [AB02] considered the uniform metric facility location problem, where the opening costs are identical, and the connection costs obey triangle inequality. They showed that no adaptive priority algorithm can achieve an approximation ratio better than $\frac{4}{3}$, when the connection costs are $\{1, 3\}$. The strategy used by the Adversary to derive the lower bound above delivers a lower bound of $\Omega(\log n)$ for the general facility location problem. This lower bound is matched by a $O(\log n)$ greedy adaptive priority approximation algorithm, showing that the bound is tight, and the greedy heuristics is optimal for the class of adaptive priority algorithms.
- They also were able to show a tight bound of 3 on the approximation ratio achieved by fixed priority algorithms for uniform metric facility location. Thus the known greedy approximation algorithm is optimal within the class of fixed priority algorithms.

[AB02] results showed that the best know algorithms for facility location in arbitrary spaces and set cover are optimal within the class of adaptive priority algorithms algorithms.

In what follows we will present precise problem definitions and the priority models built in [AB02] for the problems. Then we prove a lower bound on the approximation ratio of adaptive priority algorithms for the facility location problem.

An instance of the facility location problem is a set of facilities \mathcal{F} and a set of cities \mathcal{C} . Each facility has an opening cost f_i . For each city $j \in \mathcal{C}$ a non-negative connection cost c_{ij} represents the charge that must be paid to connect city j to the facility f_i . The problem is to connect each city in \mathcal{C} to a facility in \mathcal{F} . The objective function is to minimize the combined connection and opening costs incurred. To build a model for priority algorithm a specification of the input format and the solution format is needed. [AB02] defined an instance I of the facility location problem to be a set of facilities, where each facility is encoded as a tuple: $(f_i, c_{i1}, c_{i2}, \dots, c_{i|\mathcal{C}|})$, where f_i and c_{ij} are as above. The solution is a set of facilities $S \subseteq \mathcal{F}$ that the algorithm decided to open. The value of the objective function is determined as:

$$Alg(I) = \sum_{f_i \in S} f_i + \sum_{j \in \mathcal{C}} \min_{f_i \in S} c_{ij}$$

The first term represents the opening cost incurred by the solution of the algorithm, the second term is the combined connection cost.

The algorithm chooses the order in which it considers the facilities, and casts irrevocable decision, whether to open a facility or not. Once the algorithm chooses to open or not a facility, it can not change its mind later. The class of adaptive priority algorithms can re-order the facilities after each decision while fixed priority algorithms determine the order once.

Next we will show Adversary's strategy in the zero-sum game, proving that no adaptive priority algorithm can achieve an approximation ratio better than $\Omega \log n$ for the facility location problem in arbitrary spaces. An instance of the problem is a collection of cities \mathcal{C} , and set of facilities \mathcal{F} . Let $|\mathcal{C}| = n$, where $n = 2^k$.

We say that a facility f_i covers a city $j \in \mathcal{C}$ if $c_{ij} < \infty$, otherwise we say that the city j is not covered. Let S be set of facilities that remain in the sequence during any round of the game between the Adversary and the Algorithm. The Adversary selects as the initial input set S all possible facilities that cover exactly $\frac{n}{2}$ cities at cost 1, the remaining cities are not connected, i.e., the cost is ∞ . The facility opening cost for each facility is n . Two facilities are said to be complementary if they together cover all the cities. Obviously, each facility f has exactly one complementary facility \bar{f} and they are both in S at the beginning of the game. The game proceeds in rounds. Let $\bar{\mathcal{C}}$ denote the set of cities not covered by the Algorithm during any stage of the game. Initially, $|\bar{\mathcal{C}}| = |\mathcal{C}| = n$. At the beginning of round t of the game, the Adversary maintains the following invariant: 1. The number of the uncovered cities is $|\bar{\mathcal{C}}| = \frac{n}{2^{t-1}}$. 2. Each remaining facility $f \in S$ can cover exactly $\frac{n}{2^t}$ cities, that is, if $f \in S$ at the beginning of round t then $|f \cap \bar{\mathcal{C}}| = \frac{n}{2^t}$.

Notice that for each $f \in S$ then $\bar{f} \in S$, and thus the remaining facilities can cover the remaining uncovered cities. Obviously, the invariant holds initially. At the beginning of round 1, the Algorithm has not selected any facilities yet, thus $|\bar{\mathcal{C}}| = n$, and each of the facility selected in the initial set by the Adversary covers exactly $\frac{n}{2}$ cities.

At each round:

- Algorithm's move: The Algorithm selects a facility f from S and decides whether to open f or not.
- Adversary's move:
 1. If the Algorithm decides to reject the facility then the Adversary removes all the remaining facility except \bar{f} and ends the game. The Adversary outputs $\{f, \bar{f}\}$ as a solution, while the Algorithm failed to produce a solution.
 2. If the Algorithm decides to open f . Then the Adversary computes $\bar{\mathcal{C}} \leftarrow \bar{\mathcal{C}} \setminus f$. The Adversary removes from S all facilities covering more than $\frac{|\bar{\mathcal{C}}|}{2}$ cities. That is f remains in S if $|f \cap \bar{\mathcal{C}}| \leq \frac{|\bar{\mathcal{C}}|}{2}$ otherwise f is removed. Unless only one city remained uncovered, in which case the Adversary leaves the set S unchanged.

Claim 1 *The invariant is preserved during each round of the game.*

Proof: by induction on t . Suppose the invariant holds at the beginning of round t we show that it is preserved at the beginning of round $t+1$. If case 1 happens then the game ends. Thus we assume case 2 happens. By the hypothesis, at the beginning of round t The number of uncovered cities is $\frac{n}{2^{t-1}}$ and all facilities in S are pairs of complementary facilities such that each facility covers exactly $\frac{n}{2^t}$ cities. The algorithm chose a facility in S . Thus the number of uncovered cities at the end of the round is $\frac{n}{2^t}$, as wanted. Then the Adversary removed from S all but those facilities covering exactly half of the

$\frac{n}{2^t}$ uncovered cities. That is each facility $f \in S$ covers exactly $\frac{n}{2^{t+1}}$ uncovered cities. Thus the invariant is maintained at the beginning of round $t + 1$.

The outcome of the game is either Algorithm fails to produce a valid solution (if case 1 was to happen), or Algorithm opened exactly $\log n$ facilities at cost n each. In addition the connection cost for the n cities is 1. Thus the combined cost of the solution for the Algorithm is $n \log n + n$. The Adversary opens two complementary facilities and incurs opening cost 2, thus the total cost of the Adversary's solution is $2n + n$. And the desired bound is achieved.

4 Priority models for graph problems

In this section we will review the work on priority algorithms in [DI02]. Our goal was to capture the defining characteristics of large class of algorithms, known in the literature as greedy. We defined a general framework of adaptive and fixed priority algorithms, and memoryless priority algorithms, which is independent of the specific problem domain. The framework can be used to model scheduling problems, facility location problems, graph optimization problems. We abstracted away the information which is specific to encode a job, or a facility, a vertex, or an edge in the graph, and focused on the structure of the algorithms. Then we applied the framework to graph optimization problems and derived lower bounds on the performance of fixed and adaptive priority algorithms. An important question to answer is: are the three classes of algorithms needed? Are they equivalent in power? The existing greedy algorithms for graph problems seem to fit the framework of fixed priority algorithms and memoryless algorithms. Adaptive priority algorithms can simulate fixed priority algorithms. We would like to know whether adaptive priority algorithms are more powerful than fixed priority algorithms. We also want to evaluate the power of the priority algorithms for specific graph problems. We also want to know whether the existing greedy algorithms are optimal or not.

In what follows we show that the three classes of algorithms are distinct.

More importantly, we show that imposing a memory restriction on the algorithm limits its power, that is we prove that memoryless algorithms are less powerful than adaptive priority algorithms. Precise definition of memoryless algorithms will appear in Section 4.4

4.1 Fixed and adaptive priority algorithms

Assume a generic representation of a graph as a set of vertices, or a set of edges. We view an instance as a set of *data items*. Let the type of data item be Γ , thus an instance as a set of items of type Γ , $I \subseteq \Gamma$. Not every subset of data items constitutes a valid instance. Let the set of options (decisions) for each data item be Σ . Then a solution for an instance I , can be represented as $\{(\gamma_i, \sigma_i) | \gamma_i \in I\}$. For example, for k -colorings of graphs on n nodes, we need to assign colors to nodes. So $\Sigma = \{1, \dots, k\}$, and Γ should correspond to the *information available about a node* when the algorithm has to color it.

A data item corresponding to a vertex can naturally be encoded as 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 $1, \dots, n$, *AdjList* is a list of *NodeNames*. 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 list of neighbors. Σ varies from problem to problem; often, a solution is a subset of the nodes, corresponding to $\Sigma = \{accept, reject\}$.

For some problems the instance graph is more appropriately encoded as a set of edges. We call this model an *edge model*. The data items requiring a decision are 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 its edges. Again, the options Σ are determined by the problem, with $\Sigma = \{accept, reject\}$ when a solution is a subgraph.

As in [BNR02], we distinguish between algorithms that order their data items at the start, and those that reorder at each iteration. A fixed priority algorithm orders the data items at the start, and proceeds according to that order. The format for a fixed priority algorithm is as follows.

FIXED PRIORITY ALGORITHM

Input: instance $I \subseteq \Gamma$, $I = \{\gamma_1, \dots, \gamma_d\}$.

Output: solution $S = \{(\gamma_i, \sigma_i) | i = 1, \dots, d\}$.

- Determine a criterion for ordering the decisions, based on the data items:
 - Choose $\pi : \Gamma \rightarrow \mathbb{R}^+ \cup \{\infty\}$
- Order I according to $\pi(\gamma_i)$, from smallest to largest
- **repeat**
 - Let γ_i be the next data item according to the order
 - Make an irrevocable decision $\sigma_i \in \Sigma$ and update the partial solution S
 - 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 have the power to reorder the remaining decision points during execution.

ADAPTIVE PRIORITY ALGORITHM

Input: instance $I \subseteq \Gamma$, $I = \{\gamma_1, \dots, \gamma_d\}$.

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 solution S , and t to 1.

- repeat**
 - Determine an ordering function
 - $\pi_t : \Gamma \rightarrow \mathbb{R}^+ \cup \{\infty\}$
 - Order $\gamma \in U$ according to $\pi_t(\gamma)$
 - Observe the first unseen data item $\gamma_t \in U$
 - 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 and increment t
- until** (there are no data items not yet considered, $U = \emptyset$)

Output S

The current decision made depends in an arbitrary way on the data points seen so far. The algorithm also has implicit knowledge about the unseen data points: no unseen

point has a higher priority under π_t than γ_t .

We claim that fixed and adaptive priority algorithms are not equivalent in power and will prove that next. We define the **ShortPath** graph optimization problem as follows. 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. Each edge is represented a triple (u, v, w) , meaning the edge goes from u to v and has weight w . The set of options is $\Sigma = \{accepted, rejected\}$. The well known Dijkstra algorithm which belongs to the class of adaptive priority algorithms solves the single source shortest path (SSSP) problem exactly, thus it can solve the ShortPath problem as well. Next we show that fixed priority algorithms perform poorly for the ShortPath problem. In the game between the Adversary and the Algorithm player Adversary's strategy is to force the Algorithm to make a "wrong" or unfavorable decision. The Adversary can compute the priorities of the edges (we are considering fixed priority deterministic algorithms) and is allowed to remove edges from the graph as long as the Algorithm hasn't considered them. We give

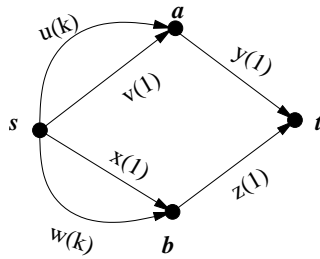


Figure 2: The initial sequence of data items selected by the Adversary: x, y, z, u, v, w .

short references to data items, for example u is an alias for data item (s, a, k) , which is an edge from s to a , with weight k . Depending on the data item selected by Algorithm and decision made, the Adversary restricts S to appropriate subset as follows. The Adversary can remove data items from the set of instances not yet considered. Since the Algorithm must assign distinct priorities to all edges. One of the edges y and z must appear first in the order. We assume, without loss of generality, that the priority of y is higher than the priority of z , thus the Algorithm would consider y before z . In this case the Adversary removes edge v and w from all instances, i.e., $S = \{u, x, y, z\}$. The Adversary waits until the Algorithm considers edge y . Remember that edge z will be considered after edge y . The set of decision options are $\Sigma = \{accepted, rejected\}$. The Adversary's strategy is:

1. The Algorithm decides $\sigma_y = rejected$. Then the Adversary presents the instance: $\{y, v\}$. And outputs solution $S_{Adv}(I) = \{y, v\}$, while the Algorithm failed to construct a path.
2. The Algorithm decides $s_y = taken$. Then the Adversary presents $\{y, z, u, x\}$. If the Algorithm picks edge z , then the Algorithm failed to satisfy the solution constraints. The only valid solution left for the Algorithm is $S_{Alg} = \{y, u\}$, while the Adversary selects $S_{Adv}(I) = \{z, x\}$. The advantage gained by the Adversary is: $\rho = \frac{S_{Adv}}{S_{Alg}} = \frac{k+1}{2}$.

If the priority of edge z is higher than y then the Adversary removes edges x and u from the original graph, leaving edges w, z, v, y and uses the same strategy as before. By giving arbitrarily heavy weights to edges u and w the Adversary can achieve any advantage over any fixed priority algorithm. Thus we proved that no fixed priority algorithm can solve the ShortPath with any constant approximation ratio. The Dijkstra's algorithm for the single source shortest path problem can be classified as an adaptive priority algorithm. It solves a more general problem, it builds a tree in which the path from a source s to any other vertex of the graph is minimum, thus it solves the ShortPath problem exactly. Thus we established a separation between the classes of fixed and adaptive priority algorithms and proved that adaptive priority algorithms are more powerful.

4.2 An example lower bound proof for adaptive priority algorithms

Next we would like to show an example lower bound proof for adaptive priority algorithms to illustrate the adaptive request response zero-sum game between the Adversary and the Algorithm. We considered the weighted vertex cover problem. The simple greedy heuristic achieves an approximation ratio 2 and fits into the adaptive priority model, and we show that no adaptive priority algorithm can do better. Thus the greedy 2-approximation algorithm is optimal for the class of adaptive priority algorithms.

We view the set cover in a node model. Here, the algorithm orders nodes based on their names, weights, and adjacency lists, and must make an irrevocable decision whether to include a node in the cover or not. We show a winning strategy for the Adversary player in the zero-sum game between the Algorithm and the Adversary. The Adversary sets S to be a finite set of $K_{n,n}$ graphs. Each vertex could have weight $w_1 = 1$ or $w_2 = n^2$ and is connected to all nodes on the opposite side. An optimum cover is to take the nodes on one side of the bipartite. The Adversary waits until the first time one of the following three events occur.

- **Event 1:** The Algorithm takes a node with weight $w_2 = n^2$.
- **Event 2:** The Algorithm rejects a node.
- **Event 3:** The Algorithm takes $n - 1$ nodes of weight $w_1 = 1$ from either side of the bipartite graph.

If **Event 1** occurs the Adversary fixes the weights of all node on the opposite side to $w_1 = 1$.

If **Event 2** occurs the Adversary fixes the weights of all unseen nodes on the opposite side to $w_2 = n^2$ and the weights of remaining nodes on the same side to $w_1 = 1$.

If **Event 3** occurs the Algorithm 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 $w_2 = n^2$ and remaining nodes on the other side are set to $w_1 = 1$.

Until Events 1 or 2 happen, the Algorithm is taking nodes of weight $w_1 = 1$. Eventually of the three events, one has to occur. Thus we analyze the performance of the strategy based on how well the Adversary performs in the different cases.

- **If Event 1 occurs first.** In this case the Adversary selects all nodes from side B as a Vertex Cover, and the cost of the cover is $C_{Adv} = nw_1 = n$, because until Event 1 occurs the nodes are assigned weight w_1 and after Event 1 occurs the Adversary assigns all remaining nodes on side B weight w_1 . The Algorithm has committed

and added a heavyweight node. Therefore Algorithm will incur cost at least n^2 which gives a ratio $\rho \geq n \geq 2$.

- If **Event 2** occurs first. The Algorithm has rejected node from side A . Algorithm will have to add all nodes on side B to cover the edges of the rejected node. Because Event 3 hasn't occurred there must be at least 2 unseen nodes on side B , therefore Algorithm will have a vertex cover of cost at least $2n^2$. Since the Adversary assigns all other nodes on side A weight $w_1 = 1$ the Adversary incurs cost at most $n^2 + n - 1$ by taking all nodes on side A . This give ratio $\rho \rightarrow 2$ as $n \rightarrow \infty$.
- If **Event 3** occurs first. Algorithm has committed to $n - 1$ nodes on side A . The Adversary assigns the remaining node on side A weight n^2 and all nodes on side B as $w_1 = 1$. The Adversary takes side B . The optimal choice for Algorithm is to take all nodes on side B as well the $n - 1$ nodes on side A giving cost $2n - 1$. This give ratio $\rho \rightarrow 2$ as $n \rightarrow \infty$.

We conclude that no adaptive priority algorithm can achieve an approximation ratio better than 2 for the weighted vertex cover problem and the greedy 2-approximation algorithm is optimal within the class.

4.3 Other results

We considered the ShortPath problem for graphs with negative weights, but no negative weight cycles. This problem can be solved by a dynamic programming algorithm and we wanted to know whether such a powerful technique was really needed. Could an adaptive priority algorithm solve this problem? We showed that no adaptive priority algorithm can solve the problem.

We also considered the independent set for graphs of degree at most 3, and the Metric Steiner tree problems. Here we were not so fortunate and could not prove tight bounds for these problems on the performance of fixed and adaptive priority algorithms. We considered the Steiner Tree problem for metric spaces in an edge model. The standard fixed priority algorithm for metric Steiner tree (building MST on the subgraph induced by the required nodes) achieves an approximation ratio of 2. We showed that no adaptive priority algorithm can achieve an approximation ratio better than 1.18, even for the special case where every positive distance is between 1 and 2, and were able to show an improved algorithm for this special case, in the adaptive priority model, achieving approximation ratio 1.875. The current gap between the lower and the upper bound is large. How can we close the gap? Can we improve the lower bound? Can adaptation help? Are there adaptive priority algorithms that can achieve approximation ratio better than 2 for the generic Metric Steiner tree problem? These are questions we hope to answer.

4.4 Memoryless Adaptive Priority Algorithms

We define a subclass of adaptive priority algorithms, in which the algorithm is restricted to remember only part of the instance processed. Most of the known greedy heuristics for graph optimization problems can be classified as memoryless algorithms so we decided to formalize this notion and evaluate the power of memoryless priority algorithms and compare it to that of adaptive priority algorithms. We want to know whether the existing algorithms are optimal or not, by comparing the class in which they belong to a different class of algorithms. For this purpose we need to define a formal model

of memoryless algorithm and a request-response zero-sum game in which we can prove lower bounds. The formal framework of memoryless algorithm is given below.

MEMORYLESS ADAPTIVE PRIORITY ALGORITHM

Input: instance $I \subseteq \Gamma$, $I = \{\gamma_1, \dots, \gamma_d\}$.

Output: solution vector $S = \{(\gamma_i, \sigma_i) \mid \sigma_i = \textit{accepted}\}$

- Initialization:

Let U be the set of unseen data items, S' be an empty partial solution, and t is a counter. $U \leftarrow I$; $S' \leftarrow \emptyset$, $t \leftarrow 1$.

- 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 $\sigma_t \in \{\textit{accepted}, \textit{rejected}\}$.

- if ($\sigma_t = \textit{accepted}$) then

i) update the partial solution: $S' \leftarrow S' \cup \{\gamma_t\}$

ii) determine an ordering function: $\pi_{t+1} : \Gamma \rightarrow \mathbb{R}^+ \cup \{\infty\}$

iii) order $\gamma \in U \setminus \{\gamma_t\}$ according to π_{t+1}

- remove the processed data item γ_t from U : $U \leftarrow U \setminus \{\gamma_t\}$; increment $t \leftarrow t + 1$.

until (there are no data items not yet considered, $U = \emptyset$)

- Output S'

What are the differences between memoryless and adaptive priority algorithms?

1. **re-ordering the inputs:** Priority algorithms with memory can reorder the remaining data items in the instance after each decision, while memoryless algorithms can re-order the remaining data items only after casting an *accepting* decision.
2. **state:** Memoryless algorithms forget data items that were rejected, while adaptive priority algorithms have no memory restriction and keep in their state information about all data items and corresponding decisions made.
3. **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.

The definition of the zero-sum game used to prove lower bounds for adaptive priority algorithms no longer reflect the capabilities of a memoryless algorithm. Thus a new definition is needed to prove lower bounds. We define a two person game between 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 $\Sigma = \{\textit{accepted}, \textit{rejected}\}$ is the set of options available for each data item. Let T be a finite collection of instances of Π . The game between the two players Solver and Adversary is as follows:

Game (Solver, Adversary)

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

2. Adversary picks any subset $\Gamma_1 \subseteq \Gamma$ with at least one instance $I \subseteq \Gamma_1, I \in T$;
Sets $R \leftarrow \emptyset$, and a counter $t \leftarrow 1$.
3. **repeat until** (Adversary decides to enter Endgame, or $\Gamma_t = \emptyset$)
begin (Round t)
 - (a) Let γ_t be the next data item in Γ_t according to the order π_t
 - (b) Solver picks a decision σ_t for γ_t
 - if ($\sigma_t = \textit{accepted}$) then Solver does the following:
 - updates her memory: $M \leftarrow M \cup \{\gamma_t\}$
 - removes γ_t from the sequence: $\Gamma_t \leftarrow \Gamma_t \setminus \{\gamma_t\}$
 - defines a new total order π_{t+1} on Γ
 - else ($\sigma_t = \textit{rejected}$)
 - Adversary updates $R \leftarrow R \cup \{\gamma_t\}$
 - Solver removes γ_t from Γ_t : $\Gamma_t \leftarrow \Gamma_t \setminus \{\gamma_t\}$
 - (c) Adversary defines $\Gamma_{t+1} \subseteq \Gamma_t$.
 - (d) if Adversary chooses to end the game or ($\Gamma_{t+1} = \emptyset$) then enter Endgame
otherwise $t \leftarrow t + 1$ and the next round begins.
- end;** (Round t)
4. Endgame: 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 Solver is awarded $\rho = 1$.
5. Solver presents a solution S_{sol} for I such that $M \subseteq S_{sol}$.
6. Solver is awarded the ratio $\rho = \frac{\mu(S_{sol})}{\mu(S_{adv})}$.

We showed that there is a strategy for Solver in the game of incomplete information 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 . Thus if there is a strategy for Adversary that guarantees a payoff ρ then a lower bound is established and there is no memoryless adaptive priority algorithm that achieves an approximation ratio better than ρ .

We also showed that there is an optimal strategy for Solver that has the following property: once Solver rejects one data item, he never accepts any later data items by proving that any strategy for 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 Solver never accepts after it has rejected a data item. Now we can classify most of the known greedy algorithms in the literature as memoryless adaptive priority algorithms. Viewed in the framework of a game between Solver and Adversary, that corresponds to a strategy of Solver to never accept after a rejection. We would like to know whether memoryless algorithms are less powerful than adaptive priority algorithms.

We were able to show a separation between the power of the two classes. We looked at the weighted independent set problem on cycles, and viewed the problem in the node model. A valid instance is a 2-regular graph. Each data item is a triple (*name*, *weight*, adjacency list), where $|\text{adjacency list}| = 2$, and $\textit{weight} \in \{1, k\}$. We proved that no memoryless adaptive priority algorithm can achieve

an approximation ratio better than 2 for the WIS on degree 2 graphs. Since any strategy for Solver (any memoryless algorithm) can be converted to a strategy with distinct accepting and rejecting phases, then we only need to consider the cases when the first decisions made by the algorithm are accepting.

Adversary selects as a set of instances all pentagons, where the weights of the nodes are 1 or k . Solver must output an order of all possible data items. The strategy for Adversary is:

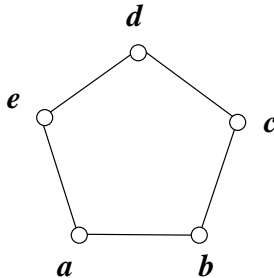


Figure 3: The nemesis graph for MIS problem.

Case 1: Solver picks data item (a, k) first, and decides $\sigma_{(a,k)} = \textit{accepted}$. Then Adversary presents the instance: $\{(a, k), (b, k), (c, 1), (d, 1), (e, k)\}$. Adversary outputs the solution $S_{Adv}(I) = \{b, e\}$, while the best Solver can do is $S_{Sol}(I) = \{a, c\}$, The advantage gained by Adversary is: $\rho = \frac{S_{Adv}}{S_{Sol}} \geq \frac{2k}{k+1} = 2 - \frac{2}{k+1}$.

Adversary can choose k arbitrarily large, thus as $k \rightarrow \infty$ thus the approximation ratio will be arbitrarily close to 2.

Case 2: Solver picks $(a, 1)$ data item first and decides $\sigma_{(a,k)} = \textit{accepted}$. Then Adversary truncates the remaining sequence and presents the instance: $\{(a, 1), (b, k), (c, 1), (d, 1), (e, k)\}$. Adversary outputs the solution $S_{Adv}(I) = \{b, e\}$, while the best Solver can do is output $S_{Sol}(I) = \{a, c\}$, The advantage gained by Adversary is: $\rho = \frac{S_{Adv}}{S_{Sol}} \geq \frac{2k}{2} = 2$.

□

To show a separation we exhibited an adaptive priority algorithm for the weighted independent set problem on 2-regular graphs, achieving an approximation ratio $1 + \frac{2}{k-1}$ on any instance I , with weights $\{1, k\}$. This proves that adaptive priority algorithms are more powerful than memoryless algorithms. Can we design improved approximation schemes for other graph optimization problems using memory? The known algorithms belong to the less powerful class of memoryless algorithms?

4.5 Limitations of the priority algorithms framework

The priority algorithms framework presented above does not capture all the known algorithms considered greedy in the literature. Here we give a high level description of the best known approximation algorithm for the independent set problem in bounded degree graphs [BF94]. The algorithm proceeds in iterations and at

each iteration full knowledge of the input graph is required. The algorithm starts at an arbitrary independent set and looks for an improvement. An improvement is a connected subgraph of size at most $O(\log n)$ nodes such that the symmetric difference of the current independent set with the improvement is a larger independent set. The algorithm finds improvements by considering all possible connected subgraphs. There are at most $n\Delta^\sigma$ connected subgraphs of size σ of a graph with n nodes and maximum degree Δ . When no further small improvements are possible the algorithm computes a complement of the current independent set and recursively runs the algorithm on the complemented graph. The bigger of the two independent sets are selected. Upon termination, the algorithm guarantees for any Δ and any $k > 0$ an approximation ratio $\frac{OPT(G)}{Alg(G)} \geq \rho_\Delta + \frac{1}{k}$, where $\rho_\Delta = \frac{\Delta+3}{5}$ for even Δ and $\rho_\Delta = \frac{\Delta+3.25}{5}$ for odd Δ . The running time of the algorithm is $n\Delta^{O(k \cdot \Delta^{4k} \cdot \log n)}$.

Is the algorithm described above a priority algorithm? No, it is not a priority algorithm. Not according to our current formalization of priority algorithms. Recall that two of the defining characteristics of priority algorithms are “one input at a time” and “irrevocable decision”. Both of these conditions are violated by the algorithm. The algorithm scans the full instance prior to making a decision. Decisions are made for set of data items at a time, since the algorithm is considering improvements. Furthermore, each node is considered many times, and the decision made is revocable. The algorithm makes a recursive call to itself with the complement of the current independent set, thus a node which was rejected initially could be accepted later, if the independent set built during the recursive call is bigger than the original IS.

5 Future work

Why is this work important? Because we believe this is a general approach of evaluating heuristics and algorithm design paradigms by proving lower bounds for all algorithms in a given class. We defined the framework of priority algorithms as a formal model for greedy algorithms. In this framework we were able to analyze the performance of all the algorithms in the class. Lower bounds establish the weaknesses of the technique while upper bounds determine the kinds of problems the technique can successfully be applied.

Can we design formal models for other algorithmic design techniques and frameworks for proving lower bounds? Can we establish both negative results of their performance and 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) then negative results will show the limits of all natural approaches to optimization.

5.1 Tighten the bounds, and separate the existing models

Many of the current bounds for priority algorithms are not tight. Can the existing upper bounds be improved by designing new algorithms, or are the current lower

bounds not good enough? For example, can we obtain an improved lower bound for the general Metric Steiner tree problem, with unrestricted edge weights? Our improved upper bound holds for metric graphs with edge weights $\{1, 2\}$ only. What lower bounds can we obtain for the Maximum Independent Set problems for graphs of arbitrary degrees?

There are problems that we didn't consider, for example what lower bounds can we prove for the graph coloring problem?

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 algorithms using memory?

5.2 Extended priority models

Next we would like to extend the priority model and prove lower bounds for those algorithms that belong to the extended models. We have seen that there are greedy algorithms that do not fit the current model of priority algorithms. We would like to extend the model so that it can capture a larger class of algorithms. For example we would like to consider a model where the algorithm is given access to "global information". The number of nodes or vertices in the graph is global information. What lower bounds can we prove for priority algorithms in this extended model?

Another extension of priority algorithms for graph problems is to redefine the notion of *local information* associated with a data item to span the data items of its neighbors, assuming the problem is viewed in the vertex model. Would that additional information, given to the algorithm during the decision-making process, increase the power of the priority algorithms?

Would randomization help? What kind of lower bounds can we prove for randomized priority algorithms? Our current lower bounds hold only for deterministic algorithms. How can we apply the developed lower bound techniques for analysis of for online algorithms to proving lower bounds and upper bounds for randomized priority algorithms?

5.3 Beyond greedy algorithms

Greedy algorithms are simple and efficient. However, dynamic programming algorithms and backtracking algorithms are more powerful. We would like to define similar general frameworks that capture the defining characteristics of those powerful classes of algorithms.

Let Γ be the type of a data item. A data item can be a vertex, an edge, a job, etc. Let $\Sigma = \{\sigma_1, \dots, \sigma_n\}$ be the set of options (decisions) for each data item. Σ is problem dependent. A backtracking algorithm builds a depth first search pruning tree. We would like to capture this in a formal model. As before we would define two classes of algorithms fixed and adaptive, depending on whether the data items can be reordered prior to making decisions or not. Thus a fixed backtracking algorithm builds the search tree by considering data items in fixed order. The decisions

are irrevocable. However, the notion of an irrevocable decision has changed, but not by much. For each data item a subset of the possible decisions is chosen and the subset cannot be changed later. Following is an example of what the framework of fixed backtracking algorithm might be.

FIXED BACKTRACKING ALGORITHM

Input: instance $I \subseteq \Gamma$, $I = \{\gamma_1, \dots, \gamma_d\}$.

Output: solution vector $S = \{(\gamma_i, \sigma_i) | 1 \leq i \leq d\}$

Initialization:

Let U be the set of unseen data items $U \leftarrow I$;

S' be an empty partial solution $S' \leftarrow \emptyset$;

I' be an empty partial instance $I' \leftarrow \emptyset$; and $t \leftarrow 1$.

Determine an ordering function $\pi : \Gamma \rightarrow \mathbb{R}^+ \cup \{\infty\}$

Order $\gamma \in U$ according to $\pi(\gamma)$

Repeat

Let $\gamma_t \in U$ be the next unseen data item according to π .

Let $\Sigma_t \subseteq \Sigma$ be the set of options for γ_t , where Σ_t is consistent with the current partial solution.

For each $\sigma_t \in \Sigma_t$ in parallel:

BRANCH on (γ_t, σ_t) , and simulate the algorithm for remaining $\gamma \in U$

Update the partial solution $S' \leftarrow S' \cup \{(\gamma_t, \sigma_t)\}$

Remove the processed data item γ_t : $U \leftarrow U - \{\gamma_t\}$;

Increment $t \leftarrow t + 1$.

until (there are no data items not yet considered, $U = \emptyset$)

Output the best solution found.

The algorithm builds a tree. A search tree built by a fixed backtracking algorithm, at level i will make a decision about data item γ_i which is the i -th according to the priority function. However the decisions options selected Σ_t could be different for each parallel branch.

Adaptive backtracking algorithms can reorder data items prior to making decisions. The decision is irrevocable and the subset of options per data item once chosen can not be changed. The proposed framework is given below.

ADAPTIVE BACKTRACKING ALGORITHM

Initialization:

Let U be the set of unseen data items $U \leftarrow I$;

S' be an empty partial solution $S' \leftarrow \emptyset$;

I' be an empty partial instance $I' \leftarrow \emptyset$; and $t \leftarrow 1$.

Repeat

Determine an ordering function $\pi_t : \Gamma \rightarrow \mathbb{R}^+ \cup \{\infty\}$

Order $\gamma \in U$ according to $\pi_t(\gamma)$

Let $\gamma_t \in U$ be the next unseen data item according to π .

Let $\Sigma_t \subseteq \Sigma$ be the set of options for γ_t , where Σ_t is consistent with the current partial solutions.

For each $\sigma_t \in \Sigma_t$ in parallel:
 BRANCH on (γ_t, σ_t) , and simulate the algorithm for remaining $\gamma \in U$
 Update the partial solution $S' \leftarrow S' \cup \{(\gamma_t, \sigma_t)\}$
 Remove the processed data item γ_t : $U \leftarrow U - \{\gamma_t\}$;
 increment $t \leftarrow t + 1$.
 until (there are no data items not yet considered, $U = \emptyset$)
 Output the best solution found.

As before the algorithm branches on decision for a data item. However, the decisions at different branches could be different and also the order of the remaining items in the separate branches could differ. We would like to answer similar questions as the ones posed before. Are the two classes of algorithms equivalent in power? We are interested both in negative results and upper bounds.

Clearly, if a backtracking algorithm is allowed to inspect all the leaves of the tree it would be able to produce an optimal solution. However, we would like to restrict the computation to polynomially many leaves in the search tree, and relate the quality of the solutions produced by backtracking algorithms and the fraction of the leaves of the search tree inspected. For example, suppose the length of the instance is n . Let $|\Sigma|$ denote the number of decision options available for each data item. The kinds of claims we would like to prove have the following formulation:

“Any backtracking algorithm that has at most m total branches will have an approximation ratio bigger than $c - \frac{(c-1)m}{|\Sigma|^n}$, where n is the number of data items considered.”

If $m = |\Sigma|^n$ then the approximation ratio is 1. But what happens to the approximation ratio c , when m is much smaller than Σ^n ?

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.
- [BEY98] Allan Borodin and Ran El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, 1998.
- [BF94] Piotr Berman and Martin Furer. Approximating maximum independent set in bounded degree graphs. *SODA*, January 1994.
- [BNR02] Allan Borodin, Morten N. Nielsen, and Charles Rackoff. (Incremental) Priority Algorithms. *Thirteen Annual ACM-SIAM Symposium on Discrete Algorithms*, January 2002.
- [DI02] Sashka Davis and Russell Impagliazzo. Models of greedy algorithms for graph problems. December 2002.