Textbook Greedy Problems

Sijian Tan

1 Problem 1

Decide whether you think the following statement is true or false. If it is true, give a short explanation. If it is false, give a counterexample.

Let G be an arbitrary connected, undirected graph with a distinct cost $c(e)$ on every edge e. Suppose e^* is the cheapest edge in G; that is, $c(e^*) < c(e)$ for every edge $e \neq e^*$. Then there is a minimum spanning tree T of G that contains the edge e^*.

1.1 Solution

This is true. Based on Kruskal's algorithm, e^* will be the first edge considered, so it will be included in MST.

2 Problem 2

For each of the following two statements, decide whether it is true or false. If it is true, give a short explanation. If it is false, give a counterexample.

(a) Suppose we are given an instance of the Minimum Spanning Tree Problem on a graph G, with edge costs that are all positive and distinct. Let T be a minimum spanning tree for this instance. Now suppose we replace each edge cost c_e by its square, c_e^2, thereby creating a new instance of the problem with the same graph but different costs.

True or false? T must still be a minimum spanning tree for this new instance.

(b) Suppose we are given an instance of the Shortest s-t Path Problem on a directed graph G. We assume that all edge costs are positive and distinct. Let P be a minimum-cost s-t path for this instance. Now suppose we replace each edge cost c_e by its square, c_e^2, thereby creating a new instance of the problem with the same graph but different costs.

True or false? P must still be a minimum-cost s-t path for this new instance.

2.1 Question a

True. If we feed the cost c_e^2 into Kruskal's algorithm, it will sort them in the same order, and hence put the same subset of edges in MST.

2.2 Question b

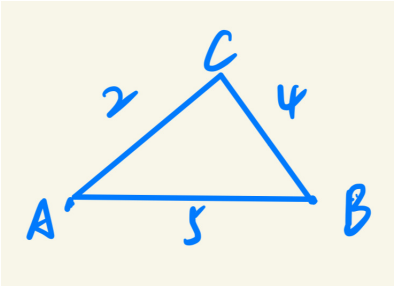

Figure 1: Problem 2

From the graph, we can see that before squaring, we will choose $[A, B]$ from A to B, after squaring, we will choose $[A, C, B]$ from A to B.

3 Problem 3

You are consulting for a trucking company that does a large amount of business shipping packages between New York and Boston. The volume is high enough that they have to send a number of trucks each day between the two locations. Trucks have a fixed limit W on the maximum amount of weight they are allowed to carry. Boxes arrive at the New York station one by one, and each package i has a weight w_i. The trucking station is quite small, so at most one truck can be at the station at any time. Company policy requires that boxes are shipped in the order they arrive; otherwise, a customer might get upset upon seeing a box that arrived after his make it to Boston faster. At the moment, the company is using a simple greedy algorithm for packing: they pack boxes in the order they arrive, and whenever the next box does not fit, they send the truck on its way.

But they wonder if they might be using too many trucks, and they want your opinion on whether the situation can be improved. Here is how they are thinking. Maybe one could decrease the number of trucks needed by sometimes sending off a truck that was less full, and in this way allow the next few trucks to be better packed.

Prove that, for a given set of boxes with specified weights, the greedy algorithm currently in use actually minimizes the number of trucks that are needed. Your proof should follow the type of analysis we used for the Interval Scheduling Problem: it should establish the optimality of this greedy packing algorithm by identifying a measure under which it "stays ahead" of all other solutions.

3.1 Solution

1. **Assume an optimal solution:** Let's assume there is an optimal solution that uses fewer trucks than the greedy algorithm. Denote the number of trucks used by the greedy algorithm as T_g and by the optimal solution as T_{opt}, with $T_{opt} < T_g$

2. **Compare the packing of the first truck:**
   Consider the first truck in the greedy algorithm, which carries the boxes $b_1, b_2, \ldots, b_k$

until the sum of the weights of these boxes is just under or exactly equal to W. That is:

$$\sum_{i=1}^{k} w_i \leq W \quad \text{and} \quad \sum_{i=1}^{k+1} w_i > W$$

In the optimal solution, the first truck can carry a different set of boxes. However, since the total weight that the first truck in the greedy algorithm carries is close to or equal to W, the first truck in the optimal solution cannot carry more than the first truck in the greedy solution.

3. **Induction on remaining trucks:**
   Now, consider the remaining trucks. By induction, we can argue that for each subsequent truck, the greedy algorithm will always pack as efficiently as possible, and since the optimal solution supposedly uses fewer trucks, there must be a point where the optimal solution packs a truck more efficiently than the greedy algorithm. However, by the nature of the greedy algorithm, which packs until it can't pack any more, this cannot happen.

4. **Contradiction:**
   The assumption that $T_{opt} < T_g$ implies that the optimal solution uses fewer trucks than the greedy solution, but this leads to a contradiction because the greedy algorithm ensures that each truck is as full as possible before moving to the next truck. Therefore, no solution can use fewer trucks than the greedy algorithm.

4 Problem 4

Some of your friends have gotten into the burgeoning field of *time-series data mining*, in which one looks for patterns in sequences of events that occur over time. Purchases at stock exchanges—what's being bought—are one source of data with a natural ordering in time. Given a long sequence S of such events, your friends want an efficient way to detect certain "patterns" in them—for example, they may want to know if the four events

```
buy Yahoo, buy eBay, buy Yahoo, buy Oracle
```

occur in this sequence S, in order but not necessarily consecutively.

They begin with a collection of possible *events* (e.g., the possible transactions) and a sequence S of n of these events. A given event may occur multiple times in S (e.g., Yahoo stock may be bought many times in a single sequence S). We will say that a sequence S' is a *subsequence* of S if there is a way to delete certain of the events from S so that the remaining events, in order, are equal to the sequence S'. So, for example, the sequence of four events above is a subsequence of the sequence

```
buy Amazon, buy Yahoo, buy eBay, buy Yahoo, buy Yahoo, buy Oracle
```

Their goal is to be able to dream up short sequences and quickly detect whether they are subsequences of S. So this is the problem they pose to you: Give an algorithm that takes two sequences of events—S' of length m and S of length n, each possibly containing an event more than once—and decides in time $O(m + n)$ whether S' is a subsequence of S.

4.1 Solution

Let the sequence S consist of $s_1, \ldots, s_n$ and the sequence S' consist of $s'_1, \ldots, s'_m$. We give a greedy algorithm that finds the first event in S that is the same as s'_1, matches these two events, then finds the first event after this that is the same as s'_2, and so on. We will use $k_1, k_2, \ldots$ to denote the match have we found so far, i to denote the current position in S, and j the current position in S'.

```
Initially i = j = 1
While i ≤ n and j ≤ m
    If s_i is the same as s'_j, then
        let k_j = i
        let i = i + 1 and j = j + 1
    otherwise let i = i + 1
EndWhile
If j = m + 1 return the subsequence found:  k_1, ..., k_m
Else return that "S' is not a subsequence of S"
```

Figure 2: Problem 4

5 Problem 5

Let's consider a long, quiet country road with houses scattered very sparsely along it. (We can picture the road as a long line segment, with an eastern endpoint and a western endpoint.) Further, let's suppose that despite the bucolic setting, the residents of all these houses are avid cell phone users. You want to place cell phone base stations at certain points along the road, so that every house is within four miles of one of the base stations.

Give an efficient algorithm that achieves this goal, using as few base stations as possible.

5.1 Greedy Algorithm

The algorithm includes the following steps:

1. Sort the houses by their position
2. Place the first base station at the position $h_1 + 4$ miles, where h_1 is the first house in the sorted list.
3. Skip all houses covered by this base station
4. Repeat the process for the next uncovered house until all houses are covered.

5.2 Proof:

5.2.1 Define Solutions:

- Let $G_1, G_2, ..., G_k$ be the positions of the base stations placed by the greedy algorithm
- Let $O_1, O_2, ..., O_m$ be the positions of the base stations in an optimal solution

We want to show that $k \leq m$, the number of stations in the greedy solution is at most the number of stations in the optimal solution.

5.2.2 Greedy Stays Ahead:

We will show that $G_i \geq O_i$ for all i, meaning that the greedy algorithm places the i^{th} base station at a position that is at least as far to the right as the i^{th} station in the optimal solution.

5.2.3 Inductive Argument

For the base case:

- Since the greedy algorithm always places the station at the furthest point possible within 4 miles of the first uncovered house, so $G_1 \geq O_1$

For the inductive step:

- Assume that for some i, the greedy algorithm has placed base stations such that $G_j \geq O_j$ for all $j \leq i$
- Now consider the $(i+1)^{th}$ base station. The greedy algorithm places G_{i+1} at the furthest point that can cover the first uncovered house after G_i. The optimal solution places O_{i+1} at some point that covers at least the same number of houses.
- Then by the inductive hypothesis, $G_i \geq O_i$. Since the greedy algorithm places the station at the furthest point, it ensures that $G_{i+1} \geq O_{i+1}$

5.2.4 Conclusion

- Since $G_i \geq O_i$ for all i, the greedy algorithm stays ahead of the optimal solution at each step.
- This implies that the greedy algorithm covers the houses at least as efficiently as the optimal solution
- Therefore, the total number of base stations used by the greedy algorithm could not be more than the number used by the optimal solution ($k \leq m$)

6 Problem 6

Your friend is working as a camp counselor, and he is in charge of organizing activities for a set of junior-high-school-age campers. One of his plans is the following mini-triathalon exercise: each contestant must swim 20 laps of a pool, then bike 10 miles, then run 3 miles. The plan is to send the contestants out in a staggered fashion, via the following rule: the contestants must use the pool one at a time. In other words, first one contestant swims the 20 laps, gets out, and starts biking. As soon as this first person is out of the pool, a second contestant begins swimming the 20 laps; as soon as he or she is out and starts biking, a third contestant begins swimming . . . and so on.)

Each contestant has a projected *swimming time* (the expected time it will take him or her to complete the 20 laps), a projected *biking time* (the expected time it will take him or her to complete the 10 miles of bicycling), and a projected *running time* (the time it will take him or her to complete the 3 miles of running). Your friend wants to decide on a *schedule* for the triathalon: an order in which to sequence the starts of the contestants. Let's say that the *completion time* of a schedule is the earliest time at which all contestants will be finished with all three legs of the triathalon, assuming they each spend exactly their projected swimming, biking, and running times on the three parts. (Again, note that participants can bike and run simultaneously, but at most one person can be in the pool at any time.) What's the best order for sending people out, if one wants the whole competition to be over as early as possible? More precisely, give an efficient algorithm that produces a schedule whose completion time is as small as possible.

6.1 Proof

Let the contestants be numbered $1, \ldots, n$, and let s_i, b_i, r_i denote the swimming, biking, and running times of contestant i. Here is an algorithm to produce a schedule: arrange the contestants in order of decreasing $b_i + r_i$, and send them out in this order. We claim that this order minimizes the completion time.

We prove this by an exchange argument. Consider any optimal solution, and suppose it does not use this order. Then the optimal solution must contain two contestants i and j so that j is sent out directly after i, but $b_i + r_i < b_j + r_j$. We will call such a pair (i, j) an *inversion*. Consider the solution obtained by swapping the orders of i and j. In this swapped schedule, j completes earlier than he/she used to. Also, in the swapped schedule, i gets out of the pool when j previously got out of the pool; but since $b_i + r_i < b_j + r_j$, i finishes sooner in the swapped schedule than j finished in the previous schedule. Hence our swapped schedule does not have a greater completion time, and so it too is optimal.

Continuing in this way, we can eliminate all inversions without increasing the completion time. At the end of this process, we will have a schedule in the order produced by our algorithm, whose completion time is no greater than that of the original optimal order we considered. Thus the order produced by our algorithm must also be optimal.

7 Problem 8

Suppose you are given a connected graph G, with edge costs that are all distinct. Prove that G has a unique minimum spanning tree.

7.1 Proof

Suppose T and T' are two distinct minimum spanning trees of G. Since T and T' have the same number of edges, but are not equal, there is some edge e' in T' but not in T. If we add e' to T, we will get a cycle C. Let e be the most expensive edge on this cycle. Then by the cycle property, e does not belong to any minimum spanning tree, contradicting the fact that it is in at least one of T or T'.

8 Problem 13

A small business—say, a photocopying service with a single large machine—faces the following scheduling problem. Each morning they get a set of jobs from customers. They want to do the jobs on their single machine in an order that keeps their customers happiest. Customer i's job will take time t_i to complete. Given a schedule (i.e., an ordering of the jobs), let C_i denote the finishing time of job i. For example, if job j is the first to be done, we would have $C_j = t_j$; and if job j is done right after job i, we would have $C_j = C_i + t_j$. Each customer i also has a given weight w_i that represents his or her importance to the business. The happiness of customer i is expected to be dependent on the finishing time of i's job. So the company decides that they want to order the jobs to minimize the weighted sum of the completion times, $\sum_{i=1}^{n} w_i C_i$.

Design an efficient algorithm to solve this problem. That is, you are given a set of n jobs with a processing time t_i and a weight w_i for each job. You want to order the jobs so as to minimize the weighted sum of the completion times, $\sum_{i=1}^{n} w_i C_i$.

Example. Suppose there are two jobs: the first takes time $t_1 = 1$ and has weight $w_1 = 10$, while the second job takes time $t_2 = 3$ and has weight $w_2 = 2$. Then doing job 1 first would yield a weighted completion time of $10 \cdot 1 + 2 \cdot 4 = 18$, while doing the second job first would yield the larger weighted completion time of $10 \cdot 4 + 2 \cdot 3 = 46$.

8.1 Algorithm

Let the jobs be numbered $1, 2, ..., n$, and let w_i, t_i and C_i denote the weight, time to complete and finishing time of job i. Arrange the jobs in order of decreasing $\frac{w_i}{t_i}$, and process the jobs in this order. We claim that this order minimizes the weighted completion time.

8.2 Proof

We prove this by an exchange argument. Consider any optimal solution, and suppose it does not use this order. Then the optimal solution must contain a pair of i and j such that j is processed directly after i, but $\frac{w_i}{t_i} < \frac{w_j}{t_j}$. We call such a pair (i, j) an inversion. Before swapping this pair, the completion time after these two jobs will be (assume the completion time before i is C_{before}):

$$C_{after} = w_i(C_{before} + t_i) + w_j(C_{before} + t_i + t_j) \tag{1}$$

Now if we swap i and j, we could get:

$$C'_{after} = w_j(C_{before} + t_j) + w_i(C_{before} + t_i + t_j) \tag{2}$$

Then after rearrangement, we could get:

$$C_{after} - C'_{after} = w_j t_i - w_i t_j \tag{3}$$

Because we know that:

$$\frac{w_i}{t_i} < \frac{w_j}{t_j} \tag{4}$$

Therefore:

$$w_i t_j < w_j t_i \tag{5}$$

$$C_{after} - C'_{after} > 0 \tag{6}$$

Therefore, this swap will actually decrease the final weighted completion time, which will not worsen the solution.

Continuing in this way, we can eliminate all inversions without increasing the weighted completion time. At the end of this process, we will have a schedule in the order produced by greedy algorithm. Thus the greedy algorithm is optimal.

9 Problem 14

You're working with a group of security consultants who are helping to monitor a large computer system. There's particular interest in keeping track of processes that are labeled "sensitive." Each such process has a designated start time and finish time, and it runs continuously between these times; the consultants have a list of the planned start and finish times of all sensitive processes that will be run that day.

As a simple first step, they've written a program called `status_check` that, when invoked, runs for a few seconds and records various pieces of logging information about all the sensitive processes running on the system at that moment. (We'll model each invocation of `status_check` as lasting for only this single point in time.) What they'd like to do is to run `status_check` as few times as possible during the day, but enough that for each sensitive process P, `status_check` is invoked at least once during the execution of process P.

Give an efficient algorithm that, given the start and finish times of all the sensitive processes, finds as small a set of times as possi- ble at which to invoke `status_check`, subject to the requirement that `status_check` is invoked at least once during each sensitive process P.

9.1 Algorithm

The algorithm includes the following steps:

1. Organize all processes in a non-decreasing order of their finish times as a sequence S
2. While some process is still not covered, insert a status_check right at the finish time of the first uncovered process in S

9.2 Proof

9.2.1 Define Solutions:

- Let $G_1, G_2, ..., G_k$ be the time that the status_check is inserted by the greedy algorithm
- Let $O_1, O_2, ..., O_m$ be the time that the status_check is inserted in an optimal solution

We want to show that $k \leq m$, the number of status_check in the greedy solution is at most the number of status_check in the optimal solution.

9.2.2 Greedy Stays Ahead

We will show that at each step i, the greedy solution has covered at least as many processes as the optimal solution. In other words, we need to prove that $G_i \geq O_i$ for all i so that to prove greedy could cover at least many processes as the optimal solution.

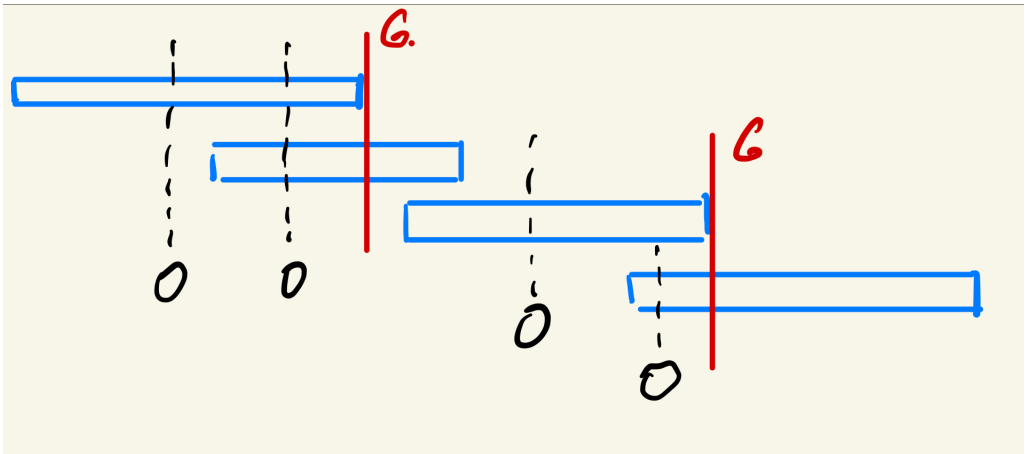

Figure 3: Problem 14

9.2.3 Inductive Argument

For the base case:

- For the first uncovered job, greedy algorithm place the status-check at the latest time that could cover this job, so $G_1 \geq O_1$

For the inductive step:

- Assume that for some i, the greedy algorithm has places the status_check such that $G_j \geq O_j$ for all $j \leq i$
- Now consider the $(i+1)^{th}$ status_check. The greedy algorithm places G_{i+1} at the latest time that can cover the first uncovered job after G_i, and the optimal solution place the status_check at some time to cover this job.
- Then by the inductive hypothesis, $G_i \geq O_i$, so $G_{i+1} \geq O_{i+1}$.

9.2.4 Conclusion

- Since $G_i \geq O_i$ for all i, the greedy algorithm stays ahead of the optimal solution at each step.
- This implies that the greedy algorithm covers the jobs at least as efficiently as the optimal solution (because it covers more time)
- Therefore, the total number of status_check used by the greedy algorithm could not be more than the number used by the optimal solution ($k \leq m$)
- Therefore, greedy is optimal.

10 Problem 15

The manager of a large student union on campus comes to you with the following problem. She's in charge of a group of n students, each of whom is scheduled to work one *shift* during the week. There are different jobs associated with these shifts (tending the main desk, helping with package delivery, rebooting cranky information kiosks, etc.), but we can view each shift as a single contiguous interval of time. There can be multiple shifts going on at once.

She's trying to choose a subset of these n students to form a *super- vising committee* that she can meet with once a week. She considers such a committee to be *complete* if, for every student not on the committee, that student's shift overlaps (at least partially) the shift of some student who is on the committee. In this way, each student's performance can be observed by at least one person who's serving on the committee.

Give an efficient algorithm that takes the schedule of n shifts and produces a complete supervising committee containing as few students as possible.

Example. Suppose $n = 3$, and the shifts are
Monday 4 p.m.–Monday 8 p.m.,
Monday 6 p.m.–Monday 10 p.m.,
Monday 9 p.m.–Monday 11 p.m.
Then the smallest complete supervising committee would consist of just the second student, since the second shift overlaps both the first and the third.

10.1 Algorithm

The algorithm steps include:

1. At all time, some intervals will be marked if they are already intersected and some will not.
2. Sort all the intervals by their finish times in ascending order.
3. Look at the unmarked interval that ends earliest, and among the intervals that intersect it, we choose the interval that ends the latest.

10.2 Proof

10.2.1 Define Solutions:

- Let $G_1, G_2, ..., G_k$ be the end time of each interval selected by the greedy algorithm
- Let $O_1, O_2, ..., O_m$ be the end time that the intervals selected in an optimal solution

We want to show that $k \leq m$, the number of committee intervals in the greedy solution is at most the number of committee intervals in the optimal solution.

10.2.2 Greedy Stays Ahead

We will show that at each step i, the greedy solution has overlapped at least as many processes as the optimal solution. In other words, we need to prove that $G_i \geq O_i$ for all i so that to prove greedy could overlap at least as many intervals.

10.2.3 Inductive Argument

For the base case:

- For the first unoverlapped interval, greedy algorithm place the intersected interval at the latest time that could overlap this job, so $G_1 \geq O_1$

For the inductive step:

- Assume that for some i, the greedy algorithm has places the intersected interval such that $G_j \geq O_j$ for all $j \leq i$
- Now consider the $(i+1)^{th}$ selected intersected interval. The greedy algorithm places G_{i+1} at the latest intersected interval that can cover the first unoverlapped interval after G_i, and the optimal solution place the intersected interval at some time to cover this interval.
- Then by the inductive hypothesis, $G_i \geq O_i$, so $G_{i+1} \geq O_{i+1}$.

10.2.4 Conclusion

- Since $G_i \geq O_i$ for all i, the greedy algorithm stays ahead of the optimal solution at each step.
- This implies that the greedy algorithm overlaps the intervals at least as efficiently as the optimal solution (because it covers more time)
- Therefore, the total number of selected intersected intervals used by the greedy algorithm could not be more than the number used by the optimal solution ($k \leq m$)
- Therefore, greedy is optimal.

11 Problem 17

Consider the following variation on the Interval Scheduling Problem. You have a processor that can operate 24 hours a day, every day. People submit requests to run *daily jobs* on the processor. Each such job comes with a *start time* and an *end time*; if the job is accepted to run on the processor, it must run continuously, every day, for the period between its start and end times. (Note that certain jobs can begin before midnight and end after midnight; this makes for a type of situation different from what we saw in the Interval Scheduling Problem.)

Given a list of n such jobs, your goal is to accept as many jobs as possible (regardless of their length), subject to the constraint that the processor can run at most one job at any given point in time. Provide an algorithm to do this with a running time that is polynomial in n. You may assume for simplicity that no two jobs have the same start or end times.

Example. Consider the following four jobs, specified by (*start-time, end- time*) pairs.

(6 p.m., 6 a.m.), (9 p.m., 4 a.m.), (3 a.m., 2 p.m.), (1 p.m., 7 p.m.).

The optimal solution would be to pick the two jobs *(9 p.m., 4 a.m.)* and *(1 p.m., 7 p.m.)*, which can be scheduled without overlapping.

11.1 Algorithm

The algorithm steps include:

1. Sort all the intervals by their finish times in ascending order.
2. For each subsequent job J_i, if its start time is after the end time of the last selected job, select it.

11.2 Proof

11.2.1 Define Solutions:

- Let $G_1, G_2, ..., G_k$ be the end time of each interval selected by the greedy algorithm

- Let $O_1, O_2, ..., O_m$ be the end time that the intervals selected in an optimal solution

We want to show that $k \geq m$, the number of intervals in the greedy solution is at least the number of intervals in the optimal solution.

11.2.2 Greedy Stays Ahead

We will show that at each step i, the greedy solution has included at least as many jobs as the optimal solution. In other words, we need to prove that $G_i \leq O_i$ for all i so that to prove greedy could include at least as many jobs than optimal.

11.2.3 Inductive Argument

For the base case:

- Among the unselected jobs, greedy algorithm select G_1 with the earliest end time, so $G_1 \leq O_1$

For the inductive step:

- Assume that for some i, the greedy algorithm has selected the jobs such that $G_j \leq O_j$ for all $j \leq i$
- Now consider the $(i+1)^{th}$ selected jobs. The greedy algorithm selects G_{i+1} with the earliest end time after G_i, and the optimal solution selects the job at some other time.
- Then by the inductive hypothesis, $G_i \leq O_i$, so $G_{i+1} \leq O_{i+1}$.

11.2.4 Conclusion

- Since $G_i \leq O_i$ for all i, the greedy algorithm stays ahead of the optimal solution at each step.
- This implies that the greedy algorithm selects the jobs at least as the optimal solution (because it selects faster)
- Therefore, the total number of selected intersected jobs used by the greedy algorithm could not be less than the number used by the optimal solution ($k \geq m$)
- Therefore, greedy is optimal.

12 Problem 18

Your friends are planning an expedition to a small town deep in the Cana- dian north next winter break. They've researched all the travel options and have drawn up a directed graph whose nodes represent intermediate destinations and edges represent the roads between them.

In the course of this, they've also learned that extreme weather causes roads in this part of the world to become quite slow in the winter and may cause large travel delays.

They've found an excellent travel Web site that can accurately predict how fast they'll be able to travel along the roads; however, the speed of travel depends on the time of year. More precisely, the Web site answers queries of the following form: given an edge $e = (v, w)$ connecting two sites v and w, and given a proposed starting time t from location v, the site will return a value $f_e(t)$, the predicted arrival time at w. The Web site guarantees that $f_e(t) \geq t$ for all edges e and all times t (you can't travel backward in time), and that $f_e(t)$ is a monotone increasing function of t (that is, you do not arrive earlier by starting later). Other than that, the functions $f_e(t)$ may be arbitrary. For example, in areas where the travel time does not vary with the season, we would have $f_e(t) = t + \ell_e$, where ℓ_e is the time needed to travel from the beginning to the end of edge e.

Your friends want to use the Web site to determine the fastest way to travel through the directed graph from their starting point to their intended destination. (You should assume that they start at time 0, and that all predictions made by the Web site are completely correct.) Give a polynomial-time algorithm to do this, where we treat a single query to the Web site (based on a specific edge e and a time t) as taking a single computational step.

12.1 Algorithm

From the start node, always choose the next node as the node that could be achieved in the shortest amount of time. (Dijkstra)

12.2 Proof

Using greedy algorithm to prove Dijkstra.