CSE 3101 Design and Analysis of Algorithms
Solutions for Practice Test for Unit 4
Greedy Algorithms

Jeff Edmonds

1. Integer-Knapsack Problem:

   Instances: An instance consists of $\langle V, \langle v_1, p_1 \rangle, \ldots, \langle v_n, p_n \rangle \rangle$. Here, V is the total volume of the knapsack. There are n objects in a store. The volume of the i^{th} object is v_i, and its price is p_i.

   Solutions: A solution is a subset $S \subseteq [1..n]$ of the objects that fit into the knapsack, i.e., $\sum_{i \in S} v_i \leq V$.

   Measure Of Success: The cost (or success) of a solution S is the total value of what is put in the knapsack, i.e., $\sum_{i \in S} p_i$.

   Goal: Given a set of objects and the size of the knapsack, the goal is fill the knapsack with the greatest possible total price.

   Failed Greedy Algorithms: Three obvious greedy algorithms take first the most valuable object, $\max_i p_i$, the smallest object, $\min_i v_i$, or the object with the highest value per volume, $\max_i \frac{p_i}{v_i}$. However, they do not work. For each of this greedy algorithms, give an instance on which it does not work.

   - Answer: Consider the counter example $\langle 8, \langle 5, 5 \rangle, \langle 4, 3 \rangle, \langle 4, 3 \rangle, \langle 1, 0 \rangle \rangle$. If one takes the highest value or highest density, then one takes the first object. If one takes the smallest volume, then one takes the last object. The only optimal solution contains the middle two objects.

2. We proved that the greedy algorithm does not work for the making change problem when the denominations of the coins are 4, 3, and 1 cent, but it does work when the denominations are 25, 10, 5, and 1. Does it work when the denominations are 25, 10, and 1, with no nickels?

   - Answer: The greedy algorithm does not work for the making change problem, when the denominations of the coins are 25, 10, and 1, with no nickels. Suppose the amount is 30. The greedy algorithm will take on quarter and five pennies. However, the optimal has only three dimes.

3. Greedy Algorithms: A man is standing on the bank of a river that he must cross by jumping on stepping stones which are spread in a line across the river. Though he can't jump far, he wants to jump on as few stones as possible. An *instance* to the problem specifies $\langle d_0, d_1, d_2, \ldots, d_n \rangle$, where $d_0 = 0$ is the man's initial location, d_i for $i \in [1, .., n-1]$ is a real number giving the distance in meters that the i^{th} stone is from him, and d_n is the distance to the opposite shore. Assume that the stones are in order, i.e. $0 = d_0 \leq d_1 \leq d_2 \leq \ldots \leq d_n$. Also assume that the stones are not more than one meter apart, i.e. $\forall i \; d_{i+1} - d_i \leq 1$.

   A *solution* is a sequence $\langle i_0, i_1, i_2, \ldots i_\ell \rangle$ of the indexes of the stones indicating that at time t the man jumps onto the ${i_t}^{th}$ stone. Such a solution is *valid* if at time $t = 0$ the man is in fact on the close shore, i.e. $i_0 = 0$, he finishes on the far shore, i.e. $i_\ell = n$, and at each time step t' the distance he jumps is at most a distance of one meter, i.e. $d_{i_{t'}} - d_{i_{t'-1}} \leq 1$. The *cost* of a solution is the number ℓ of stones jumped onto.

   Note that the assumption on the input ensures that a valid solution always exists.
   Specify a greedy algorithm for finding an optimal solution for this problem. Imagine that you are crossing a river on stones. How would you choose which stone to jump on next.

   As done in the steps, prove that your algorithm always returns an optimal solution.
   Warning: To ensure that a solution is valid, you really must check the distance jumped at *each* time step t'.

   - Answer:

The Greedy Choice: Each iteration the algorithm grabs the object which according to some simple criteria, seems to be the "best" (or "worst") from amongst the unconsidered objects in the instance. **Suppose that the current time is $t-1$ and the last stone jumped on is the ${i_{t-1}}^{th}$ stone. In a greedy way, have the man jump on the stone that is as far from i_{t-1} as he can jump, i.e. the largest i_t for which $d_{i_t} - d_{i_{t-1}} \leq 1$. This is an adaptive decision, i.e. it does depend on what objects have been seen already.**

You can think of any stone that is before stone i_{t-1} as having greedy criteria weight $-\infty$ because we don't want to jump backwards and any stone that is more than one meter ahead of stone i_{t-1} also has the greedy criteria weight $-\infty$ because we don't want to land in the water and each other stone j has greedy criteria $d_j - d_{i_{t-1}}$. Then you let i_t be the stone j that maximizes this greedy criteria. The algorithm then makes an irrevocable decision about this object. **Assuming such a stone exists, the algorithm has the man jump on it. Otherwise, the algorithm stops and states that it is impossible.**

The Loop Invariant: We have not gone wrong. If there is a solution, then there is at least one optimal solution S_t extending the choices A_t made so far by the algorithm.

Initially ($\langle pre \rangle \rightarrow \langle LI \rangle$): Initially no choices have been made and hence all optimal solutions are consistent with these choices.

Maintaining the Loop Invariant ($\langle LI_{t-1} \rangle$ & not $\langle exit \rangle$ & $code_{loop} \rightarrow \langle LI_t \rangle$): Consider an arbitrary iteration.

Fly in From Mars: The algorithm has iterated a few times. **Assume that it is time** $t-1$ **and that the algorithm has already decided for the man to step on the stones indexed by $\langle i_0, i_1, i_2, \ldots, i_{t-1} \rangle$.** All that we know is that the loop invariant is true and the exit condition is not.

S_{t-1}: The loop invariant states that there is at least one optimal solution extends the choices A_{t-1} made by the algorithm before this iteration. **Let $S_{t-1} = \langle i_0, i_1, i_2, \ldots, i_{t-1}, j_t, j_{t+1}, j_{t+2}, \ldots, j_\ell \rangle$** denote one such solution. (Note that this is a full solution and as such specifies a decision about each object in the instance and that the first t stones do agree with what the algorithm has done.) I like to have a fairy god mother hold it.

Taking a Step: During the iteration, the algorithm proceeds to choose the "best" object from amongst those not considered so far and makes an irrevocable decision about it. **In a greedy way, the algorithm just had the man jump on the stone that is as far from i_{t-1} as he can jump, i.e. the largest i_t for which $d_{i_t} - d_{i_{t-1}} \leq 1$. There is only this case, because we are assuming that such a stone exists.**

Instructions for Modifying S_{t-1}: The prover says to the fairy god mother, "Fairy god mother, I know that you have your full sequence of stones across the river planned out and that like the algorithm, at time $t-1$, you are standing on the ${i_{t-1}}^{th}$ stone. If from there you jump to the same stone as the algorithm did, i.e. $j_t = i_t$, then no changes need to be made to your solution, i.e. $S_t = S_{t-1}$." Otherwise, because the algorithm stepped as far as possible, she must have stepped closer, i.e. her stone j_t must be between the stones i_{t-1} and i_t. (If she jumped on more than this one stone between i_{t-1} and i_t then her solution could not have been optimal.) Tell her to jump from stone i_{t-1} not to j_t but to go farther to i_t. From i_t, she should continue on as she did before, i.e. $S_t = S_{t-1} - j_t + i_t = \langle i_0, i_1, i_2, \ldots, i_{t-1}, i_t, j_{t+1}, j_{t+2}, \ldots, j_\ell \rangle$.

ProvingS_t is an Optimal Solution: By the loop invariant, S_{t-1} is one of the valid solutions for this instance with minimum number of stones. We must prove that going from S_{t-1} to S_t does not increase the number of stones. We added one stone so we must make sure that we also delete at least one. To prove this it is sufficient to prove that the next stone j_t that she jumps on after i_{t-1} is

to the left of stone i_t. We know that the algorithm chose to jump from stone i_{t-1} to stone i_t because i_t is as the farthest from the initial shore that can be jumped on from i_{t-1}. We also know that the fairy god mother legally jumps from i_{t-1} to j_t. We conclude that j_t is either the same as stone i_t or is closer to stone i_{t-1} than it. It follows that at least one stone j_t is included in the sequence deleted. Hence S_t is at least as good as S_{t-1}. Hence, S_t is also an optimal solution.

ProvingS_t is a Valid Solution: By the loop invariant S_{t-1} is a valid solution. (i.e. the initial stone is on the initial shore, the final stone is on the final shore, and each jump has distance at most one.) The prover made sure that his modifications did not make it invalid. We have $S_t = \langle i_0, i_1, i_2, \ldots, i_{t-1}, i_t, j_{t+1}, j_{t+2}, \ldots, j_\ell \rangle$. We know all the jumps from $i_{t'}$ to $i_{t'+1}$ for $t' \in [0, t-2]$, have distance at most one because both the algorithm and the fairy god mother in S_{t-1} have assured it. We know the jump from i_{t-1} to i_t has distance at most one because the algorithm chose it this way. We know the jump from i_t to j_{t+1} has distance at most the distance from j_t to j_{t+1} because we proved that j_t is to the left of i_t, and we know that this distance is at most one because the fairy god mother manages to jump this in S_{t-1}. Finally, we know all the jumps from $j_{t'}$ to $j_{t'+1}$ for $t' \geq t + r$ have distance at most one because the fairy god mother in S_{t-1} has assured it. In conclusion, S_t is a valid solution.

Proving S_t Extends A_t: By the loop invariant S_{t-1} extends all the decisions $\langle i_0, i_1, i_2, \ldots, i_{t-1} \rangle$ made by algorithm before this iteration. The prover made sure that his modifications did not change any of these decisions and changed S_{t-1}'s decision j_t about the latest object to be consistent with what the algorithm did, i.e. i_{t+1}. Hence, S_t is consistent with both the earlier decisions made by algorithm and this most recent decision, i.e. extends A_t.

$\rightarrow$ **$\langle LI_t \rangle$:** Because S_t witnesses the fact that there is at least one valid optimal solution that extends A_t,, we know that the loop invariant has been maintained.

Exiting Loop ($\langle LI \rangle$ & $\langle exit \rangle \rightarrow \langle post \rangle$): By the exit condition the algorithm has made a decision about every object in the instance. Hence, the algorithm has a full solution A_{exit}. By the loop invariant, this extends an optimal valid solution S_{exit}. Hence, the algorithm must have an optimal valid solution.

4. Job/event scheduling problem with multiple rooms:
Review the job/event scheduling problem from Section 16.2.1. This problem is the same except you have r rooms/processors within which to schedule that the set of jobs/events. An instance is $\langle r, \langle s_1, f_1 \rangle, \langle s_2, f_2 \rangle, \ldots, \langle s_n, f_n \rangle \rangle$, where as before $0 \leq s_i \leq f_i$ are the starting and finishing times for the i^{th} event. But now the input also specifies the number of rooms r. A solution for an instance is a schedule $S = \langle S_1, \ldots, S_r \rangle$ for each of the rooms. Each of these consists of a subset $S_j \subseteq [1..n]$ of the events that don't conflict by overlapping in time. The success of a solution S is the number of events scheduled, that is, $|\cup_{j\in[r]} S|$. Consider the following four algorithms:

   (a) Find the greedy solution for the first room. Then find the greedy solution for the second from the remaining events. Then the third room. And so on.

   (b) It starts by sorting the events by their finishing times, just like in the one-room case. Then, it looks at each event in turn, scheduling it, if possible. If it can be scheduled in more than one room, we assign it in the first room at which it fits. I.e./ first try room one, then room two, and so on until it fits. If it cannot be scheduled in any room, then it is not scheduled.

   (c) Same except, the next event is scheduled in the room with the latest last-scheduled finishing time. For example, suppose the last event scheduled in rooms 1, 2, and 3 finishes at times 10, 15, and 18, and the next event starts at time 17. Then the next event could be scheduled into either room 1 or 2 but not in 3. This algorithm would schedule it in room 2. Note this is the room that minimizes the gap between starting time of the new job and the finishing time of the previously scheduled job for the room. Here a gap of size $17 - 15 = 2$ is better than one of size $17 - 10 = 7$ or of size $17 - 18 = -1$.

(d) Same except, the next event is scheduled in the room with the earliest last-scheduled finishing time. Note this is the room that maximizes the said mentioned gap.

Prove that three of these algorithms do not lead to an optimal schedule and that the remaining one does.

- Answer: Algorithms (a), (b), and (d) are sub-optimal for the following counterexample instance.

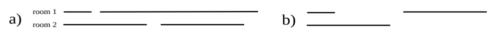


Fig a gives the events in the instance and the optimal schedule in two rooms. Fig b gives the suboptimal schedule produced by these three algorithms. Note that the third algorithm, which schedules the next event in the schedulable room with the latest last-scheduled finishing time, gives the optimal schedule. We will now prove that it always gives an optimal solution.

Specifications: The objects are the events. For each, we must either choose a room for it to be scheduled in or choose not to schedule it. Such a sequence of decisions makes a valid solution if no two events scheduled in the same room overlap. The measure according to which the solution is optimal is the total number of events scheduled.

The Greedy Choice: Each iteration the algorithm grabs the object which according to some simple criteria, seems to be the "best" (or "worst") from amongst the unconsidered objects in the instance. This algorithm considers the events in the *fixed* order sorted by their finishing times. Lets denote this next event by Obj_t or by more simply by i. The algorithm must make an irrevocable decision about which room it will be scheduled into. When considering the next event, the algorithm considers the set of rooms in which this event can be scheduled in. Of those, it is scheduled in the room with the latest last-scheduled finishing time. If there are no such rooms, then the event is not scheduled.

The Loop Invariant: We have not gone wrong. If there is a solution, then there is at least one optimal solution S_t that extends the choices A_t made so far by the algorithm.

Initially ($\langle pre \rangle \rightarrow \langle LI \rangle$): Initially no choices have been made and hence all optimal solutions extend choices A_0.

Maintaining the Loop Invariant ($\langle LI_{t-1} \rangle$ & not $\langle exit \rangle$ & $code_{loop} \rightarrow \langle LI_t \rangle$): Consider an arbitrary iteration.

S_{t-1}: The loop invariant states that there is at least one optimal solution that extends the choices A_{t-1} made by the algorithm before this iteration. Let S_{t-1} denote one such solution. (Note that this is a full solution and as such specifies a decision about each object in the instance.) If you like have a fairy god mother hold it.

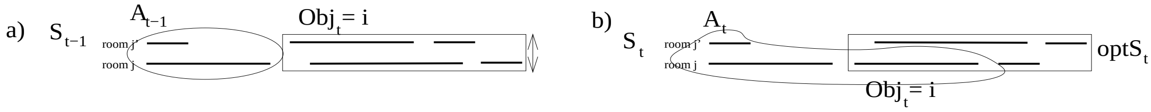
Second Case: If our greedy algorithm did not schedule the next event i, then this event must conflict in each room with a previously scheduled event. Hence, S_{t-1} cannot have this next event i scheduled either because it too has scheduled these previous events. Hence, S_{t-1} itself is already consistent this with the most recent choice.

Taking a Step: Now assume that our greedy algorithm scheduled the next event i in room j.

Instructions for Modifying S_{t-1}: There are three cases. If both our greedy algorithm and S_{t-1} scheduled the next event i in room j then no changes need to be made and we are done.

If our greedy algorithm scheduled the next event i in room j and S_{t-1} does not schedule this event at all, then we modifying the schedule S_{t-1} into S_t by adding i to room j and removing any events from j that conflict with it. Just as done with the one room scheduling algorithm in Section 16.2.1, we can prove that only one event is removed and hence S_t is valid, is optimal, and extends A_t.

The remaining case occurs when our greedy algorithm scheduled the next event i in room j and S_{t-1} schedules it in room j'. (See fig a.)



We modify the schedule S_{t-1} into S_t as follows. (See fig b.) We cannot move the events in A_{t-1} whose schedule has already been committed to by the algorithm because S_t must continue to extend these choices. (See the events in the circle.) We need to move event i from room j' to room j so that it too is consistent with what the algorithm has done. But doing this change may create conflicts, i.e. overlapping events. We fix any newly created overlaps as follows. Put in a rectangle every non-A_{t-1} event scheduled in room j or j'. Note these all have finishing times of event i or later. We swap the room j rectangle events with the room j' ones.

We now prove that the resulting solution S_t is valid, extends A_t, and is optimal.

Proving S_t is a Valid Solution: By the loop invariant S_{t-1} is a valid solution, i.e. events in the same room do not overlap in time. The prover made sure that his modifications did not make events overlap. The changes may have made conflicts (events overlap), but all of these were fixed. Hence, S_t is a valid solution. More specifically, there are no new overlaps between events in A_{t-1} (circle) because they did not change. There are no new overlaps between the non-A_{t-1} (rectangle) events because they flipped rooms all together. Hence, we only need to worry about the boundary between the A_{t-1} and the non-A_{t-1} events. The non-A_{t-1} events that flipped from room j' to j don't now overlap with the A_{t-1} events in room j, because they are led by event i and the algorithm scheduled it there. The non-A_{t-1} events that flipped from room j to j' don't now overlap with the A_{t-1} events in room j', because they did not conflict with those in room j and we know by the algorithm's choice of room j that the last A_{t-1} event's finishing time for j is later than that for room j'.

Proving S_t Extends Algorithm's Choices A_t: By the loop invariant S_{t-1} extends all the decisions A_{t-1} made by algorithm before this iteration. The prover made sure that his modifications did not change any of these decisions and changed S_{t-1}'s decision about the latest event i to be consistent with what the algorithm did. We did not move any events in A_{t-1}. We moved event i from room j' to room j to make S_t consistent with this most recent choice. Hence, S_t is consistency both with earlier decisions A_{t-1} made by algorithm and this most recent decision i and hence extends A_t.

ProvingS_t is an Optimal Solution: By the loop invariant S_{t-1} is one of the valid solutions for this instance with optimal (minimum or maximum as the case may be) measure of success. The prover made sure that his modifications did not make S_t worse than S_{t-1}. The schedule S_t has the same number of events as S_{t-1}. Hence, S_{t-1} is an optimal solution.

$\rightarrow$ **$\langle LI' \rangle$:** Because S_t witnesses the fact that there is at least one optimal solution that extends A_t, we know that the loop invariant has been maintained.

Exiting Loop ($\langle LI \rangle$ & $\langle exit \rangle \rightarrow \langle post \rangle$): By the exit condition the algorithm has made a decision about every object in the instance. Hence, the algorithm has a full solution. By the loop invariant, this extends an optimal valid solution. Hence, the algorithm must have an optimal valid solution.