CS310: Advanced Data Structures and Algorithms

Fall 2022 Assignment 4

Due: Thursday, November 26, 2022 on Gradescope

Goals

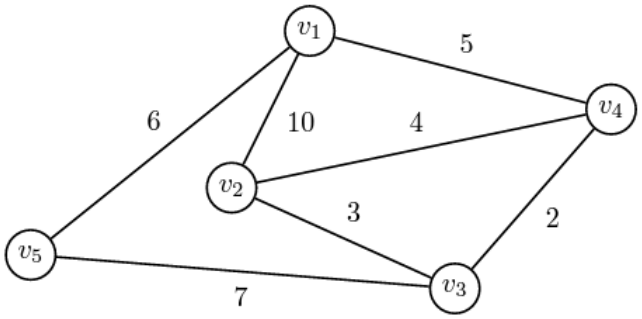
Greedy algorithms (graphs and others), Dynamic Programming

Reading

K&T, Chapter 4 (greedy algorithms), Chapter 6 (dynamic programming) S&W, Chapter 4.4 (MSTs)

Questions

1. Given the following weighted, undirected graph:



   (a) Show the process of Kruskal's algorithm to find its minimal spanning tree. The format of your answer should be the following: write down the edges in the order in which they are processed, and indicate for each edge whether it appears in the final MST or not.

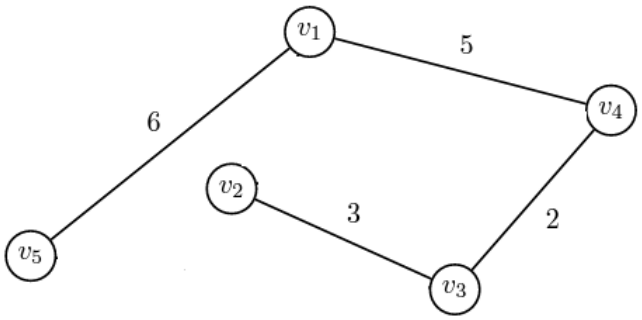
   $v_3 \to v_4$
   $v_2 \to v_3$
   $v_2 \to v_4$ (not a tree edge, closes a cycle)
   $v_1 \to v_4$
   $v_1 \to v_5$

   All other edges are not tree edges.

   (b) Do the same with Prim's algorithm. Start from v_1.

   $v_1 \to v_4$
   $v_3 \to v_4$
   $v_2 \to v_3$
   $v_1 \to v_5$

   (c) Draw the final MST (despite possibly selecting the edges in a different order, the MST should be the same for (a) and (b)!).

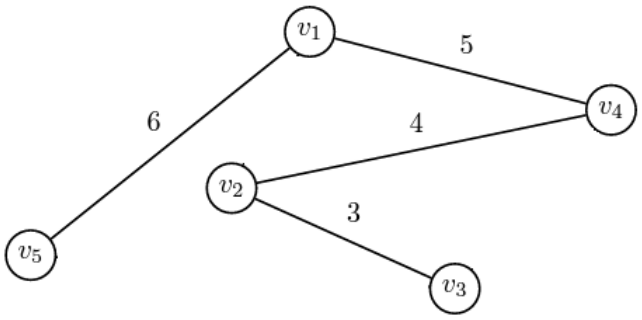


2. (Adapted from K&T, 4.9): One of the basic motivations behind the Minimum Spanning Tree Problem is the goal of designing a spanning network for a set of nodes with minimum total cost. Here we explore another type of objective: designing a spanning network for which the most expensive edge is as cheap as possible. Specifically, let $G = (V, E)$ be a connected graph with n vertices, m edges, and positive edge costs that you may assume are all distinct. Let $T = (V, E)$ be a spanning tree of G; we define the *bottleneck edge* of T to be the edge of T with the greatest cost. A spanning tree T of G is a minimum-bottleneck spanning tree (MBST) if there is no spanning tree T' of G with a cheaper bottleneck edge.

   (a) Show that every minimum spanning tree of G is also a minimum-bottleneck tree of G (it's easiest to prove by contradiction IMO).

   Let us define an MST T, which is not equal to an MBST. In this case, there is a tree T' whose bottleneck edge $e' = (v', w')$ is lighter than the bottleneck edge $e = (v, w)$ of the MST T. Let us define a cut in the graph such that v and v' are on one side and w and w' are on the other (the other vertices can be assigned at random). Both e and e' cross the cut, and obviously e, the edge selected for the MST, is not lightest edge that crosses the cut (e' is lighter than it). But we showed previously that an MST always contains a lightest edge that crosses any cut, contradicting the fact that T is an MST.

   (b) The opposite is not always true. Show an example of a minimum-bottleneck tree of G which is not a minimum spanning tree of G. (**Hint:** Look at the example from HW3 and play with it a little).



3. Adapted from K&T, 4.6: 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-triathlon 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 triathlon: 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 triathlon, 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 is 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 *greedy* algorithm that produces a schedule whose completion time is as small as possible. You should show why your ordering is optimal. Use an exchange argument.
Hint: Notice that whatever order you pick, the overall pool usage time is the same since only one student can use it at any time. The difference in the overall finish time is therefore determined by the combined bike+run time.

First focusing on the hint, we can say that if we have n participants with each one $i = 1 \dots n$ and their pool time p_i, then the overall pool time, regardless of the order of students that use it (only 1 can use it at a time) is:

$$\sum_{i=1}^{n} p_i$$

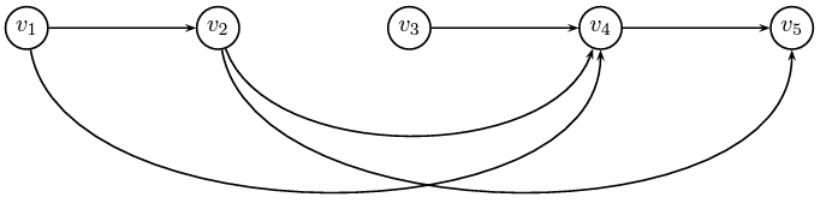
This means that the only difference in overall completion time is the time spent outside the pool. Therefore, we can define a greedy algorithm for this problem as one that sorts the students' combined run and bike times in order least to greatest.

To prove this is the most efficient solution, we will assume that this solution holds. If there is an adjacent pair of students i and j where i's run and bike time is less than j's. If i were to enter the pool before j, then the total time would be the sum of the two students' swim times and student j's run and bike time. If we were to send the students in the pool in the reverse order, the total time would be either the sum of the two swim times and student i's run and bike time, or the total time for student j. This proves that it is more efficient to send the students in order of run and bike time.

4. (Adapted from K&T 6.3) Here is a suggested greedy algorithm to find the *longest* path in a DAG (directed acyclic graph):

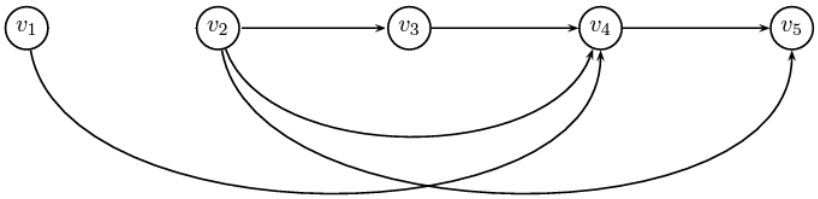
    1. Let $w = v_1$
    2. Let $L = 0$ (the length of the longest path so far)
    3. While there is an edge out of w:
        i. Choose an edge (w, v_j) such that j is minimum (in the example below when v_1 is considered it would be v_2, since 2 is the minimum out of v_2 and v_4, the outgoing neighbors of v_1).
        ii. $w \leftarrow v_j$
        iii. $L \leftarrow L + 1$
    4. Return L

    (a) Show that the algorithm above indeed gives the longest path in the DAG below:



The path returned by the algorithm is $v_1 \to v_2 \to v_4 \to v_5$, which is the longest path.

(b) Slightly modify the graph in (a) such that this algorithm no longer gives the longest path (you may delete or add edges, as long as the graph is still a DAG and still connected). Explain briefly why the algorithm above doesn't work on your example.



The path using the previous algorithm is $v_1 \to v_4 \to v_5$ while the longest path is now $v_2 \to v_3 \to v_4 \to v_5$.

(c) It is possible to find the longest path in a DAG using dynamic programming. This is done by calculating, for every vertex v_i, the longest path that ends in v_i. Notice that any vertex v_i extends the longest paths ending at each one of its incoming neighbors by 1 (this is true only in DAGs, of course, due to the fact that paths only go one way). Therefore, the length of the longest path ending at v_i is the length of the longest path of all of v_i's predecessors + 1. The algorithm is given below:

i. Topologically sort the graph

ii. For each v_i whose in-degree is 0, set $LP(v_i) = 0$ (this is the longest path ending at v_i).

iii. For each of the other vertices w_i, in the topologically sorted order:
set $LP(w_i) = \max_{v_j s.t (v_j, v_i) \in E} LP(v_j) + 1$

iv. Return $max_{v_i \in V}(LP(v_i))$

For each vertex in the DAG shown in (a) above, fill the length of the longest path ending in it.

v_i	$LP(v_i)$
v_1	0
v_2	1
v_3	0
v_4	2
v_5	3

(d) What is the run time of the algorithm in (c) above as a function of $|V|$ (the number of vertices) and/or $|E|$ (the number of edges)? Explain briefly.

$O(|V| + |E|)$ for both the topological sorting and the longest path search. The inner loop is repeated a number of times equal to the number of edges.

5. (Adapted from K&T 6.15): On most clear days, a group of your friends in the Astronomy Department gets together to plan out the astronomical events they're going to try observing that night. We'll make the following assumptions about the events:

- There are n events, which for simplicity we'll assume occur in sequence separated by exactly one minute each. Thus event j occurs at minute j; if they don't observe this event at exactly minute j, then they miss out on it.

- The sky is mapped according to a one-dimensional coordinate system (measured in degrees from some central baseline); event j will be taking place at coordinate d_j, for some integer value d_j. The telescope starts at coordinate 0 at minute 0.
- The last event, n, is much more important than the others; so it is required that they observe event n.

The Astronomy Department operates a large telescope that can be used for viewing these events. Because it is such a complex instrument, it can only move at a rate of one degree per minute. Thus they do not expect to be able to observe all n events; they just want to observe as many as possible, limited by the operation of the telescope and the requirement that event n must be observed. We say that a subset S of the events is viewable if it is possible to observe each event $j \in S$ at its appointed time j, and the telescope has adequate time (moving at its maximum of one degree per minute) to move between consecutive events in S. Notice that you can move either forward or backwards.

The problem: Given the coordinates of each of the n events, find a viewable subset of maximum size, subject to the requirement that it should contain event n. Such a solution will be called optimal.

Example: Suppose the one-dimensional coordinates of the events are as shown here.

Event	1	2	3	4	5	6	7	8	9
Coordinate	1	-4	-1	4	5	-4	6	7	-2

Then the optimal solution is to observe events 1, 3, 6, 9. Note that the telescope has time to move from one event in this set to the next, even moving at one degree per minute.

(a) Show that the following greedy algorithm **does not** always correctly solve this problem, by giving an instance on which it does not return the correct answer. Show an example where the correct answer is not returned, and say what it should be. (Notice that it actually returns the correct answer in the example above, but show an example where it doesn't. You can start from the example above and modify it a little).

- Mark all events j with $|d_n - d_j| > n - j$ as illegal (as observing them would prevent you from observing event n)
- Mark all other events as legal
- Initialize current position to coordinate 0 at minute 0
- While not at end of event sequence:
  - Find the earliest legal event j that can be reached without exceeding the maximum movement rate of the telescope
  - Add j to the set S
  - Update current position to be coord. d_j at minute j
- Output the set S

In order to prove that the greedy algorithm isn't always the best solution, we can create a case where an earlier legal event overshadows a later, bigger set of events.

Event	1	2	3	4	5	6	7	8	9
Coordinate	1	0	3	4	5	-4	6	7	4

The solution for the greedy algorithm would be 1,2,9, but the optimal solution is 1,2,3,4,5,7,9.

(b) Give a dynamic programming algorithm that takes values for the coordinates $d_1, d_2, \ldots d_n$ of the events and returns the *size* of an optimal solution.
Hint: Notice that the notion of legal events is still needed here, and it can be applied to any two events i and j such that $i < j$. In other words, if $|d_j - d_i| > j - i$, you can't reach event j after event i.
Another hint: First think of a recursive solution to the algorithm (that is– you know you have to include event n, so you should find the optimal solution for the *legal* events before n and attach n to it. Recursively, this applies to the rest of the events. Flesh it out.

The dynamic programming algorithm is:

- Initialize L_j to 0 for all j
- For each j from 1 to n:
  - If j is illegal with respect to n, continue
  - Select the $i < j$ that is legal with respect to j and maximizes $L(i)$
  - Set $L(j)$ to max $L(i) + 1$
- Return $L(n)$