Week 6 Applied Sheet

(Solutions)

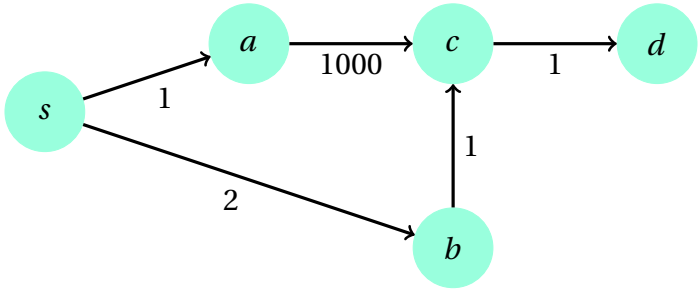
Useful advice: The following solutions pertain to the theoretical problems given in the applied classes. You are strongly advised to attempt the problems thoroughly before looking at these solutions. Simply reading the solutions without thinking about the problems will rob you of the practice required to be able to solve complicated problems on your own. You will perform poorly on the exam if you simply attempt to memorise solutions to these problems. Thinking about a problem, even if you do not solve it will greatly increase your understanding of the underlying concepts. Solutions are typically not provided for Python implementation questions. In some cases, pseudocode may be provided where it illustrates a particular useful concept.

Problems

Problem 1. Consider a buggy implementation of Dijkstra's algorithm in which the distance estimate to a vertex is only updated the first time the vertex is discovered, and not in any subsequent relaxations. Give a graph on which this bug will cause the algorithm to not produce the correct answer.

Solution

To break such a bug, we need a vertex whose initial distance estimate is very high, but should be subsequently lowered by a better path found later. The following graph will cause this problem.



The distance estimate to c will get locked at 1001, even though it should be updated to 3, and hence the distances to c and d will be wrong.

Problem 2. Can Prim's and Kruskal's algorithms be applied on a graph with negative weights to compute a minimum spanning tree? Why or why not?

Solution

Yes, these algorithms can be used to compute a minimum spanning tree on a graph with negative weights. These algorithm greedily select edges with the smallest weights and are not affected by whether the weights are positive or negative. This is different from Dijkstra's algorithm which accumulates the edge weights to obtain a distance where a negative weight may result in the total distance being reduced. Also, the proofs of correctness for Prim's and Kruskal's algorithm do not assume non-negative weights (whereas the correctness of Dijkstra's algorithm relies on the edge weights being non-negative).

Another way to look at this is to add a constant weight to each edge in the graph to make each edge weight a non-negative value. Prim's and Kruskal's algorithms can be applied on this modified graph and the resulting MST will contain the same edges as the MST in the original graph.

Problem 3. Consider the problem of planning a cross-country road trip. You have a map of Australia consisting of the locations of towns, each of which has a petrol station, with the corresponding petrol prices (which may be

different at each town). You are currently at a particular town s and would like to travel to town t. Your car has a fuel capacity of C litres, and for each road on the map, you know the amount of petrol it will take to travel along it. Your tank can only contain non-negative integer amounts of petrol, and all roads cost an integer amount of petrol to travel along. You cannot travel along a road if you do not have enough petrol to make it all the way. You may refuel at any petrol station whether your tank is empty or not (but only to integer values), and you are not required to fill your tank. Assuming that your tank is initially empty, describe an algorithm for determining the cheapest way to travel to city t. [Hint: You should model the problem as a shortest path problem and use Dijkstra's algorithm.]

Solution

Our first thought might be to weight the edges of the graph by how much petrol they use and then run Dijkstra's algorithm, but this will miss quite a few things. Remember that we need to take into account:

- The cost of petrol at various towns.
- We can fuel up partially at any town, and leave the town with any amount of petrol.
- We might not be able to make it across a road without running out.

In order to account for these things, we create a new graph that encapsulates more information than just our location on the vertices. Suppose that there are n towns. Let's make a graph that contains $(C+1)n$ vertices, one for each possible combination of town and amount of petrol that we could possible have. That is, each town has $(C+1)$ different vertices, one for when we are in that town with no petrol, one for when we are there with 1 litre, 2 litres and so on. Let's use the notation $\langle u, c\rangle$ to denote the vertex corresponding to town u where we have c litres of petrol currently in the tank.

If a road from town u to town v takes x litres of petrol to travel, then for each of the vertices $\langle u, c\rangle$ corresponding to town u with $c \geq x$, we create an edge to the corresponding vertex $\langle v, c-x\rangle$ of town v. For example, if $x = 5$, then we create an edge from $\langle u, 10\rangle$ to $\langle v, 5\rangle$, and from $\langle u, 9\rangle$ to $\langle v, 4\rangle$, from $\langle u, 8\rangle$ to $\langle v, 3\rangle$, and so on. Importantly, these edges have weight zero, since we are going to use edge weights to model the cost of purchasing fuel.

To account for buying new fuel, suppose we are currently in the town u whose petrol price is p. Then we should add an edge from every vertex $\langle u, c\rangle$ with $c < C$ to the vertex $\langle u, c+1\rangle$ with weight p. Note that we could have added edges from $\langle u, c\rangle$ to $\langle u, c+2\rangle$ with weight $2p$ and so on, but this would just make the graph unnecessarily denser and hence make the algorithm slower for no reason, so it is best to simply add edges that add 1 litre of petrol at a time.

We can then run Dijkstra from the vertex $\langle s, 0\rangle$ to find the shortest path to $\langle t, 0\rangle$, which will be the cheapest way to make it from s to t.

Problem 4. Consider a variant of the shortest path problem where instead of finding paths that minimise the total weight of all edges, we instead wish to minimise the weight of the largest edge appearing on the path. Let's refer to such a path as a *bottleneck path*.

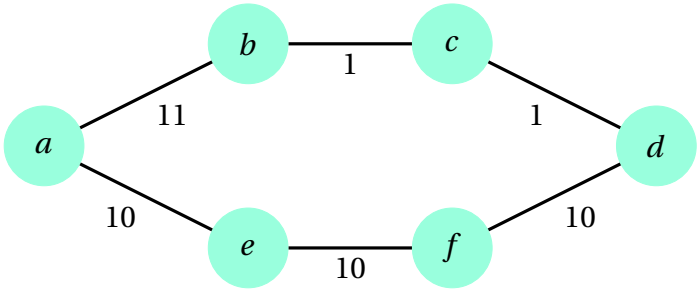
(a) Give an example of a graph in which a shortest path between some pair of vertices is not a bottleneck (and where the graph also contain a bottleneck path which is not a shortest path).

(b) Prove that all of the paths in a minimum spanning tree M of a graph G are bottleneck paths in G.

(c) Prove that not all bottleneck paths of G are paths in a minimum spanning tree M of G.

Solution

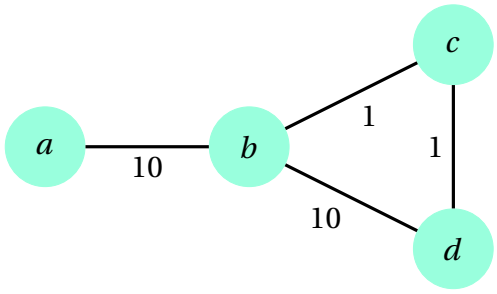
(a) Consider the following graph



The shortest path from a to d has total length 13, but it is not a bottleneck path since it uses an edge of weight 11, while it is possible to travel from a to d using only weight 10 edges. The bottleneck path has total length 30 and hence is not a shortest path.

(b) Consider a weighted, connected, undirected graph G, some minimum spanning tree M of G, and a pair of vertices $s, t \in V$. Since M is a tree, there is a unique path p between s and t in it. Suppose for contradiction that p is not a bottleneck path, i.e. there exists another $s-t$ path p' in G such that the maximum edge weight used in p' is strictly less than the maximum edge weight in p. Let's remove the heaviest edge e on p from M, leaving us with two disconnected subtrees. Since p' connects s and t in G, at least one of its edges must jump between these two subtrees, and since every edge on this path is lighter than e, we can add any such edge back to M, reconnecting it, and yielding a lighter spanning tree than we had initially. This is a contradiction since M was initially a minimum spanning tree, and hence p must be a bottleneck path.

(c) Consider the following graph



The path $a \to b \to d$ is a bottleneck path, but it is not a part of any minimum spanning tree since the minimum spanning tree of this graph has weight 12, which is less than 20.

Problem 5. Consider the following algorithm quite similar to Kruskal's algorithm for producing a minimum spanning tree

```
1: function MINIMUM_SPANNING_TREE(G = (V, E))
2:     sort(E, descending, key((u,v)) = w(u, v))    // Sort edges in order from heaviest to lightest
3:     T = E
4:     for each edge e in nonincreasing order do
5:         if T - {e} is connected then
6:             T = T - {e}
7:     return T
```

Instead of adding edges in order of weight (lightest first) and keeping the graph acyclic, we remove edges in order of weight (heaviest first) while keeping the graph from becoming disconnected.

(a) Identify a useful invariant maintained by this algorithm.

(b) Prove that this invariant is maintained by the algorithm.

(c) Deduce that this algorithm correctly produces a minimum spanning tree.

Solution

(a) At each iteration, T is a superset of some minimum spanning tree of G.

(b) We follow a proof very similar to that of Kruskal's algorithm. Initially, T contains every edge, so it is definitely a superset of some minimum spanning tree's edges.

Suppose that at some iteration, T is a superset of some minimum spanning tree M. Call the next heaviest edge whose removal does not disconnect the graph e. We need to argue that $T-\{e\}$ is a superset of a minimum spanning tree. If $e \notin M$, then $T-\{e\}$ is a superset of M and we are done. Otherwise $e \in M$. Suppose we remove e from M, leaving us with two disconnected subtrees. Since removing e from T does not disconnect T, it must be contained in some cycle in T. This cycle must contain an edge $e' \neq e$ that connects the two subtrees made by removing e from M. Since e was the heaviest edge whose removal does not disconnect the graph, and e' would also not disconnect the graph since it is contained in a common cycle with e, we have that $w(e') \leq w(e)$. Therefore if we join together the two subtrees formed by removing e by adding e', we obtain a spanning tree M' whose weight is

$$w(M') = w(M) - w(e) + w(e') \leq w(M),$$

but since M is a minimum spanning tree $w(M') \leq w(M)$ and hence $w(M') = w(M)$ from which we can deduce that M' is also a minimum spanning tree. Since $e' \in T$ and $e \notin M'$, it is true that M' is a subset of $T-\{e\}$ and hence $T-\{e\}$ is a superset of some minimum spanning tree. Therefore the invariant is maintained.

(c) This algorithm will produce a graph that is connected since it will never remove an edge that causes a disconnection. Suppose it produces a graph containing a cycle, then it contains an edge whose removal would not disconnect the graph, but such an edge would be removed by Line 6. Therefore this algorithm produces a connected graph with no cycles, i.e. a spanning tree. By the invariant shown in (b), the spanning tree produced must be a superset of some minimum spanning tree, but since all spanning trees contain the same number of edges, it must itself be a minimum spanning tree. Therefore this algorithm correctly produces a minimum spanning tree.

Problem 6. Recall from the seminars that breadth-first search can be used to find single-source shortest paths on unweighted graphs, or equivalently, graphs where all edges have weight one. Consider the similar problem where instead of only having weight one, edges are now allowed to have weight zero or one. We call this the *zero-one* shortest path problem. Write pseudocode for an algorithm for solving this problem. Your solution should run in $O(V+E)$ time (this means that you cannot use Dijkstra's algorithm!) [Hint: Combine ideas from breadth-first search and Dijkstra's algorithm]

Solution

Recall that in a breadth-first search, vertices are visited in order by their distance from the source vertex. Suppose that the vertex at the front of the queue has distance d. Any unvisited adjacent vertices will be inserted into the back of the queue at distance $d+1$. This implies that the queue maintained by BFS always contains vertices whose distances differ by at most one. In essence, the queue contains vertices on the current level at the front, and vertices on the next level at the back.

We wish to modify BFS to allow it to handle edges with zero weight. Notice that if we wish to traverse an edge of zero weight, then the target vertex will have the same distance as the current distance. Therefore we would simply like to insert this vertex not at the back, but at the front of the queue! To support this, we could use a data structure called a "deque". A deque is short for *double-ended queue*, which simply means a queue that can be inserted and removed from at both ends. A deque can be implemented quite easily using a standard dynamic array. Alternatively, if you do not wish to use a deque, you can simply maintain two queues, one for the vertices at the current distance d, and one for the vertices at the next

distance $d+1$.

Another interpretation of this algorithm is as a modification of Dijkstra's. Dijkstra's algorithm would solve this problem, but in $O((V+E)\log(V))$ complexity, which is a log factor too high due to the use of a heap-based priority queue. However, as per the observation above, the queue for this algorithm will only ever contain vertices at distance d and distance $d+1$, hence we only need a priority queue that supports having at most two keys at once. Having a double-ended queue and appending the higher keys onto the end and the smaller keys onto the front satisfies this requirement.

Finally, note that unlike ordinary BFS, we also have to check the distances when visiting neighbours now since we cannot guarantee that the first time we discover a vertex that we necessarily have the smallest distance (since we may discover it via a weight one edge when some other node can reach it via a weight zero edge). This is similar to how Dijkstra's algorithm relaxes outgoing edges. A pseudocode implementation in presented below.

```
1:  function ZERO_ONE_BFS(G = (V, E), s)
2:      Set dist[1..n] = ∞
3:      Set pred[1..n] = null
4:      Set deque = Deque()
5:      deque.push((s, 0))
6:      dist[s] = 0
7:      while deque is not empty do
8:          (u, d) = deque.pop()
9:          if d = dist[u] then    // Ignore entries in the queue for out-of-date distances (like Dijkstra's)
10:             for each vertex v adjacent to u do
11:                 if dist[u] + w(u, v) < dist[v] then
12:                     dist[v] = dist[u] + w(u, v)
13:                     pred[v] = u
14:                     if w(u, v) = 0 then deque.push_front((v, dist[v]))    // Add to the front of the queue
15:                     else deque.push_back((v, dist[v]))    // Add to the back of the queue
```

There are other ways to solve this problem. One very different solution is to perform a depth-first search on the graph and find all of the components that are connected by zero-weight edges. Since vertices in these components can reach each other with zero distance, they will all have the same distances in the answer, hence these components can be contracted into single vertices, and then we can perform an ordinary breadth-first search on this contracted graph.

Supplementary Problems

Problem 7. You are the manager of a super computer and receive a set of requests for allocating time frames in that computer. The i-th request specifies the starting time s_i and the finishing time f_i. A subset of requests is compatible if there is no time overlap between any requests in that subset. Develop an algorithm to choose a compatible subset of maximal size (i.e., you want to accept as many requests as feasible, but you cannot select incompatible requests). [Hint: Develop a greedy algorithm.]

Solution

The solution to this problem is a greedy algorithm that works as follows. First, the requests are sorted according to their finishing time. The intuition is that at any point, from the remaining compatible requests, you want to select the one with the smallest finishing time, so that the super computer becomes available again as soon as possible.

The algorithm loops through the sorted list of requests proceeding as follows: if the request is compatible

with the previously chosen requests, then it is selected; otherwise it is discarded.

The idea of the proof of correctness that this algorithm obtains an optimal solution (i.e., a maximal compatible subset) is to demonstrate that the solution output by this algorithm "stays ahead" of any other solution. Let $i_1, i_2, \ldots, i_m$ be the subset of requests output by the algorithm (in time order, remember that this subset is compatible so there is no overlap). Suppose, for contradiction, that there is a bigger compatible subset and let $j_1, j_2, \ldots, j_n$ be its requests (also in time order) with $n > m$. Prove by induction that $f_{i_k} \leq f_{j_k}$ for $k = 1, 2, \ldots, m$. But then j_{m+1} is compatible with $i_1, i_2, \ldots, i_m$, and should have been selected by the algorithm, which is a contradiction. Therefore the algorithm obtains a compatible subset of maximal size.

Note that sorting by starting time and proceeding greedily (by processing the requests in increasing order of starting time and adding the compatible ones) would not work, as the request that starts first can also be the last one to finish.

The idea of ordering the requests according to the time that they take and proceeding greedily would also not work. Consider, for instance, the case where the first request starts at 1 and finishes at 10, the second starts at 9 and finishes at 12, and third request starts at 11 and finishes at 20. The first and third requests are compatible, but if you select the second request, then you cannot select any other request.

Problem 8. For the union-find data structure, there are multiple heuristics to handle how we union two disjoint-set trees. One such heuristic is "union-by-size", where the disjoint-set with less elements is appended to the disjoint set with more elements. We want to prove that the worst-case cost of all the union operations is $O(V \log(V))$ for a disjoint set of V elements, when using this heuristic.

(a) First prove the following lemma: For any disjoint-set tree constructed via "union-by-size" heuristic, $\text{size}(r) \geq 2^{\text{height}(r)}$, where r represents the root node of some disjoint-set tree, $\text{size}(r)$ represents the number of nodes in the disjoint set tree, and $\text{height}(r)$ represents the height of the tree.

(b) Using the lemma from part (a), prove that a union takes $O(\log(V))$.

(c) Hence, prove that all the unions take $O(V \log(V))$.

Solution

(a) We will prove the lemma inductively.

Base Case:

When we initialise the union-find data structure, every tree is just a singular node. We have that the $\text{size}(r) = 1$ and the $\text{height}(r) = 0$. The proposed lemma clearly holds, hence the base case is true.

Inductive step:

Assume that after k unions the statement is true for all disjoint-set trees in the data structure. In the $(k+1)$-th union, say we are merging two disjointed-set trees, rooted at s and r. Without loss of generality let's assume that $\text{size}(r) \geq \text{size}(s)$ which means the tree rooted at s is put beneath r.

After performing the $(k+1)$-th merge we denote the old size and height as as $\text{size}(r)$, $\text{height}(r)$ and the new size and height as $\text{size}'(r)$, $\text{height}'(r)$.

Case 1: $\text{height}(r) > \text{height}(s)$

After performing the $(k+1)$-th merge we have

$$\text{size}'(r) \geq \text{size}(r)$$

We then invoke our hypothesis where $\text{size}(r) \geq 2^{\text{height}(r)}$. Therefore

$$\text{size}'(r) \geq 2^{\text{height}(r)}$$

Since height(r) > height(s) and the disjoint-set tree rooted at s is put beneath r, we must have height(r) = height$'(r)$. Therefore we have that

$$\text{size}'(r) \geq 2^{\text{height}'(r)}$$

This proves the lemma for this case.

Case 2: height$(r) \leq$ height(s)

After performing the $(k+1)$-th merge we have

$$\text{size}'(r) = \text{size}(r) + \text{size}(s)$$

From our assumption of size$(r) \geq$ size(s), we have that

$$\text{size}'(r) \geq 2 \times \text{size}(s)$$

We then invoke our hypothesis where size$(s) \geq 2^{\text{height}(s)}$. Therefore we have that

$$\begin{aligned}\text{size}'(r) &\geq 2 \times 2^{\text{height}(s)} \\ &\geq 2^{\text{height}(s)+1}\end{aligned}$$

As we are putting the disjoint-set tree rooted at s beneath r, and height$(r) \leq$ height(s), we must then have height$'(r)$ = height$(s)+1$. Therefore we have

$$\text{size}'(r) \geq 2^{\text{height}'(r)}$$

This proves the lemma for this case. Therefore, by induction, it is true that size$(r) \geq 2^{\text{height}(r)}$.

(b) The cost of any union is simply $O(\text{height}(r)+\text{height}(s))$. It follows from the lemma above that

$$\begin{aligned}\text{size}(r) &\geq 2^{\text{height}(r)}, \\ \log_2(\text{size}(r)) &\geq \text{height}(r).\end{aligned}$$

Since size(r) for a disjoint set of V elements is bounded by V. We have that

$$\log_2(V) \geq \text{height}(r).$$

Similarly, we can show that

$$\log_2(V) \geq \text{height}(s).$$

Thus any union is worst case $O(\log(V))$.

(c) A union operation is only ever performed if two nodes are in different disjoint-sets. Thus, in the worst-case lifespan of the union-find data structure of V elements, only $V-1$ unions can be performed. Using the above it follows that the worst-case cost of all the unions is $O(V\log(V))$.

Problem 9. Suppose that we wish to solve the single-source shortest path problem for graphs with non-negative bounded integer edge weights, i.e. there is a constant c such that for all edges (u,v) it holds that $0 \leq w(u,v) \leq c$. Explain how we can modify Dijkstra's algorithm to run in $O(V+E)$ time in this case. [Hint: Improve the priority queue]

Solution

The part of Dijkstra's algorithm that adds the log factor is the priority queue, so let's try to improve that. Given that the edge weights are bounded above by c, and a shortest path can contain at most $V-1$ edges,

the largest distance estimate that we can ever have is less than cV. Also observe that since we remove vertices from the priority queue in distance order, the minimum element in the priority queue never decreases. We can use these two facts to make a faster priority queue for Dijkstra's.

Let's just store an array of size cV, where at each entry, we store a linked list of vertices who have that distance estimate. Adding an element to this priority queue takes $O(1)$ since we can simply append to the corresponding linked list. Updating a distance estimate can also be achieved in $O(1)$ since we can append to linked lists in O(1) (using the approach to implementing Dijkstra described in the notes). Since the minimum distance for entries obtained from priority queue never decreases, we can simply move through the priority queue, starting at distance zero and moving onto the next distance when there are no vertices left to process at the current distance. We never move back to a previous distance.

All updates to the priority queue take $O(1)$. Finding the minimum element in the priority queue could take up to $O(V)$, but note that since we never go backwards, the total time complexity of **all** find minimum operations will be $O(V)$, since scanning all of the distances takes $O(cV) = O(V)$ as c is a constant. The total time complexity of Dijkstra's using this priority queue will therefore be $O(V+E)$.

Problem 10. Consider the following supposed invariant for Kruskal's algorithm: At each iteration, each connected component in the current spanning forest is a minimum spanning tree of the vertices in that component.

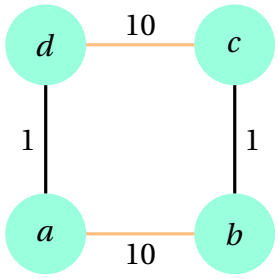
(a) Prove or disprove whether Kruskal's algorithm maintains this invariant.

(b) Can this invariant be used to prove that Kruskal's algorithm is correct?

Solution

(a) Kruskal's algorithm does maintain this invariant, as it is a strictly weaker version of the invariant we usually show for Kruskal. Recall that Kruskal's maintains that the current set of selected edges is always a subset of some minimum spanning tree. Let G be a weighted, connected, undirected graph and let T be a subset of some minimum spanning M tree of G. Consider a connected component of T. Suppose for contradiction that this component is not a minimum spanning tree on its vertices. Then we could replace the edges of this component in M with the edges of a minimum spanning tree on these vertices and reduce the weight of M, meaning that M was not a minimum spanning tree. This is a contradiction, and hence all of the connected components of T must be minimum spanning trees on their respective vertices.

(b) Despite the fact that Kruskal's maintains this invariant, it is not strong enough to prove the algorithm's correctness. Consider the following graph for example.



The edges (a,b) and (c,d) are both minimum spanning trees of their connected components, but no matter how we connect them, we will not obtain a minimum spanning tree for the entire graph. Therefore this invariant alone is too weak to show that Kruskal's is correct.

Note: Clearly the situation above could never actually arise during the execution of Kruskal's algorithm (because Kruskal's is correct, which we have proven elsewhere). It is, however, a valid coun-

terexample to the suggested invariant.