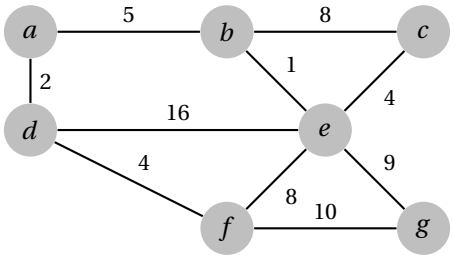
Week 11 Tutorial Sheet

(Solutions)

Useful advice: The following solutions pertain to the theoretical problems given in the tutorial 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 the tutorial 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, psuedocode may be provided where it illustrates a particular useful concept.

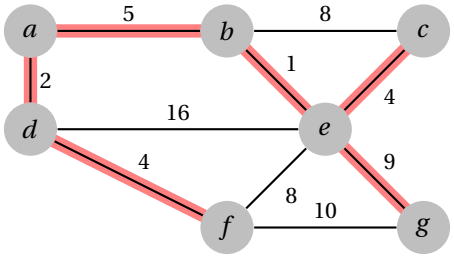
Assessed Preparation

Problem 1. Show the steps taken by Prim's and Kruskal's algorithms for computing a minimum spanning tree of the following graph. Use vertex a as the root vertex for Prim's algorithm. Make sure that you indicate the order in which edges are selected, not just the final answer.



Solution

For Prim's algorithm, we begin at vertex a and select edges in the following order: (a,d), (d,f), (a,b), (b,e), (e,c), (e,g). We end up with the following minimum spanning tree.



Kruskal's algorithm will select edges in the following order: (b,e), (a,d), (d,f), (e,c), (a,b), (e,g). Note that (d,f) and (e,c) could be selected in either order since they have the same weight. The final minimum spanning tree is the same one as obtained by Prim's.

Problem 2. Revise the Union-Find data structure, described in lectures and in section 14.3 of the unit notes.

Tutorial Problems

Problem 3. In the union-find data structure, when performing a union operation, we always append the set with fewer elements to the set with more elements. This heuristic is called union by size heuristic. Prove that in

the worst case we can perform all union operations in $O(V \log(V))$ time when using union by size.

Solution

Find operations cost $O(h)$, where h is the height of the tree, since we need to traverse up to the root. Union operations require two find operations, to identify the two roots involved, and then $O(1)$ additional work to update one of the roots to be the child of the other, and to update the size of the continuing root. So the cost of a union is also $O(max(h_1, h_2))$, where h_1 and h_2 are the heights of the two trees involved in the union.

Since we always append the smaller tree to the larger, we can show that the height of all trees is bounded by $O(\log(N))$, where N is the number of nodes in that tree. This can be proven by induction (left as an exercise). Since $N \leq V$, the heights of all trees are bounded by $O(\log(V))$, so any union operation takes at most $O(\log(V))$. Since we start with V single-element sets, we need to perform $V-1$ unions, so the total work is bounded by $O(V \log(V))$.

Problem 4. Discuss union by rank heuristic and path compression technique to improve the performance of the union-find data structure.

Solution

See Chapter 14 of the Lecture Notes (available on Moodle in week 10 tab).

Problem 5. 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.*

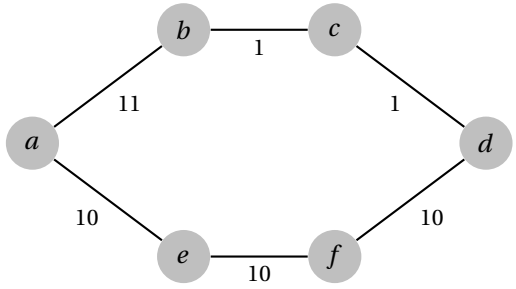
(a) Give an example of a graph in which a shortest path between some pair of vertices is not a bottleneck path and vice versa

(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

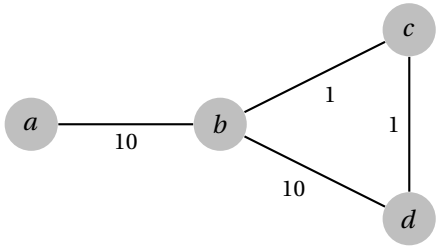


The shortest path from a to d has total length 13, but it is not a bottleneck path since it uses an edge of weight 11, while it is possible to travel from a to d using only weight 10 edges. The bottleneck path has total length 30 and hence is not a shortest path.

(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 6. 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.

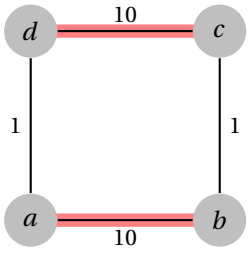
(a) Prove or disprove whether Kruskal's algorithm maintains this invariant.

(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.

Problem 7. 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:          end if
8:      end for
9:      return T
10: end function
```

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. 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 minimum 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 8. Can Prim's and Kruskal's algorithm 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 a MST in the original graph.

Supplementary Problems

Problem 9. Implement Kruskal's algorithm.