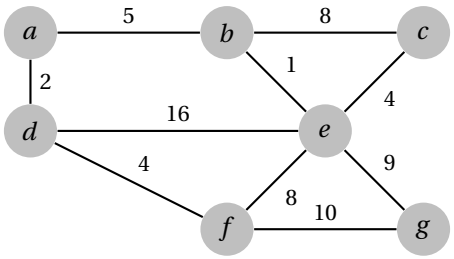
Week 11 Tutorial Sheet

(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.

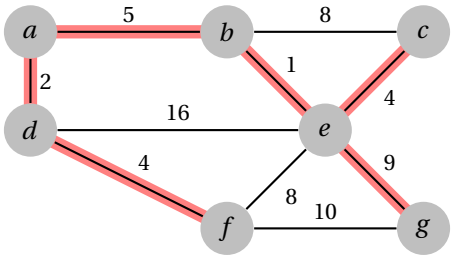
Assessed Preparation

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. Read Chapter 14 "Dynamic Connectivity in Graphs" in Lecture Notes (see Moodle week 10). Compare "The first approach – disjoint set linked lists" with the Union-Find data structure discussed in lectures. Do they have same time and space complexities?

Solution

The two data structures are essentially the same. For the data structure in the lecture notes, each node has a pointer to the head (representative) of the linked list it belongs to which is similar to maintaining the set id using a map array in the lecture notes. Taking the union of two sets is also similar. Thus, the two approaches are very similar and have same the space and time complexities.

Tutorial Problems

Problem 3. In the union-find data structure, when performing a union operation, we always append the shorter list to the longer list, resulting in fewer updates being required. This heuristic is called union by size heuristic. Prove that in the worst case we can perform all n union operations in $O(n\log(n))$ time when using union by size.

Solution

Find operations cost $O(1)$ since each element maintains a pointer to its head element, so we only need to consider the cost of the union operations. Since we append the shorter list to the longer list, we only need to update the head pointers for the elements in the short list. Instead of trying to analyse the amount of work done per union, it is much simpler to instead consider the amount of work done per node of the linked list over all of the union queries.

Consider a particular node. If it is part of the shorter list in a union query and needs to be updated, then the list that it is now contained in is at least double the size of its former list. Therefore the list size for each node at least doubles per union operation. Initially there are n lists of length 1, and after performing all $n-1$ (non-redundant) union operations, we are left with a single list of length n. Since the list size of a node at least doubles when it is updated, it can only be updated at most $\log_2(n)$ times, since after the this the list must be of size n, meaning that all union operations have been performed. Since each node can only be updated $\log_2(n)$ times, the amount of work done over all of the nodes in all of the union queries is $O(n\log(n))$.

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 the following abridged attempted proof of correctness for the invariant of Prim's algorithm. Recall that Prim's algorithm maintains the invariant, at each iteration, the currently selected set of edges T is a subset of some minimum spanning tree of G.

- Suppose that T is a subset of some minimum spanning tree M
- Prim's algorithm next selects the edge $e = (u, v)$
- Suppose for contradiction that e is not a part of any minimum spanning tree
- Let M' be a minimum spanning tree not containing e
- There must be a path connecting u and v in M'. Let e' be the first edge on this path
- We can replace e' with e and still have a spanning tree.

- Since $w(e) \leq w(e')$ (or Prim's would have selected e' instead), this spanning tree is also a minimum spanning tree
- Therefore by contradiction, e is indeed part of a minimum spanning tree
- Therefore $T \cup \{e\}$ is a subset of some minimum spanning tree

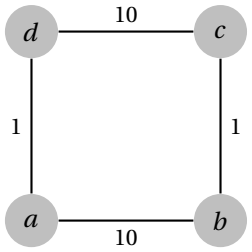
There are several mistakes in this proof. Identify them, explain why they are incorrect, and explain how to correct them to obtain a valid proof.

Solution

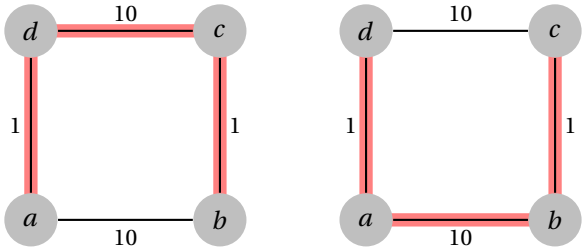
There are three primary mistakes in this proof. Of course, since the proof can be corrected in many ways, it is possible to diagnose the mistakes and correct them in different ways. We give one such way for each example.

The contradictory assumption

The first mistake is dot points three and eight. Assuming that e is not a part of **any** minimum spanning tree will not lead to the desired result. If we prove that this assumption is wrong by showing that there does exist **some** minimum spanning tree containing e, this is not sufficient to show that $T \cup \{e\}$ is a subset of a minimum spanning tree. For example, consider the following graph:



There are two possible minimum spanning trees, shown below.



In this case, every edge is part of **some** minimum spanning tree, which shows us that proving that e is part of some minimum spanning tree tells us no useful information about whether it is a part of the particular spanning tree that Prim's is currently building. For instance, if we had currently selected the edges (a, d) and (d, c), it would be wrong to select (a, b) next, but the proof above would not argue against this since (a, b) is a part of **some** minimum spanning tree.

In order to correct this, the contradictory assumption could be changed to assume that $T \cup \{e\}$ is not a subset of any minimum spanning tree, since proving this to be wrong proves precisely the invariant that we wish to maintain. Although, it turns out that the contradictory assumption in this proof is actually superfluous and not necessary anyway, so another valid answer is to simply remove the contradiction part of the proof.

Let M' be a minimum spanning tree not containing e

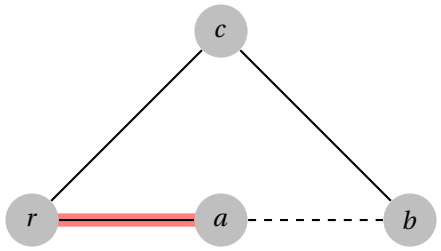
This line has some problems similar to the issues above. Allowing M' to be any MST not containing e means that it does not necessarily have to have the same structure as the MST T that we are currently

building. Since our end goal is to show that $T \cup \{e\}$ is a subset of an MST, we need M' to contain T, as otherwise, when we make the replacement in dot points 6 and 7, we may indeed have an MST, but it will not be a superset of T.

To correct this, we use the fact that the invariant asserts that T is a subset of a minimum spanning tree M. If M contains e then nothing is done as the invariant is satisfied. Otherwise, we consider M in place of M', since it is guaranteed that M is a superset of T, and that is does not contain e, and the proof can proceed.

Breaking the path connecting u and v

It is true that since M' is a tree that it necessarily contains a path from u to v. However, selecting the first edge on this path as the replacement is not guaranteed to work, since it is possible that the first such edge is already in T. Consider the following graph as an illustrative example.



If the edge e under consideration in the proof above is (a,b), and the MST M turns out to consist of the edges $(r,a),(r,c),(c,b)$, then the path from a to b crosses back over the edge (r,a), which means that we can not swap it out and still have a superset of T. Instead, the correct thing to do is to consider any edge on the path that is not contained in T. At least one such edge is guaranteed to exist since a and b are not already connected in T.

Problem 6. 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.*

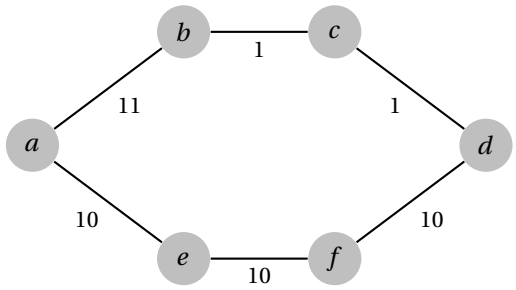
(a) Give an example of a graph in which a shortest path between some pair of vertices is not a bottleneck path and vice versa

(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

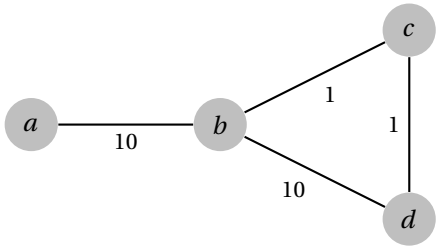


The shortest path from a to d has total length 13, but it is not a bottleneck path since it uses an edge

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 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.

Problem 9. Implement Kruskal's algorithm.