Comp 251: Practice problems

Instructor: Jérôme Waldispühl

This is a collection of problems that you can use to prepare the COMP 251 midterm exam.

- The first part of the exam (40 points) will be composed of multiple choice questions similar to the quizzes following each lecture.
- The second part of the exam (60 points) will be composed of longer problems such are those illustrated below.

Hashing

1. Suppose that you use a hash table and a hash function implementing the division method. The following keys are inserted: 5, 28, 19, 15, 20, 33, 12, 17, 10 and m = 9 (for simplicity, here we do not distinguish a key from its hashcode, so we assume h(key)=key). In which slots do collisions occur?

   Solution: The keys 5, 28, 19, 15, 20, 33, 12, 17, 10 map to slots 5, 1, **1**, 6, 2, **6**, 3, 8, **1**. Collisions are shown in bolded fonts.

2. Now suppose you use open addressing with linear probing and the same keys as above are inserted. More collisions occur than in the previous question. Where do the collisions occur and where do the keys end up?

   Solution: The key 19 collides with key 28 at slot 1 and 19 goes into slot 2. Key 20 collides with key 19 at slot 2 and needs to go to slot 3. Key 33 collides with key 15 at slot 6 and needs to go to slot 7. Key 12 collides with key 20 at slot 3 and needs to go to slot 4. Key 17 goes into position 8. Key 10 collides with key 28 at slot 1, and then collides with 19 at slot 2, etc until it finally finds an open slot at position 0. The final positions of the keys are [10, 28, 19, 20, 12, 5, 15, 33, 17].

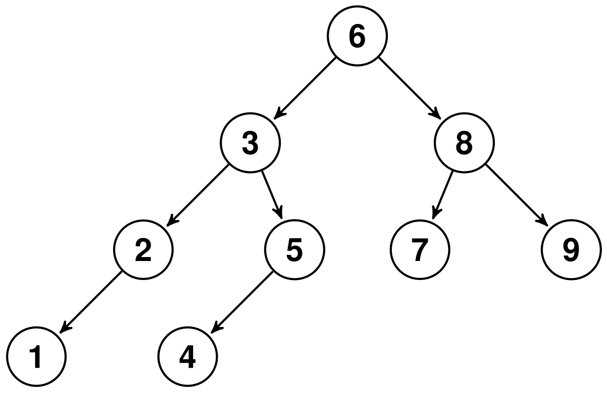
Balanced binary search trees

3. What is the maximum number of nodes in an AVL tree of a given height h?

Solution: This would be a complete binary tree of height h. Such a tree has $2^{h+1} - 1$ nodes (See COMP 250!).

4. Starting with an empty tree, construct an AVL tree by inserting the following keys in the order given: 2, 3,5, 6, 9, 8, 7, 4, 1. If an insertion causes the tree to become unbalanced, then perform the necessary rotations to maintain the balance. State where the rotations were done.

Solution: Add(2), add(3), add(5), rotateLeft(2), add(6), add(9), rotateLeft(5) and note that this solves the imbalance at 3 also, add(8), rotateLeft(3), add(7), rotateRight(9), add(4), add(1). The final AVL tree is:



For your practice, we recommend you to draw all intermediate steps.

Heaps

5. Suppose you have a heap which supports the usual add() and removeMin() operations, but also supports a changePriority (name, new Priority) operation. How could you combine these operations to define a remove(name) operation?

Solution: Check the current minimum and find out what its priority is. (You can remove the min, check its value, and then add it back in.) Then use the changePriority() method to reduce the priority of the element that you wish to remove, such that its new priority is less than that of the current minimum. Then, remove the (new) minimum.

6. Suppose you used an ordered array to implement a priority queue. Give the O( ) time for the operations removeMin(), add(element, key), findMin() take?

Solution: If you order from small to large then removeMin() would be O(n). It would be better would be to order from large to small so that removeMin would be O(1). Adding an arbitrary element would be O(n) since you would have to shift all the elements in the worst case. findMin would be O(1) since the array is ordered.

Disjoint sets

7. Consider the set of all trees of height h that can be constructed by a sequence of "union-by-height" operations. How many such trees are there?

Solution: It was a trick question. There are infinitely many of these trees, since there is no constraint given on the number of nodes and "union-by-height" trees (trees built by union-by-height operations) have no constraint on the number of children of any node.

8. Consider a tree formed by union-by-height operations, without path compression. Suppose the tree has n nodes and the tree is of height h. Show that n is greater than or equal to 2^h. (Note that the tree need not be a binary tree, and so we cannot just apply properties of binary trees to this problem. Indeed, for binary trees of height h, we can only say that the number of nodes at most $2^{h+1} - 1$, which is looser than the bound stated in the question.)

Solution: Here is a proof by induction on the tree height k. The base case $k = 0$ is easy, since a tree of height 0 always has just $1 = 2^0$ node (the root). Suppose the claim is true for $h = k$. Now consider a union-by-height tree of height $k + 1$. There must have been a union that brought two trees together and increased the height of one of them from k to $k + 1$. Let those two trees (at the time of that union) be T1 and T2. We know that both T1 and T2 were of height k before the union. [Why? If one of them were of height less than k, then union-by-height would have changed the root of that shorter one to make it point to the root of the taller one, and the height of the unioned tree would still be k. But its not the unioned tree is of height k+1.]
Now we can apply the induction hypothesis: the trees T1 and T2 each have at least 2^k nodes. Thus, the unioned tree has at least $2^k + 2^k = 2^{k+1}$ nodes.

Minimum spanning-trees

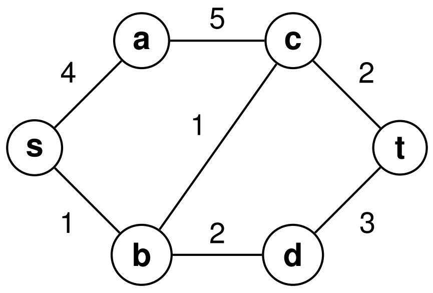
9. Prove that for any weighted undirected graph such that the weights are distinct (no two edges have the same weight), the minimal spanning tree is unique.

Solution: Suppose there are two MSTs, call them T1 and T2. Let the edges of T1 be $\{e_{i_1}, e_{i_2}, ...e_{i_{n-1}}\}$ and the edges of T2 be $\{e_{k_1}, e_{k_2}, ...e_{k_{n-1}}\}$. If the trees are different, then these sets of edges must be different.
So, let e* be the smallest cost edge in T1 that is not in T2. Deleting that edge from T1 would disconnect T1 into two components. Since T1 chose that edge, it must be the smallest cost crossing edge for those two components. But by the cut property, that edge must therefore belong to every minimum spanning tree. Thus it must belong to T2 as well. But this contradicts the definition of e*.

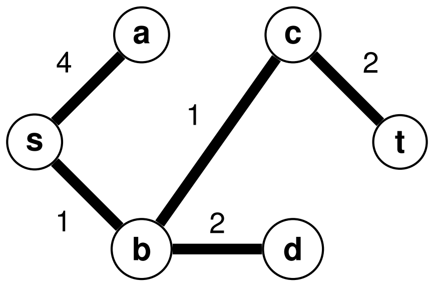
10. If a connected undirected graph has n vertices, then any spanning tree has n-1 edges.

Solution: Consider the edges in a spanning tree T and consider a graph with no edges, but all n vertices. Now add the edges of the spanning tree one by one. Each edge is a crossing edge between two connected components and adding the edge reduces the number of connected components by 1. Since adding all the edges of T (one by one) reduces the number of connected components from n down to 1, it must be that T has n-1 edges.

11. Consider the flow graph below. Apply the Kruskal algorithm to calculate the minimum spanning tree.



Solution: The minimum spanning tree is:



Single-source shortest paths

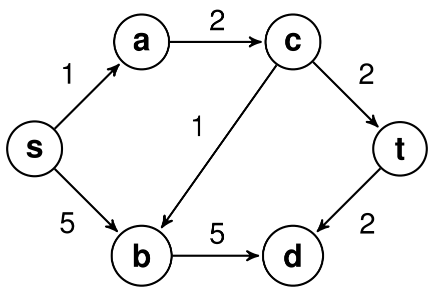
12. In breadth first search, each vertex has a "visited" field which is set to true before the vertex is put in the queue. What happens if BFS instead sets the visited field to true when the vertex is removed from the queue? Does the algorithm still work? Does it run just as fast? What if we want to find the shortest path between every pair of vertices in the graph ?

Solution: The algorithm still works. However, multiple copies of the vertex can be put the queue. The maximum number of copies is the in-degree of the vertex. The size of the queue grows as O(|E|) rather than O(|V|).

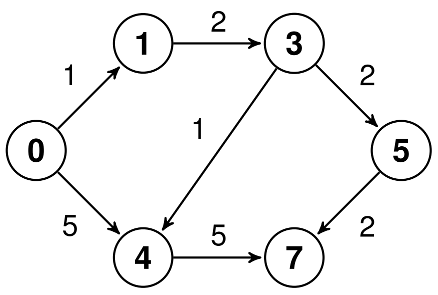
13. Dijkstra's algorithm assumes the edges have non-negative weights. (Where does this come up in the proof of correctness?) But suppose we have a graph with some negative weights, and let edge e be such that cost(e) is the smallest (most negative). Consider a new graph in which we add cost(e) to all the edge weights, thus making all weights in the new graph non-negative. Now the conditions hold for Dijkstra to find the shortest paths, so we could now run Dijkstra. Is this a valid way to solve for shortest paths in the case that some edges have negative weights? Justify your answer.

Solution: No. For any path in the original graph, the distance of that same path P in the new graph will be greater by cost (e) multiplied by the number of edges in the path P, since we incremented each edges cost by cost(e). But for two paths with a different number of edges, the totals for the extra costs that we have added will be different. So there is no reason why you should expect to get the same solutions in the two cases. [Note that by ?same solutions? here, I don?t just mean the same distances of the shortest paths; I mean the paths themselves!] For example, take the graph with three edges cost(u,v) = -2, cost(v,w) = 2, cost(u, w)=1. The shortest path to w is (u, v, w). But adding 2 to the cost of each edge and then running Dijkstra would give the shortest path as (u, w).

14. Consider the flow graph below. Apply the Dijkstra's algorithm to calculate the shortest paths from s.



Solution: The solution is:



Where the shortest path estimates are stored inside the vertices.

Bipartite graphs

15. Given the preferences shown here, use the Gale-Shapley algorithm to find a stable matching.

A	A's preferences	B	B's preferences
α_1	β_1, β_2, β_3	β_1	α_3, α_1, α_2
α_2	β_1, β_2, β_3	β_2	α_1, α_3, α_2
α_3	β_1, β_2, β_3	β_3	α_3, α_2, α_1

Solution: The answer is $(\alpha_1, \beta_2), (\alpha_2, \beta_3), (\alpha_3, \beta_1)$.

16. Consider an instance of the stable matching problem in which, for all $\alpha \in A$, α's first choice is β if and only if β's first choice is α. In this case, there is only one stable matching. Why?

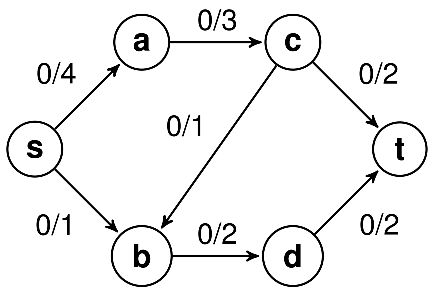
Solution: By definition, a matching is unstable if there exists an $\alpha \in A$ and a $\beta \in B$ that prefer each other over their present partners. Take any matching for which some $\alpha \in A$ doesn't get its first choice. Let the first choice of α be β. Then β is not getting its first choice either, since β's first choice is α. But then this matching is not stable. Thus, for any stable matching, every α gets its first choice, and so every β (by the constraints given in this question) gets its first choice too.

Flow networks

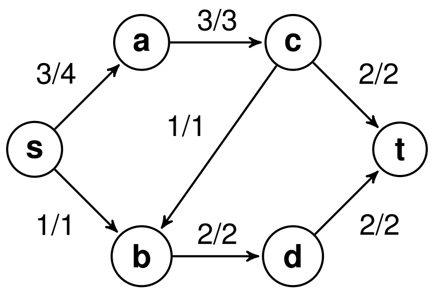
17. Suppose the capacities in a network flow are not integers. How would this change the O( ) runtime of the Ford Fulkerson algorithm? Does the algorithm terminate?

Solution: The stated runtime of Ford-Fulkerson is O(C m) as explained in the lecture, where C is an integer namely the sum of integer capacities of edges out of s. If the capacities are not integers, then this O( ) no longer makes sense because O( ) always assumes we are talking about integers i.e. number of operations. More concretely, if we applied the Ford-Fulkerson algorithm to a flow network having non-integer capacities, we would still find that the flow increases in every pass through the main loop. However, the amount by which the flow increases might be a very small number and this number might get smaller and smaller indefinitely. There is no reason why the while loop would ever terminate. (Here we ignore the precision limits of representing floating point numbers on real computers, which you will learn about in COMP 273 or ECSE 221, if you haven't already)

18. Consider the flow graph below. Apply the Ford-Fulkerson algorithm to calculate the maximum flow.



Solution: The final solution is:



a
3/3
c
3/4
2/2
1/1
s
t
1/1
2/2
2/2
b
d