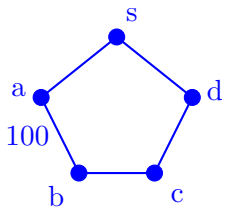
Assignment 4 Solutions

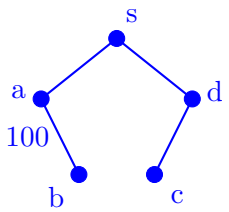
1 Graph algorithms [10 marks]

(i) Give an example of a connected, weighted, undirected graph G and a start vertex s such that neither the DFS spanning tree nor the BFS spanning tree of G from s is a minimum weight spanning tree of G, regardless of how the adjacency lists are ordered. Justify the correctness of your counter-example.

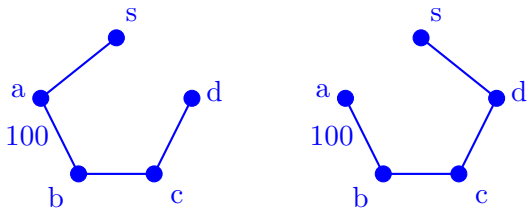
Solution. Consider the graph



with all edge weights 1 except for the edge (a, b) which has weight 100. The BFS tree starting at s yields the spanning tree



with weight 103 and the two possible spanning trees obtained from DFS started at s are



which both also have weight 103. The spanning tree obtained by removing edge (a, b), by contrast, has total weight 4.

(ii) Prove that if the BFS and DFS spanning trees of a connected undirected graph G from the same start vertex s are equal to each other, then G is a tree.

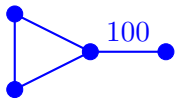
Solution. Assume for contradiction that G is a c connected graph that is not a tree and that there is a vertex s such that the BFS tree T and the DFS tree T' are equal to each other. Since G is not a tree, there must be some cycle in G and some non-tree edge (u, v) connecting u to one of its ancestors v in the DFS tree T'. But then u cannot be a descendant of v in the BFS tree T unless it is its child (in which case (u, v) is a tree edge) so $T \neq T'$.

2 Minimum spanning trees [10 marks]

Prove or disprove each of the following statements, where in each case $G = (V, E)$ is a connected undirected weighted graph with n vertices and m edges.

(i) If G has $m \geq n$ edges and a unique heaviest edge e, then e is not part of any minimum spanning tree of G.

Solution. **False.** Consider the graph



that has 4 vertices and 4 edges, three of which of weight 1 and one of which has weight 100. The unique heaviest edge of weight 100 must be included in any spanning tree of the graph, so it is also in the minimum spanning tree.

(ii) If G has $m \geq n$ edges and a unique lightest edge e, then e is part of every minimum spanning tree of G.

Solution. **True.** The proof is very similar to that of the Cut Property in Lecture 14. Assume for contradiction that there is a MST T of G that does not include e. Then if we add the edge e to T, we create a cycle in the graph and we can remove any other edge $e' \neq e$ in the cycle to obtain another spanning tree T' of G. But the resulting tree T' has smaller weight than T, contradicting the fact that T was a MST of G.

(iii) If e is a maximal weight edge of a cycle of G, then there is a minimum spanning tree of G that does not include e.

Solution. **True.** Let $T = (V, E')$ be a MST of G that includes e. Let $X = E' \setminus \{e\}$ be the set of edges in T *except* for e. Then X is part of a MST of G and there must be a set $S \subseteq V$ (that includes exactly one of the two end vertices of e) for which X has no edges that crosses the cut $(S, V \setminus S)$. The edge e crosses the cut, but since e is part of a cycle, there must be another edge $e' \neq e$ in the cycle that in the minimum-weight edge crossing the cut. (The edge e cannot be the unique minimal weight edge crossing the cut since it has maximal weight among the edges in the cycle.) By the Cut Property Lemma, the tree T' with edges $X \cup \{e'\}$ is a MST of G that does not contain the edge e.

(iv) Prim's algorithm returns a minimum spanning tree of G even when the edge weights can be either positive or negative.

Solution. **True.** The analysis of Prim's algorithm in the written notes for Lecture 14 does not assume anything about the edge weights, so it works with both positive and negative weights. If we want to be extra careful and assume that this analysis only works for non-negative weights, then for any graph G with some negative edge weights, let $-\gamma$ denote the minimum weight of any edge of G. Let $G' = (V, E)$ be the weighted graph with weight function $w' : E \to \mathbb{R}$ defined by $w'(e) = w(e) + \gamma$. Then the graph G' has non-negative edge weights and Prim's algorithm computes a MST T' of G'. The same tree is also a MST of G since the for every tree T, $\text{cost}_w(T) = \text{cost}_{w'}(T) - (n-1)\gamma$.

3 Shortest paths [10 marks]

A *vertex-and-edge-weighted graph* is a directed graph $G = (V, E)$ where each vertex $v \in V$ has a *cost* $c(v)$ and every edge $e \in E$ has a *weight* $w(e)$. The *length* of a path in G is the sum of the weights of the edges in the path *and* the cost of the vertices in the path. In the Single-Source Shortest Path (SSSP) problem on vertex-and-edge-weighted graphs, we are given G and a source vertex s and we want to determine the minimum length of a path from s to v for every $v \in V$. (Note that the minimum length of a path from s to s is $c(s)$.)

Solve the SSSP problem on vertex-and-edge-weighted graphs in the special case where all the weights and the costs are positive integers.

Solution.

Algorithm description. There are a few different ways that we can reduce the problem to SSSP on (regular) weighted graphs. Here's a reduction that works for both directed and undirected graphs.

Given a vertex-and-edge-weighted graph G, construct the weighted graph $G' = (V, E)$ with edge weight function $w' : E \to \mathbb{R}$ defined by

$$w'(u, v) = w(u, v) + \tfrac{1}{2}c(u) + \tfrac{1}{2}c(v).$$

We then solve the SSSP problem on G' with the same initial vertex s using Dijkstra's algorithm to obtain the shortest path distance $d'(v)$ from s to v for each $v \in V$. For each vertex $v \in V$, we return the shortest path distance

$$d(v) = d'(v) + \tfrac{1}{2}c(s) + \tfrac{1}{2}c(v).$$

Pseudocode. The algorithm is as follows.

Algorithm 1: DijkstraW+C($G = (V, E)$, w, c, s)

for *each* $(u, v) \in E$ **do**
  $w'(u, v) \leftarrow w(u, v) + \frac{1}{2}c(u) + \frac{1}{2}c(v)$;

$d' \leftarrow$ Dijsktra(G, w', s);

for *each* $v \in V$ **do**
  $d(v) \leftarrow d'(v) + \frac{1}{2}c(s) + \frac{1}{2}c(v)$;

return d;

Proof of correctness. We know from lectures and from the textbook that Dijkstra's algorithm correctly computes the weight of the shortest path between s and every vertex v in G'. The correctness of our algorithm follows from the following identity.

Theorem 1. *The length of every path $P = (v_0, v_1, \ldots, v_k)$ in G satisfies*

$$\sum_{i=1}^{k} w(v_{i-1}, v_i) + \sum_{j=0}^{k} c(v_j) = \sum_{i=1}^{k} w'(v_{i-1}, v_i) + \tfrac{1}{2}c(v_0) + \tfrac{1}{2}c(v_k).$$

Proof. By definition, the sum of the weights w' in the path satisfies

$$\sum_{i=1}^{k} w'(v_{i-1}, v_i) = \sum_{i=1}^{k} \big(w(v_{i-1}, v_i) + \tfrac{1}{2}c(v_{i-1}) + \tfrac{1}{2}c(v_i)\big).$$

For each $1 \leq j < k$, the expression $\frac{1}{2}c(v_j)$ appears twice (when $i = j$ and $i = j + 1$) and the terms $\frac{1}{2}c(v_0)$ and $\frac{1}{2}c(v_k)$ appear once, so

$$\begin{aligned}\sum_{i=1}^{k} w'(v_{i-1}, v_i) &= \sum_{i=1}^{k} w(v_{i-1}, v_i) + \sum_{j=1}^{k-1} c(v_j) + \tfrac{1}{2}c(v_0) + \tfrac{1}{2}c(v_k) \\ &= \sum_{i=1}^{k} w(v_{i-1}, v_i) + \sum_{j=0}^{k} c(v_j) - \tfrac{1}{2}c(v_0) - \tfrac{1}{2}c(v_k),\end{aligned}$$

as we wanted to show. □

The theorem implies that the shortest path from s to v in G is also the shortest path from s to v in G' and that their lengths are related by the identity $d(v) = d'(v) + \frac{1}{2}c(s) + \frac{1}{2}c(v)$ so our algorithm computes the lengths of all shortest paths in G correctly.

Time complexity analysis. The time complexity of our algorithm is $T(n, m) + O(m + n)$ where $T(n, m)$ is the time complexity of Dijkstra's algorithm on graphs with n vertices and m edges. The most efficient implementation of Dijkstra's algorithm, as seen in the course textbook, has time complexity $O(n \log n + m)$, so this is also the time complexity of our algorithm.

4 Water puzzles [10 marks]

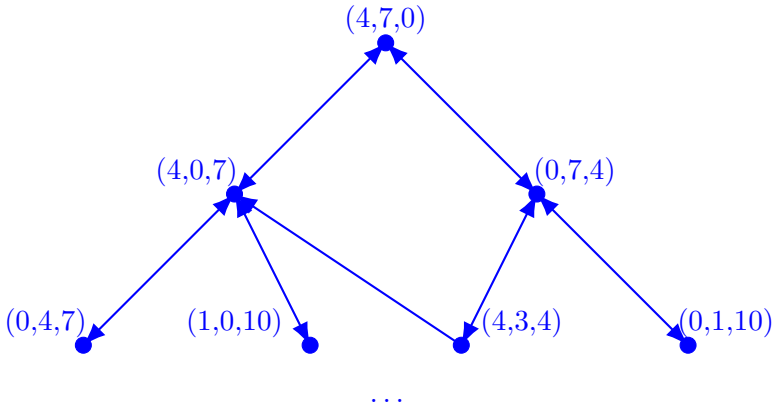
You are given three water jugs. The first one can contain 4L of water and is initially full, the second one can contain 7L and is also initially full, and the third one contains up to 10L of water but is initially empty. You can move the water between the jugs by pouring from one to the other until either (a) the jug being filled becomes completely full, or (b) the jug you are pouring from becomes empty. Can you pour the water between the containers in a way that you end up with exactly 2L of water in the first container?

(i) Model the question above as a graph problem. Your answer should include a specific directed graph, a description of what the vertices and the edges of your graph represent, and the problem that you want to solve on this graph to get the answer to the original question, as well as the answer itself.

Solution. We model the question as a problem on a directed graph $G = (V, E)$. We create a vertex for each possible configuration of the water jugs—the easiest way to do this is to label each vertex in our graph with three numbers corresponding to the amount of water in each jug at the time. Then the initial configuration corresponds to the vertex labelled $(4, 7, 0)$; other possible configurations include $(0, 7, 4)$, $(4, 3, 4)$, $(0, 3, 8)$, etc.

We create a directed edge from vertex (a, b, c) to vertex (a', b', c') if there is a way to pour one jug into another that goes from the first configuration to the second one. For example, we create an edge from $(4, 7, 0)$ to $(0, 7, 4)$ (for pouring jug 1 into jug 3), but there is no edge from $(4, 7, 0)$ to $(0, 3, 8)$ since no single pour lets us go from the first configuration to the second one.

The part of the resulting graph that includes all configurations reachable after at most 2 pours looks like this:



To determine the answer to the original question, we want to run a graph exploration algorithm on the resulting graph to see if we can reach any vertex labelled $(2, x, y)$ for some x, y from the original vertex $(4, 7, 0)$. We can do this, for example, by running DFS and implementing PreVisit to check whether the vertex is of the form $(2, x, y)$ and if so to set some global flag

`True` (initializing the flag to `False`, originally); once the DFS is complete, we return the value of that flag.

This algorithm will return the answer `True` (or `Yes`) to the original question. One path from the initial configuration to the configuration $(2, 7, 2)$ in the graph is

$$(4, 7, 0) \to (0, 7, 4) \to (0, 1, 10) \to (4, 1, 6) \to (0, 5, 6) \to (4, 5, 2) \to (2, 7, 2).$$

(ii) What graph algorithm lets you determine not just whether it is possible to end up with 2L of water in the first container, but the minimum number of pours needed to end up in this state (if it is reachable)?

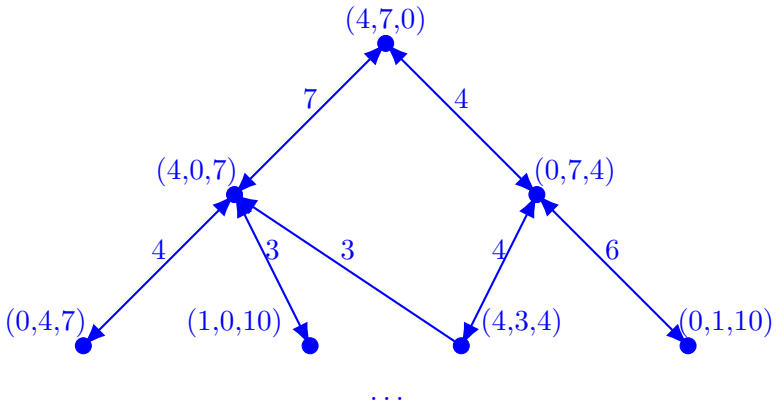
Solution. The minimum number of pours needed to end up in one of the configurations with 2L in the first container corresponds to the length of the shortest path between the vertex corresponding to the initial configuration $(4, 7, 0)$ and one of the $(2, x, y)$ configurations. Since we are solving this problem in an unweighted graph, we can use the BFS algorithm to build the BFS tree of the configuration graph (or, more precisely: of the connected component of this graph that includes the initial configuration) and find the minimum depth of a $(2, x, y)$ vertex in this graph.

The path described in part (i) is in fact a shortest path to a $(2, x, y)$ vertex, so the minimum number of pours required in our original question is 6.

(iii) Let's say each jug pours 1L of water per minute and we now want to know how quickly we can end up with 2L of water in the first container (if that is possible), how do we need to modify our graph representation and what algorithm do we want to run on the graph to solve this problem?

Solution. We model this variant of the problem using a weighted directed graph. The directed graph itself is as defined in part (i), and the weight of the edge (u, v) is defined to be the number of litres of water poured from one jug to another to go from configuration u to the configuration v.

Part of the resulting graph now looks like this:



To solve this problem, we now want to solve SSSP (single-source shortest path) on the graph with the vertex $(4, 7, 0)$ as our start vertex. Since all the edge weights are positive, we can use Dijkstra's algorithm to solve this problem. We then compare the shortest path lengths to each vertex $(2, x, y)$ and output the minimum length.

(The solution in part (i) shows that the minimum pour time is at most 24 minutes, but I don't know whether that's the optimal solution.)

5 Programming question [20 marks]

Alice and Bob plan to run a Dynamic Relay Race together. In this race, they must go from the startpoint 0 to the finish line n by travelling between checkpoints $0 \to 1 \to 2 \to \cdots \to n$ in order. Each segment $i \to i+1$ in the race is run by either Alice or Bob. The teammate who is not running is brought to the next checkpoint to wait for the runner to finish the segment. The rules of the race state that both Alice and Bob need to run exactly $n/2$ of the segments. (The organizers guarantee that n is always even.)

Alice and Bob know exactly how fast they can run each segment: Alice takes a_i minutes to cover the segment $i \to i+1$, Bob takes b_i minutes to run the same segment.[1] And for every checkpoint $1 \leq i < n$, there is a fixed penalty of p_i minutes incurred if they switch places (i.e., if Alice ran the segment $i-1 \to i$ and Bob runs segment $i \to i+1$ or vice-versa). Their goal is to determine who should run each segment to finish the Dynamic Relay Race as quickly as possible.

(i) Design a dynamic programming algorithm with time complexity $O(n^2)$ that finds the minimum time required for Alice and Bob to run the Dynamic Relay Race. As part of your algorithm description, you should include a precise definition of the subproblems solved by the algorithm as well as the recurrence relation and base case(s) used to solve the subproblems. (As usual, you also need to provide a proof of correctness and time complexity analysis.)

Solution.

Algorithm description. We obtain easier subproblems when we consider only the first $k+\ell$ segments of the race with Alice running k segments and Bob running ℓ segments (for each $k \leq n/2$ and $\ell \leq n/2$). But to implement this approach, we also need to consider one more factor in our subproblem: which of Alice or Bob ran the last segment.

Subproblems. For each $1 \leq k \leq n/2$ and $1 \leq \ell \leq n/2$, we define

$M_A(k,\ell) =$ minimum time required for Alice and Bob to run the first $k+\ell$ segments of the race, when Alice runs k of those segments and ran the last segment.

$M_B(k,\ell) =$ minimum time required for Alice and Bob to run the first $k+\ell$ segments of the race, when Alice runs k of those segments and Bob ran the last segment.

Base cases. The only base cases that we need are that for every $1 \leq k \leq \frac{n}{2}$, we define

$$M_B(k,1) = \sum_{i=0}^{k-1} a_i + p_k + b_k$$

and for every $1 \leq \ell \leq \frac{n}{2}$ we define

$$M_A(1,\ell) = \sum_{i=0}^{\ell-1} b_i + p_\ell + a_\ell.$$

[1] Alice and Bob are in fantastic shape, so that their time to run a segment is the same no matter how many segments they have already run previously.

For completeness, we could also define the additional base cases

$$\begin{aligned} M_A(0,0) &= 0, \\ M_B(0,0) &= 0, \\ M_A(k,0) &= \sum_{i=0}^{k-1} a_i \qquad \forall k \leq \tfrac{n}{2}, \text{ and} \\ M_B(0,\ell) &= \sum_{i=0}^{\ell-1} b_i \qquad \forall \ell \leq \tfrac{n}{2} \end{aligned}$$

but those base cases are not used in the algorithm.

Recurrence. For every $2 \leq k \leq n/2$ and $2 \leq \ell \leq n/2$,

$$M_A(k,\ell) = a_{k+\ell-1} + \min\{M_A(k-1,\ell), M_B(k-1,\ell) + p_{k+\ell-1}\}$$

and

$$M_B(k,\ell) = b_{k+\ell-1} + \min\{M_B(k,\ell-1), M_A(k,\ell-1) + p_{k+\ell-1}\}.$$

Pseudocode. The algorithm solves the subproblems defined above in increasing order of k and ℓ using the recurrence above.

Algorithm 2: DynRace($a_0, \ldots, a_{n-1}$, $b_0, \ldots, b_{n-1}$,$p_1, \ldots, p_{n-1}$)

```
for k = 1, ..., n/2 do
    M_B[k,1] ← Σ_{i=0}^{k-1} a_i + p_k + b_k;
for ℓ = 1, ..., n/2 do
    M_A[1,ℓ] ← Σ_{i=0}^{ℓ-1} b_i + p_ℓ + a_ℓ;
for k = 2, ..., n/2 do
    for ℓ = 2, ..., n/2 do
        M_A[k,ℓ] ← a_{k+ℓ-1} + min{M_A[k-1,ℓ], M_B[k-1,ℓ] + p_{k+ℓ-1}};
        M_B[k,ℓ] ← b_{k+ℓ-1} + min{M_B[k,ℓ-1], M_A[k,ℓ-1] + p_{k+ℓ-1}};
return min{M_A[n/2, n/2], M_B[n/2, n/2]};
```

Proof of correctness. The base cases $M_B(k,1)$ are correct because the only way that Alice can run k segments and Bob run 1 segment *and* we have Bob running the last segment is if Alice runs the first k segments (in time $\sum_{i=0}^{k-1} a_i$) then they switch places (with penalty time p_k) and Bob runs the last segment (in time b_k). The base cases $M_A(1,\ell)$ are correct by the same reasoning.

The recurrence for $M_A(k,\ell)$ is correct because there are exactly two possibilities to consider

(a) Alice runs the next-to-last segment. Then the minimal time to run the first $k+\ell-1$ segments under this condition is $M_A(k-1,\ell)$, Alice needs an extra $a_{k+\ell-1}$ minutes to run the last segment, and there is no penalty to add to that time because we don't change runners between the last two segments.

(b) Bob runs the next-to-last segment. Then the minimal time to run the first $k+\ell-1$ segments under this condition is $M_B(k-1,\ell)$, we need to pay a penalty of $p_{k+\ell-1}$ minutes to change runners from Bob to Alice, and Alice needs an extra $a_{k+\ell-1}$ minutes to run the last segment.

The fastest time to run $k+\ell$ segments with Alice running k of them and running the last segment is obtained by taking the minimum of the above two possibilities.

The correctness of the $M_B(k,\ell)$ recurrences is established with the same argument.

Time complexity analysis. Because we implemented the initializing loops in a naïve way (where the sums $\sum_{i=0}^{k} a_i$ and $\sum_{i=0}^{\ell} b_i$ are recomputed entirely in each loop iteration), they both run in time $\Theta(n^2)$. And the main loop of the algorithm runs $\Theta(n^2)$ times and does a constant amount of work at each iteration, so our overall time complexity is $\Theta(n^2)$.

(Note that we can optimize the inialization loops to avoid recomputing the sums at each iteration. This results in the initialization loops having time complexity $\Theta(n)$ each, but the overall time complexity of the algorithm remains $\Theta(n^2)$.)