CS 170 Algorithms Fall 2014 David Wagner — Final Soln

Problem 1. [True or false] (6 points)

Circle TRUE or FALSE. Do not justify your answers on this problem.

(a) [TRUE] or FALSE: If we perform DFS on an undirected graph, there are no cross edges.

(b) [TRUE] or FALSE: If the DFS tree has no back edges, then there are no cycles in the graph.

(c) [TRUE] or FALSE: Any tree with n vertices is guaranteed to have exactly $n-1$ edges.

(d) TRUE or [FALSE]: Suppose G is a directed graph $G=(V,E)$, and define the undirected graph G' by $G'=(V,E')$ where E' is the set of edges $\{u,v\}$ such that $(u,v)\in E$ or $(v,u)\in E$. Then, if G is *not* strongly connected, then it's guaranteed that G' is not connected.

(e) TRUE or [FALSE]: If G is a strongly connected directed graph, then there is guaranteed to exist a vertex such that deleting that vertex leaves you with a graph that is also strongly connected.

(f) TRUE or [FALSE]: Suppose we have a directed acyclic graph (dag) G with at least $n-1$ edges and where a topological sort yields the vertices $v_1,\ldots,v_n$, in that order. If we add the edge (v_n,v_1), then the resulting graph is guaranteed to be strongly connected.

(g) TRUE or [FALSE]: Dijkstra's algorithm will always work correctly on any graph that has at most two negative edges.

(h) [TRUE] or FALSE: If an undirected graph with n vertices has k connected components, then it must have at least $n-k$ edges.

(i) TRUE or [FALSE]: If a graph with integer lengths on the edges (some of them possibly negative) has no negative cycle reachable from s, then Dijkstra's algorithm works correctly for finding the length of the shortest path from s to every other vertex reachable from s.

(j) TRUE or [FALSE]: If we have a linear program where all of the coefficients and constants in every linear inequality are integers, then the optimum solution to this linear program must assign an integer value to every variable.

(k) TRUE or [FALSE]: If we have an undirected graph $G=(V,E)$ where $|V|=|E|$, then G is guaranteed to be a tree.

(l) TRUE or [FALSE]: If G is a graph with positive lengths on the edges, and if all edge lengths are distinct, and if s,t are two vertices, then there is a unique shortest path from s to t.

Problem 2. [Running time] (6 points)

Answer the following questions. No need to justify your answer. Use $\Theta(\cdot)$ notation.

(a) What is the asymptotic running time of decomposing a directed graph into strongly connected components, if the graph has n vertices and $n \lg n$ edges?

Solution: $\Theta(n \lg n)$.

(b) What is the asymptotic running time of Dijkstra's algorithm, on a graph with n vertices and m edges (assuming all edges have non-negative length), if we implement the priority queue with a binary heap?

Solution: $\Theta((n+m) \lg n)$.

We'll also accept $\Theta((n+m) \lg m)$.

(c) What is the asymptotic running time of the Bellman-Ford algorithm, on a graph with n vertices and m edges?

Solution: $\Theta(nm)$.

(d) What is the asymptotic solution to the recurrence $F(n) = 3F(n/2) + \lg n$?

Solution: $\Theta(n^{\lg 3})$.

Problem 3. [Short answer] (6 points)

Answer the following questions, giving a short justification (a sentence or two).

(a) Suppose we have already trained two classifiers. Classifier A is a k-nearest neighbors classifier with a fixed $k = 25$, using a linear search implementation (not a k-d tree). Classifier B is a random forest classifier with a fixed $T = 25$ (i.e., 25 decision trees), where each decision tree has depth at most 10. Now we want to know the running time to classify 100 test points. If the number of observations in the training set is very large, which classifier will be faster at this task? In other words, if we consider the asymptotic running time for this task as a function of n, which classifier will have an asymptotically faster running time?

Solution: Classifier B. The running time of B is fixed regardless of n, whereas the running time of A is proportional to n.

Comment: If there are d dimensions, Classifier A takes $\Theta(nd)$ time to linearly scan through all of the points and $\Theta(k \lg k) = \Theta(1)$ time to keep them in a heap and find the k closest. The running time of Classifier B depends only on the depth of the trees; if each tree has constant depth, its running time will be $\Theta(T) = \Theta(1)$.

(b) Let n be a power of two. Suppose you don't know the coefficients of a degree $n-1$ polynomial f, but you have its values at the nth roots of unity. How quickly could you find the value of f at some given other point x? In other words, what would the asymptotic running time be, as a function of n? You can assume that it takes $O(1)$ time to add or multiply two numbers.

Solution: $\Theta(n \lg n)$. Use the FFT to recover the coefficients of the polynomial (takes $\Theta(n \lg n)$ time), then evaluate it at x (takes $\Theta(n)$ time).

(c) In a strongly connected directed graph with integer weights (some of them possibly negative), is it possible that there are vertices s and t with no shortest path from s to t? Write "yes" or "no" and justify your answer in a sentence or two. Then, if you wrote "yes", name an algorithm we could use to detect whether this is the case; if you wrote "no", name an algorithm we could use to detect whether there are two vertices with no *longest* path from one to the other (i.e., the same task except with shortest replaced by longest).

Solution: Yes. If there is a negative cycle that contains s and t, then there is no shortest path from s to t: for any path from s to t you might have in mind, I can find one that is even shorter (by traversing the cycle an extra time). Bellman-Ford can detect presence of negative cycles.

Problem 4. [More running time] (2 points)

We have a directed graph $G = (V, E)$ with n vertices and m edges and integer lengths on all the edges. Only k of the edges have negative lengths; all the others have non-negative lengths. There are no negative cycles.

Your friend has a clever idea for computing the length of the shortest path from a vertex s to a vertex t. He will construct a new graph G' whose vertices are the endpoints of the negative edges of G, together with s and t. Each edge (u, v) in G' represents the length of the shortest path from u to v in G without using any negative-length edge. He'll use Dijkstra's algorithm to compute each of these lengths. Then, he'll add in all of the negative edges to G', and run Bellman-Ford on the result.

What is the asymptotic running time of your friend's algorithm, as a function of n and m and k? Give a short justification (a sentence or two).

Solution: $\Theta(k(n+m)\lg n + k^3)$. We do Dijkstra's algorithm $2k$ times, once for each endpoint of a negative edge, and that takes $\Theta((n+m)\lg n)$ time per endpoint; then Bellman-Ford takes $\Theta(|V| \times |E|) = \Theta(k \times k^2) = \Theta(k^3)$ time.

Comment: The most common mistake was to misunderstand the number of times we need to run Dijkstra's algorithm. We only need to run Dijkstra's algorithm once per vertex u in G'; this suffices to find the length of all of the edges (u, v) in G' (i.e., the distance from each u to each v, without using negative edges). Since there are $\Theta(k)$ vertices in G', we only need to run Dijkstra's algorithm $\Theta(k)$ times. There is no need to run Dijkstra's algorithm $\Theta(k^2)$ times: we only need to run it once per possible value of u, since Dijkstra's algorithm finds the distance from the starting point (u) to every other vertex (all possible v's).

Problem 5. [Algorithm design: sorted lists] (7 points)

Given two **sorted** lists of size m and n, we want to find the kth smallest element in the concatenation of the two lists. Fill in the blanks below so we get an algorithm whose running time is $O(\lg m + \lg n)$. Assume $k < m$ and $k < n$.

Find_kth_elem($A[1..m], B[1..n], k$):
1. Let $x := A[\lfloor k/2 \rfloor]$ and $y := B[\lceil k/2 \rceil]$.
2. If $x == y$:

3. Return $\boxed{x}$
4. If $x > y$:
5. Return Find_kth_elem($A[\boxed{1}..\boxed{\lfloor k/2 \rfloor}]$, $B[\boxed{\lceil k/2 \rceil + 1}..\boxed{k}]$, $\boxed{\lfloor k/2 \rfloor}$)
6. If $x < y$:
7. Return Find_kth_elem($A[\boxed{\lfloor k/2 \rfloor + 1}..\boxed{k}]$, $B[\boxed{1}..\boxed{\lceil k/2 \rceil}]$, $\boxed{\lceil k/2 \rceil}$)

Now, answer the following question. Which algorithm design paradigm does this algorithm best represent?

1. Divide-and-conquer
2. Greedy algorithm
3. Dynamic programming
4. Linear programming
5. Reduce to network flow
6. None of the above

Comment: For line 5, it's also OK to put

5. Return Find_kth_elem($A[\boxed{1}..\boxed{m}]$, $B[\boxed{\lceil k/2 \rceil + 1}..\boxed{k}]$, $\boxed{\lfloor k/2 \rfloor}$)

We even gave credit for

5. Return Find_kth_elem($A[\boxed{1}..\boxed{m}]$, $B[\boxed{\lceil k/2 \rceil + 1}..\boxed{n}]$, $\boxed{\lfloor k/2 \rfloor}$)

and symmetrically for line 7.

The important thing was that on line 5, you should discard the first $k/2$ elements of B, and reduce the third parameter to $k/2$. Similarly, on line 7, you should discard the first $k/2$ elements of A, and reduce the third parameter to $k/2$. We did not worry about floors or ceilings or off-by-one errors.

Note that our original problem was missing a base case. There ought to be a line 0 that contains a base case: e.g., if $k = 1$ or $k > 2m$ or $k > 2n$, then solve this separately using some other method.

Problem 6. [Linear programming: space radar] (7 points)

A set of n space stations need your help in building a radar system to track spaceships traveling between them. The ith space station is located in 3D space at coordinates (x_i, y_i, z_i). The space stations never move. Each space station will have a radar of some power, say r_i for the ith space station, where r_i is to be determined.

You want to figure how powerful to make each space station's radar transmitter, so that whenever any spaceship travels in a straight line from one space station to another, it will always be in radar range of either the first space station (its origin) or the second space station (its destination). A

radar with power r is capable of tracking space ships anywhere in the sphere with radius r centered at itself. Thus, a space ship is within radar range through its trip from space station i to space station j if every point along the line from (x_i, y_i, z_i) to (x_j, y_j, z_j) falls within either the sphere of radius r_i centered at (x_i, y_i, z_i) or the sphere of radius r_j centered (x_j, y_j, z_j). The cost of each radar transmitter is proportional to its power, and you want to minimize the total cost of all of the radar transmitters.

You are given all of the $(x_1, y_1, z_1), \ldots, (x_n, y_n, z_n)$ values, and your job is to choose values for $r_1, \ldots, r_n$. Express this problem as a linear program.

(a) Describe your variables for the linear program.

Solution: $r_i =$ the power of the ith radar transmitter.

(b) Write out the objective function.

Solution: $r_1 + r_2 + \cdots + r_n$.

(c) Between each pair of space stations, say station i and station j, we need one constraint (one linear inequality). What is it?

Solution: $r_i + r_j \geq \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2}$.

Or, $r_i + r_j \geq d_{i,j}$, where $d_{i,j}$ is the distance from station i to station j.

Problem 7. [Algorithm design: pizza] (6 points)

Our spaceship's cook has baked a zero-gravity pizza and cut it into n slices, but the lack of gravity made him clumsy and the pizza wasn't evenly sliced: the n slices have size $s_1, s_2, \ldots, s_n$. There are n hungry space travelers on board who each want to eat a slice of pizza. Suppose the ith traveler would be happy with any slice whose size is at least t_i. Give an efficient algorithm to determine whether it is possible to distribute the pizza slices so everyone is happy.

(a) What algorithm design paradigm is most appropriate for this problem? Circle one of the following.

(a) Divide-and-conquer
(b) Greedy algorithm
(c) Dynamic programming
(d) Linear programming
(e) Reduce to network flow
(f) None of the above

(b) Show pseudocode for your algorithm. (No proof of correctness needed.)

Solution:

1. Sort $s_1, \ldots, s_n$ into increasing order.
2. Sort $t_1, \ldots, t_n$ into increasing order.
3. If there is any i such that $s_i < t_i$, return "Not possible." Otherwise, return "Possible."

(c) What is the asymptotic running time of your algorithm? You don't need to justify your answer.

Solution: $\Theta(n \lg n)$.

Comment: This is a greedy algorithm because the idea is to assign the smallest slice to the person who wants the least, the second-smallest slice to the person who wants the second-least, and so on. Then, we check whether everyone is happy.

Comment: We'll also give partial credit for circling "Reduce to network flow", if you constructed a bipartite graph on $2n$ vertices (one vertex per traveller, one vertex per slice, with an edge from each traveller to each slice he/she would be happy with), and then looked for a perfect bipartite matching using network flow (treat each edge as capacity 1, check whether the max flow is n).

We might also give partial credit for circling "Divide-and-conquer" and giving the sorting-based pseudocode above, using merge sort as your sorting algorithm (maybe you were thinking that merge sort uses divide-and-conquer, though that's really the paradigm of merge sort rather than the paradigm used by the algorithm above).

Problem 8. [Algorithm design: sums] (7 points)

You're given a positive integer K and a set of n different positive integers $A = \{a_1, a_2, \ldots, a_n\}$. Your job is to determine whether there exists two sets S, T such that (1) S and T are subsets of A, (2) S and T are disjoint (i.e., $S \cap T = \emptyset$), (3) $S \cup T = A$, and (4) $(\sum_{s \in S} s) - (\sum_{t \in T} t) = K$.

Design an efficient algorithm for this problem.

(a) What algorithm design paradigm is most appropriate for this problem? Circle one of the following.

(a) Divide-and-conquer
(b) Greedy algorithm
(c) Dynamic programming
(d) Linear programming
(e) Reduce to network flow
(f) None of the above

(b) Show pseudocode for your algorithm. (No proof of correctness needed.)

Solution:

1. Set $D_0 := \{0\}$ (the empty set).
2. For $j := 1, 2, \ldots, n$:
3. Set $D_j := \{d + a_j : d \in D_{j-1}\} \cup \{d - a_j : d \in D_{j-1}\}$.
4. Return true if $K \in D_n$, else return false.

Comment: Here's where this came from. Let $D(j)$ be the set of difference values that can be attained using $\{a_1, \ldots, a_j\}$. (In other words, $d \in D(j)$ if there exists two sets S, T that satisfy conditions (1)–(3) and such that $\sum_{s \in S} s - \sum_{t \in T} t = d$.) Then we have the recursive formula

$$D(j) = \{d \pm a_j : d \in D(j-1)\}.$$

That leads to the pseudocode shown above.

Another way to solve this is to let $f(j,d) =$ true if there is a way to achieve the difference d using $\{a_1, \ldots, a_j\}$. Then we have the recursive formula

$$f(j,d) = f(j-1, d+a_j) \vee f(j-1, d-a_j).$$

So, the following pseudocode is another way to solve the problem:

1. Set $A[0,d] :=$ false for $d := -L, \ldots, -1, 1, \ldots, L$. Set $A[0,0] :=$ true.
2. For $j := 1, 2, \ldots, n$:
3. Set $t := a_1 + \cdots + a_j$.
4. For $d := -t, -t+1, \ldots, t-1, t$:
5. Set $A[j,d] := A[j-1, d+a_j] \vee A[j-1, d-a_j]$.
6. Return $A[n,K]$.

Alternative solution: Let $L = a_1 + a_2 + \cdots + a_n$. Then the answer to the original problem is yes if and only if there exists a subset $S \subseteq A$ such that $\sum_{s \in S} s = (K+L)/2$. (Why? Define $x = \sum_{s \in S} s$. Then $\sum_{t \in T} t = L - x$. We want $x - (L-x) = K$, but this is equivalent to $x = (K+L)/2$.)

So this problem reduces to subset sum (i.e., knapsack without repetition, where the weight is equal to the value), and can be solved with the dynamic programming algorithm we've seen in this class. For example,

1. Set $A[0,t] :=$ false for $t = 1, 2, \ldots, L$ and $A[0,0] :=$ true.
2. For $i := 1, 2, \ldots, n$:
3. For $t := 0, 1, \ldots, L$:
4. Set $A[i,t] := A[i-1, t] \vee A[i-1, t-a_i]$.
5. Return $A[n, (K+L)/2]$.

Comment: We gave no credit on part (b) for algorithms whose running time is exponential in n (even after memoization). For instance, algorithms that involved trying all subsets of A take at least 2^n time and thus got no credit. As another example, algorithms where the arguments to the function were sets typically fell into this category, as did dynamic programming approaches where the subproblems could include all possible subsets of A. (One way to avoid this was to have the argument(s) be an integer that holds the sum of the elements in the set, rather than using the set itself as the argument—this way the number of subproblems solved will be $\Theta(L)$ rather than $\Theta(2^n)$.)

Comment: Linear programming is not correct. This can be modelled as an *integer* linear program, by introducing integer variables $x_1, \ldots, x_n$ where $x_i = 1$ if $a_i \in S$ and $x_i = 0$ if $a_i \in T$. We add the constraints $0 \le x_i \le 1$ and $a_1(2x_1 - 1) + \cdots + a_n(2x_n - 1) = K$, and check for a

feasible solution. (Here $2x_i - 1$ converts from 0 or 1 to -1 or $+1$.) However, this requires integer linear programming (which is NP-hard, sadly). It cannot be solved this way with linear programming, because a LP solver might find an assignment where each x_i is fractional, and then there's no way to interpret that as a solution to the original problem.

(c) What is the asymptotic running time of your algorithm from part (b)? Write your answer as a function of n, K, and/or L, where we define $L = a_1 + a_2 + \cdots + a_n$. You don't need to justify your answer.

Solution: $O(nL)$.

Comment: For the first solution, each set D_j can have at most $2L+1$ elements, since every element of each set is in the range $-L..L$. Therefore, line 3 runs in $O(L)$ time. We do n iterations of the loop, so the total running time is $O(nL)$.

For the second solution, line 5 runs in $O(1)$ time. We do at most $2L+1$ iterations of the loop in line 4 (since $t \leq L$), and n iterations of the loop in line 2, so the total running time is $O(nL)$.

Problem 9. [Algorithm design: knapsack] (7 points)

Consider the discrete knapsack problem where all items have the same value 42. In other words, we have n items, where w_i is the weight of the ith item and every item has value 42, and we have a knapsack of capacity C. (This is knapsack without repetition: each item can be chosen at most once.) We want to find a set of items whose total weight is at most C and whose total value is as large as possible.

Devise an efficient algorithm for this problem, then answer the following two questions about it.

(a) What algorithm design paradigm is most appropriate for this problem? Circle one of the following.

(a) Divide-and-conquer
(b) Greedy algorithm
(c) Dynamic programming
(d) Linear programming
(e) Reduce to network flow
(f) None of the above

(b) What is the asymptotic running time of your algorithm? Justify your answer in a sentence or two.

Solution: $\Theta(n \lg n)$: $\Theta(n \lg n)$ to the sort the items, then a linear scan to take the lightest ones first (takes $O(n)$ time).

Comment: The algorithm is: sort them by increasing weight. Scan through them, in order of increasing weight, and take each one that's available until your knapsack is full.

Comment: Alternatively, you could solve this using dynamic programming. Define $f(i,w) =$ the maximum number of items you can choose (out of the first i items) using total weight at most w. Then $f(i,w) = \max(f(i-1,w-w_i)+1, f(i-1,w))$. Consequently, the running time will be $\Theta(nC)$. However, this can potentially be much slower than $\Theta(n \lg n)$. In particular, the value of C might be exponential in the length of the input (suppose that the capacity C is a 100-bit number; then $C \approx 2^{100}$, so this algorithm will be slow in that case).

Problem 10. [Algorithm design: phone numbers] (4 points)

You're the manager for the phone company on board a new space station, and you're trying to assign numbers to various services. To increase dialing speed, you want a call to be placed as soon as a valid number is dialed. For example, if 911 is the number for emergency services, then entering those three digits starts the call immediately, so you can't assign the number 9113 to some other service, since anyone trying to dial it would end up calling 911.

There are n phone services. You have statistics for each service from other space stations; you know that the ith service is used on average f_i times per year. Given $f_1, \ldots, f_n$, you want to assign phone numbers to the services (using only the digits 0–9) so that services which are used more often will have shorter numbers. In particular, you want to minimize the average number of digits dialed (with the average taken over all calls to a phone service).

There's an algorithm we've seen in class that is perfect for solving this problem, if you make a modification. Which one? What modification do you need to make?

Solution: Huffman coding. We need to modify it to use a tree with branching factor 10, instead of a binary tree.

Comment: Actually modifying the algorithm is a little tricky: the base cases need to be handled carefully. Consider, for instance, a case where we have 11 phone services: it's not optimal to first take the 10 least-commonly used ones and group them together as children of a single node (giving them all two-digit phone numbers). However, you didn't need to explain how to deal with this situation.

Problem 11. [Algorithm design: catch the thief] (7 points)

Oh no! Someone has stolen the draft of Rohit's paper proving $\mathsf{P} = \mathsf{NP}$. The thief is trying to get out of Berkeley before getting caught. The Berkeley police have asked you to help them block the thief's escape.

The roads and intersections are represented as an unweighted directed graph $G = (V,E)$. We're also given a subset $C \subseteq V$ of possible current locations of the thief's car and a set $P \subseteq V$ of all exit points out of Berkeley. The police want to set up roadblocks on a subset of the edges to prevent the thief from escaping (i.e., from reaching one of the exit points). Clearly the police could just put a roadblock at all the edges to each exit point, but there might be a better way: for example, if the thief is known to be at a particular intersection, they could just block all roads coming out of it. You may assume that roadblocks can be set up instantaneously.

Design an efficient algorithm to find a way to stop the thief using the smallest possible number of roadblocks.

(a) What algorithm design paradigm is most appropriate for this problem? Circle one of the following.

(a) Divide-and-conquer
(b) Greedy algorithm
(c) Dynamic programming
(d) Linear programming
(e) Reduce to network flow
(f) None of the above

(b) Describe your algorithm. (You don't need to prove it correct.)

Solution: Add a source node s, with an edge from s to each vertex in C. Add a sink node t, with an edge from each vertex in P to t. Make each edge from the original graph have capacity one, and each new edge have capacity ∞. Now find a minimum (s,t)-cut $(S, V-S)$, and place a roadblock on each edge that crosses the cut in the $s \to t$ direction, i.e., on each edge $(u,v) \in E$ such that $u \in S$ and $v \notin S$. We can find the minimum cut by computing the max flow and then using the max-flow min-cut theorem.

(c) What is the asymptotic running time of your algorithm in part (b)? Your answer can be a function of $|C|$, $|P|$, $|V|$, and/or $|E|$. You don't need to justify your answer.

Solution: $\Theta(|E|^2)$.

Comment: Ford-Fulkerson's running time is $\Theta(f \cdot |E|)$, where f is the value of the maximum flow. In this case, the value of the max flow must be at most $|E|$, so we get a $\Theta(|E|^2)$ running time.

Alternative solution: $\Theta(|V| \cdot |E|)$.

Comment: Use the fancy algorithm that can compute the max flow in an arbitrary graph in $O(|V| \cdot |E|)$ time: Orlin's algorithm plus the King-Rao-Tarjan algorithm.

Other variants, like $\Theta(|V|(|E|+|P|+|C|))$ or $\Theta((|E|+|P|+|C|)^2)$, are also reasonable. However $|E|+|P|+|C| = \Theta(|E|)$ [assuming the graph is connected, or at least that $|V| \leq |E|$], as $|E|+|P|+|C| \leq |E|+|V|+|V| \leq 3|E|$, so the answer above are just as good.

Problem 12. [Algorithm design: maze] (7 points)

You've landed on an alien planet, in the middle of some strange maze, and you want to find the quickest way out.

The maze can be thought of as a $m \times n$ grid where some of the squares in the grid are blocked. You know the entire maze (including the size of the grid, which squares are blocked, and your current location). You can drive north, south, west, or east from each square in your hovercar, but not diagonally.

Unfortunately, your hovercar was damaged during the rough landing onto the planet, and it can't turn left or make U-turns: the hovercar can only go straight or turn right. For instance, if you leave

the current square by going north, you'll only be able to leave the next square by going north or east; west is not possible, because that would require a left turn, and south is not possible either, because would require a U-turn. Your hovercar was landed facing north, so on your first move you will need to drive north.

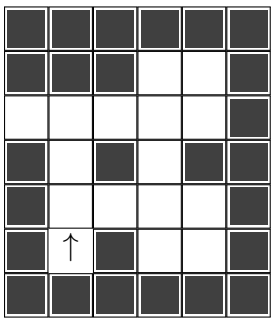
Find an efficient algorithm to find the shortest route to escape from the maze. We want you to do this by creating an appropriate directed graph G and then using breadth-first search on G. Answer each of the following questions. You don't need to justify any of your answers.

(a) Explain how to create the graph G.

Solution: Have four vertices for each grid square, according to the four directions (N, E, S, W) by which you entered that grid square.

Add an edge from (s,d) to (s',d') if (1) you can get from grid square s to grid square s' by moving in direction d', (2) grid square s' is not blocked, and (3) either $d' = d$ or $d' = 90$ degrees clockwise from d.

Comment: The most common mistake was to try to have one vertex per grid square (instead of four). Some people added edges between all pairs of adjacent grid squares; some tried to do some sort of DFS to find out in which directions it is possible to leave each grid square, and added edges accordingly. However, none of the solutions of this form worked correctly. For instance, consider the following grid layout:



The hovercar starts at the square marked with the arrow (↑). The dark grid squares are blocked. If you have one vertex per non-blocked (white) grid square, and edges between adjacent squares, then BFS will find a path of length 4 (go north 3, then west 1). However, this path does not correspond to a valid route you could drive in the hovercar, because it requires turning left. The shortest way out involves a more complex path (go north 1, east 3, south 1, west 1, north 4, east 1, south 1, and finally west 4). Therefore, if your graph involves one vertex per grid square, it will most likely not work correctly with the above example.

We found similar counterexamples for several variations on this idea. To get a graph that works, you need to take into account the directions you can travel by making 4 copies of each vertex, one for each direction.

(b) How many vertices does your graph G have?

Solution: $4mn$.

(c) How many edges does your graph G have? (at most)

Solution: At most $8mn$.

(d) What is the asymptotic running time of the resulting algorithm, as a function of m and n?

Solution: $\Theta(mn)$.

Comment: BFS runs in $\Theta(|V|+|E|)$ time. Here $|V| = \Theta(mn)$ and $|E| = \Theta(mn)$, so BFS's running time is $\Theta(mn)$.

Problem 13. [Stock trading] (7 points)

By exploiting a handy wormhole to travel through time, Daniel knows what the stock price of Intergalactic Starships, Inc. will be on every day over the next n days: on day i, each share of stock costs p_i. Daniel starts out with \$1000. He plans to pick one day to buy as many shares of stock as he can afford and then sell them all on some subsequent day. Design an efficient algorithm to help Daniel figure out what's the maximum profit he can achieve. (Warning: Stock prices for starship companies can fluctuate a lot. There is no guarantee that the day with the lowest stock price will come before the day with the highest stock price.)

(a) Show your pseudocode. You don't need to prove your pseudocode correct.

Solution:

1. Set $S_0 := 0$.
2. For $i := 1, 2, \ldots, n$:
3. Set $S_i := \max(S_{i-1}, 1000/p_i)$.
4. Return $\max(S_0 p_1, S_1 p_2, S_2 p_3, \ldots, S_{n-1} p_n) - 1000$.

Comment: We're using dynamic programming. Here S_i = the maximum number of shares that Daniel can own at the end of day i, by issuing a buy order on one of the first i days. Therefore $S_{j-1} p_j - 1000$ is the maximum amount of profit Daniel can possibly make if he sells on day j.

This could be further optimized to

1. Set $s := 0$ and $f := -\infty$.
2. For $i := 1, 2, \ldots, n$:
3. Set $f := \max(f, s \cdot p_i)$ and $s := \max(s, 1000/p_i)$.
4. Return $f - 1000$.

The running time is the same but we've reduced the space complexity.

Alternative solution: We could use divide-and-conquer. Define the following procedure:

Profit($x..z$):
1. If $z = x$, return 0. If $z = x+1$, return $p_z 1000/p_x - 1000$.
2. Set $y := \lfloor (x+z)/2 \rfloor$.
3. Set $\alpha = \max(p_{y+1}, \ldots, p_z) 1000 / \min(p_x, \ldots, p_y) - 1000$.
4. Return $\max(\text{Profit}(x..y), \text{Profit}(y+1..z), \alpha)$.

The answer to the original can be obtained by running Profit$(1..n)$.

The idea is that Profit$(x..z)$ computes the maximum profit that can be obtained if Daniel buys at some day in the range $x, x+1, \ldots, z$ and then sells at some later day (also required to be in the range $x, x+1, \ldots, z$). Why does this work? Well, there are three cases for when Daniel could consider buying and selling:

- Daniel buys on some day in the range $x..y$ and sells on some day in the range $x..y$. The best profit attainable in this way is computed by Profit$(x..y)$.
- Daniel buys on some day in the range $x..y$ and sells on some day in the range $y+1..z$. The best profit attainable in this way is exactly α, since Daniel's best strategy is to buy on the day in $x..y$ when the stock price is lowest and sell on the day in $y+1..z$ when the stock price is highest.
- Daniel buys on some day in the range $y+1..z$ and sells on some day in the range $y+1..z$. The best profit attainable in this way is computed by Profit$(y+1..z)$.

The algorithm above considers all three cases, and those are the only three possible cases.

Comment: It's not enough to find the index i of the day when the stock price is cheapest, and the index j of the day when the stock price is largest, and buy on day i and sell on day j. Consider the sequence of prices $5, 4, 3$: here we have $i = 3$, $j = 1$, but we can't buy on day 3 and sell on day 1 (we can't sell any shares until after we've bought some).

Finding the pair of indices $i < j$ such that $p_j - p_i$ is as large is possible does not yield the correct solution. Consider the sequence of prices $1000, 1100, 1, 5$. The difference is maximized if we choose $i = 1$, $j = 2$; then we get the difference $p_2 - p_1 = 1100 - 1000 = 100$. However, that gives Daniel a profit of only \$100, since he buys one share on day 1 and then sells it on day 2. If he waits to buy until day 3 and then sells on day 4, Daniel makes \$4000 of profit.

(b) What is the asymptotic running time of your algorithm? Justify your answer concisely.

Solution: $\Theta(n)$. (for the dynamic programming algorithm above)

Or, $\Theta(n \lg n)$ for the divide-and-conquer algorithm above.

(c) Now suppose Daniel is willing to do at most k trades, where each trade constitutes buying as many shares as he can afford on some day, or selling all his shares on some day. Daniel wants an efficient algorithm to calculate the maximum profit he can achieve.

What algorithm design paradigm is most appropriate for this problem? Circle one of the following.

(a) Divide-and-conquer
(b) Greedy algorithm
(c) Dynamic programming
(d) Linear programming
(e) Reduce to network flow
(f) None of the above

You do not need to describe your algorithm; just answer the above question.

Comment: The details are a little messy, but the basic idea is to work out the maximum number of shares Daniel might have on day i if he has done at most j trades, and to work out the maximum number of dollars Daniel might have on day i if he has done at most j trades. These can be computed using dynamic programming.

Divide-and-conquer would not be a good choice: it will be inefficient. You would have to consider all possible ways to write $k = k_1 + k_2$ (where $0 \le k_1, k_2 \le k$), and for each such way, do two recursive calls to find the maximum amount of profit that Daniel can make by doing k_1 trades in the first $n/2$ days and k_2 trades in the last $n/2$ days. In this way, each invocation will end up doing $2k$ recursive calls (each of which will in turn make recursive calls, and so on), so the resulting algorithm would be significantly slower than the dynamic programming solution.

Problem 14. [Algorithm design: wormholes] (7 points)

The inhabitants of a far off star system have developed a transportation system between their planets. There are n planets in the system. Some planets have a wormhole between them, which allows you to travel between planet i to planet j in $t_{i,j}$ seconds. All times $t_{i,j}$ are positive (you can't travel back in time). Some pairs of planets may not have a wormhole between them.

You are currently at planet 0 and wish to reach planet $n-1$. However, your ship's navigation commands are starting to malfunction. When you are at a planet and tell the ship to go through a certain wormhole, it may misinterpret the command and instead choose a completely random outgoing wormhole instead. Luckily, you know that you that this will happen at most once. You don't know when the malfunction might happen, but you are going to assume it will happen at the most inconvenient time.

You want to choose a strategy that will minimize your total travel time in the very worst case (no matter when the malfunction happens or which outgoing wormhole it takes you on). Design an efficient algorithm for this. Specifically, your algorithm should output at what time you can be guaranteed to reach planet $n-1$, if we use the optimal strategy.

You don't need to prove your algorithm correct or justify the stated running time. You may assume it's possible to reach planet $n-1$ from any other planet via some sequence of wormholes.

(a) Describe your main idea, in enough detail that another CS 170 student could implement your algorithm. (No pseudocode or proof of correctness needed.)

Solution: Define the following quantities:

$$\begin{aligned}
d(v) &= \text{shortest possible time to get from } v \text{ to } n-1, \text{ assuming no malfunctions} \\
m(v) &= \text{shortest time to get from } v \text{ to } n-1, \text{ if there is a malfunction happens at } v \\
&= \max\{t_{v,w} + d(w) : (v,w) \in E\} \\
f(v) &= \text{shortest time to get from } v \text{ to } n-1, \text{ if the malfunction might occur anywhere} \\
&= \max(m(v), \min\{t_{v,w} + f(w) : (v,w) \in E\}). \\
&= \min\{\max(m(v), t_{v,w} + f(w)) : (v,w) \in E\}.
\end{aligned}$$

We can compute the $d(\cdot)$ values using Dijkstra's algorithm backwards from t, and then the $m(\cdot)$ values using a linear scan.

We can compute the $f(\cdot)$ values using a modified Dijkstra's algorithm backwards from t, where the priority queue is keyed by $f(\cdot)$-value and each edge (v,w) is processed like this:

1. If $\max(m(v), t_{v,w} + f(w)) < f(v)$:
2. Set $f(v) := \max(m(v), t_{v,w} + f(w))$. Decreasekey($H$, v).

Comment: If we were promised that the graph was a dag, we could topologically sort the graph, scan through the vertices in reverse topologically sorted order, and compute the $f(\cdot)$ values directly. However, there was no promise that we'd have a dag (there might be cycles), which is what requires the more complex procedure above.

(b) What's the asymptotic running time of your algorithm from part (a), if there are n planets and m wormholes? No justification needed.

Solution: $\Theta((n+m)\lg n)$.

Comment: Basically, we run Dijkstra's algorithm twice.

Problem 15. [Algorithm design: disease] (7 points)

You are a government bureaucrat responsible for preventing some new disease from spreading from planet to planet through the galactic empire and making its way to the galactic capital. You have the power to require medical screening on any travel routes between the planets you want. The goal is to find a minimal subset of routes to ensure any new disease will be detected before it can reach the capital, without subjecting anyone to medical screening twice.

We'll represent this as a graph problem: we have a directed acyclic graph $G = (V,E)$ with a source vertex s and sink vertex t. Call a set U of edges *unavoidable* if every path from s to t has exactly one edge that's in U (at least one edge of the path is in U, but the path doesn't contain more than one edge that's in U). Your job is to design an efficient algorithm to find the smallest unavoidable set U possible.

(Each vertex other than s represents a planet, t is the galactic capital, and s is an artificial vertex with an edge from s to every planet where a new infectious disease might arise.)

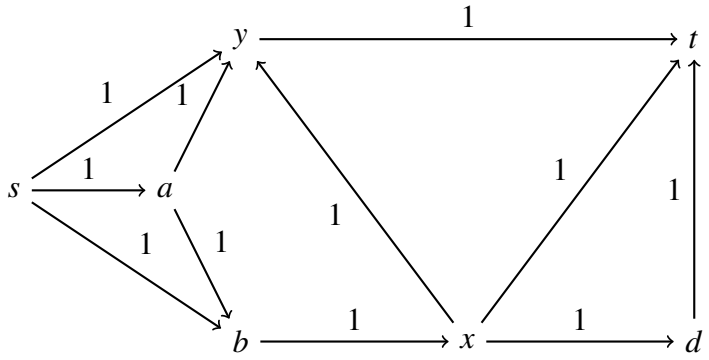
(a) Suppose we treat every edge of G as having capacity 1, find a minimum (s,t)-cut $(S, V-S)$ for this graph, and let $T = \{(u,v) \in E : u \in S, v \notin S\}$ be the set of edges that cross this cut in the $s \to t$ direction. Is T guaranteed to be an unavoidable set? Why or why not?

Solution: No. The path might leave S, then come back into S, then leave S a second time, thus traversing two edges from T.

Alternative solution: No. Consider the graph below: $S = \{s,a,b,y\}$ is a minimum cut, but the path $s \to b \to x \to y \to t$ has two edges in T (the $b \to x$ edge and the $y \to t$ edge).

(b) Consider the graph G shown below. Suppose we want to find the minimum (s,t)-cut $(S, V-S)$ in G such that either $x \in S$ or $y \notin S$ (i.e., such that (x,y) doesn't cross the cut in the $t \to s$

direction). This can be done by adding one edge to get a new graph G', giving the new edge an appropriate capacity, and then finding a minimum (s,t)-cut in G'. Draw below the edge you should add to guarantee that this approach will give a correct answer, and label it with its capacity.



Solution: Add an edge from y to x with capacity ∞ (or very large).

(c) We want an efficient algorithm for the following problem: given a dag G, source vertex s, and sink vertex t, find an unavoidable set U containing as few edges as possible.

What algorithm design paradigm is most appropriate for this problem? Circle one of the following.

(a) Divide-and-conquer

(b) Greedy algorithm

(c) Dynamic programming

(d) Linear programming

(e) Reduce to network flow

(f) None of the above

(d) Show pseudocode for the problem in part (c). (You don't need to prove your answer correct.)

Solution:

1. Make all edges in E have capacity 1.
2. For each edge $(u,v) \in E$, add the edge (v,u) with capacity ∞.
3. Find the max flow in the resulting graph, and let $(S, V-S)$ be a minimum (s,t)-cut.
4. Return $U = \{(u,v) \in E : u \in S, v \notin S\}$.

(e) What is the asymptotic running time of your algorithm? Justify your answer concisely (a sentence or two should suffice).

Solution: $\Theta(|V| \cdot |E|)$, since the value of the max flow is at most $|V|$ (at most the number of edges out of s).

Comment: Ford-Fulkerson's running time is $\Theta(f \cdot |E|)$, where f is the value of the maximum flow. In this case, the value of the max flow must be at most $|V|$ (at most the number of edges out of s), so we get a $\Theta(|V| \cdot |E|)$ running time.

Alternatively, we can use the fancy algorithm that can compute the max flow in an arbitrary graph in $O(|V| \cdot |E|)$ time: Orlin's algorithm plus the King-Rao-Tarjan algorithm.

Problem 16. [Algorithm design: rockets] (7 points)

You're leading an exploration of space, and it is your job to design a spaceship that will go as far as possible. To do this, you have n available rockets. The ith rocket is capable of accelerating the ship by a rate of a_i meters per second squared and can do this for a total of s_i seconds. The ship is initially at rest (its initial velocity is zero).

Your job is to determine the order in which to fire the rockets. You will choose one rocket, fire it until s_i seconds have elapsed, then immediately switch to another unused rocket. Each rocket can only be used once, and you may ignore the time it takes to switch between rockets.

Given the accelerations and durations of each rocket, design an efficient algorithm to determine the optimal order to fire the rockets, to maximize the total distance traveled.

Physics background: if our velocity is v before we start firing a single rocket whose acceleration is a then, after t seconds, we will have traveled a total $vt + \frac{1}{2}at^2$ meters during that time, and our final velocity will be $v' = v + at$.

(a) Suppose we have two rockets, with $a_1 = 2, s_1 = 1, a_2 = 3, s_2 = 2$. How far do we travel if we fire rocket 1 then rocket 2?

Solution: $[\frac{1}{2} \cdot 2 \cdot 1^2] + [2 \cdot 1 \cdot 2 + \frac{1}{2} \cdot 3 \cdot 2^2] = 11$ meters.

How far do we travel if we fire rocket 2 then rocket 1?

Solution: $[\frac{1}{2} \cdot 3 \cdot 2^2] + [3 \cdot 2 \cdot 1 + \frac{1}{2} \cdot 2 \cdot 1^2] = 13$ meters.

(b) (Part (b) is optional. You can skip it. Only do it if you get stuck on parts (c) and (d) and need a hint.)

Suppose we have two rockets, with $a_1 = 6, s_1 = 3, a_2 = 4, s_2 = 7$. How far do we travel if we fire rocket 1 then rocket 2? Leave your answer as an unevaluated expression.

Solution: $\frac{1}{2} \cdot 6 \cdot 3^2 + 6 \cdot 3 \cdot 7 + \frac{1}{2} \cdot 4 \cdot 7^2 = 251$ meters.

How far do we travel if we fire rocket 2 then rocket 1? Leave your answer as an unevaluated expression.

Solution: $\frac{1}{2} \cdot 4 \cdot 7^2 + 4 \cdot 7 \cdot 3 + \frac{1}{2} \cdot 6 \cdot 3^2 = 209$ meters.

Circle the terms in common to both answers. Can they be ignored for our purposes?

Solution: The $\frac{1}{2} \cdot 6 \cdot 3^2$ and $\frac{1}{2} \cdot 4 \cdot 7^2$ terms appear in both. Yes, we can ignore them: since we only compare about which of those two sums is larger, we can cancel them both out from both sums. We're basically comparing $a_1 s_1 s_2 = 6 \cdot 3 \cdot 7$ to $a_2 s_2 s_1 = 4 \cdot 7 \cdot 3$.

(c) What algorithm design paradigm is most appropriate for finding the optimal order to fire the rockets? Circle one of the following.

(a) Divide-and-conquer
(b) Greedy algorithm
(c) Dynamic programming
(d) Linear programming
(e) Reduce to network flow
(f) None of the above

(d) Design an algorithm to determine the optimal order to fire the rockets. Show your pseudocode. (You don't need to provide anything other than your pseudocode.)

Solution:

1. Sort the rockets by decreasing acceleration (decreasing a_i).

Comment: In general, we can ignore all the $\frac{1}{2}at^2$ terms, so the total distance traveled is

$$a_1s_1s_2 + (a_1s_1 + a_2s_2)s_3 + \cdots + (a_1s_1 + \cdots + a_{n-1}s_{n-1})s_n.$$

Equivalently, this is

$$a_1s_1(s_2 + s_3 + \cdots + s_n) + a_2s_2(s_3 + s_4 + \cdots + s_n) + \cdots + a_{n-1}s_{n-1}s_n.$$

If we swap rockets i and $i+1$, the distance traveled increases by $a_{i+1}s_{i+1}s_i - a_is_is_{i+1}$ (all the other terms cancel). Thus, swapping rockets i and $i+1$ increases the total distance traveled if $a_{i+1} > a_i$. So, we should use the rockets from highest-acceleration to lowest-acceleration (any other order is suboptimal: it can be improved by swapping some pair of adjacent rockets).

(e) What is the running time for your algorithm in part (d)? (No justification needed.)

Solution: $\Theta(n \lg n)$.