

## CS 6150 - Fall 2025 - HW2

### Greedy algorithms, Local search, Graph traversal & shortest paths

Submission date: Friday, Oct 31, 2025 (11:59 PM)

This assignment has 6 questions, for a total of 110 points. You will still be graded out of 100, and any points you earn above 100 will count as bonus and can compensate for a low score on other homeworks. Unless otherwise specified, complete and reasoned arguments will be expected for all answers.

Question	Points	Score
Santa's tradeoffs	15	
$t$ farthest elements from each other	23	
Set Cover Revisited	24	
Kruskal's MST algorithm	10	
Let's plan a road trip	15	
All-Pairs Shortest Paths (APSP)	23	
Total:	110	

Note: Unless otherwise specified, complete and well-reasoned arguments for correctness and running time are expected for all answers. For the problems based on graphs, the different graph algorithms we covered in class (breadth-first/depth-first traversal, Dijkstra, Bellman-Ford, etc.) can be used as black-boxes if you apply them directly as we learned them. However, if you modify them to suit a given problem, spell out the modifications clearly (i.e. they are no longer black-boxes) with their effect on correctness and running time. Assume that, by default,  $n$  represents the number of vertices while  $m$  represents the number of edges in a graph.

Question 1: Santa's tradeoffs ..... [15]

Recall the matching problem we saw in class: there are  $n$  gifts, and  $n$  children, and each child has a non-negative valuation for each gift. Formally, the value of gift  $j$  to child  $i$  is given by  $V_{i,j}$ . We assume that all  $V_{i,j} \geq 0$ . Santa's goal is to give one gift to each child, so as to maximize the *total value* (of course, a gift cannot be given to more than one child). Suppose we now perform a more elaborate local search, this time picking every *triple* of edges in the current solution, and seeing if there is a reassignment of gifts between the end points of these edges that can improve the total value. Prove that a locally optimal solution produced this way has a value that is at least two-thirds ( $2/3$ ) of the optimum value. [This kind of trade-off is typical in local search – each iteration is now more expensive ( $O(n^3)$  instead of  $O(n^2)$ ), but the approximation ratio is better.]

To start this proof assume we begin with a locally optimum solution. This means that no swaps of three pairs will result in a higher happiness score. To demonstrate the relationship that comes from this assume we have 4 kids  $i, j, k, l, m$  the corresponding gifts for those 3 kids are  $a, b, c, d, e$ . Our goal here is to compare the locally optimum solution to the global. When comparing our solutions let  $j$  be the child that receives  $a$ 's optimal gift,  $k$  be the child that receives  $j$ 's optimal gift,  $d$  be the child that receives  $c$ 's optimal gift, and finally let  $e$  be the child that receives  $m$ 's optimal gift. Using this information we can get inequalities that will allow us to compare our two solutions.

$V_{i,a} + V_{j,b} + V_{k,c} \geq V_{i,b} + V_{j,c} + V_{k,a}$  With our current problem structure we don't have any information on the happiness of  $V_{k,a}$  so we will simply consider the worst case where the happiness received from the mapping of  $V_{j,a}$  is 0. Another thing to remember here is that  $i$ 's optimal gift is  $b$  and  $j$ 's optimal gift is  $c$

Using the same logic we can do the next level of the inequalities which would result in  $V_{j,b} + V_{k,c} + V_{l,d} \geq V_{j,c} + V_{k,d} + V_{l,b} \rightarrow V_{j,b} + V_{k,c} + V_{l,d} \geq V_{j,c} + V_{k,d}$

To make the pattern clear we will go one more level.  $V_{k,c} + V_{l,d} + V_{m,e} \geq V_{k,d} + V_{l,e}$ . Once again we consider the worst case for the last gift.

Putting all of these inequalities together show us the pattern that will allow us to finish our proof

$$V_{i,a} + V_{j,b} + V_{k,c} \geq V_{i,b} + V_{j,c} \quad (1)$$

$$V_{j,b} + V_{k,c} + V_{l,d} \geq V_{j,c} + V_{k,d} \quad (2)$$

$$V_{k,c} + V_{l,d} + V_{m,e} \geq V_{k,d} + V_{l,e} \quad (3)$$

From this we can see that each edge in the locally optimum solution appears 3 times and each edge in the globally optimum solution appears 2 times. This pattern will continue for all children gift pairs so long as they are setup as above. A simplified expression for what this means is shown below

$3 * L \geq 2 * G$ . Where  $L$  is all of the edges in the locally optimal solution and  $G$  is all of the solutions in the globally optimal solution. If we simplify this expression we get that  $L \geq \frac{2}{3}G$  Therefore we have proven that a locally optimal solution produced this way has a value that is at least two-thirds the optimum value.

Question 2:  $t$  farthest elements from each other ..... [23]

A common problem in returning search results is to display results that are *diverse*. A simplified formulation of the problem is as follows. We have  $n$  points in Euclidean space of  $d$ -dimensions, and suppose that by distance, we mean the standard Euclidean distance. The goal is to pick a subset of  $t$  (out of the  $n$ ) points, so as to maximize the sum of the pairwise distances between the chosen points. I.e., if the points are denoted  $P = \{p_1, p_2, \dots, p_n\}$ , then we wish to choose an  $S \subseteq P$ , such that  $|S| = t$ , and  $\sum_{p_i, p_j \in S} d(p_i, p_j)$  is maximized.

A common heuristic for this problem is local search. Start with some subset of the points, call them  $S = \{q_1, q_2, \dots, q_t\} \subseteq P$ . At each step, we check if replacing one of the  $q_i$  with a point in  $P \setminus S$  improves the objective value. If so, we perform the swap, and continue doing so as long as the objective improves. The procedure stops when no improvement (of this form) is possible. Suppose the algorithm ends with  $S = \{q_1, \dots, q_t\}$ . We wish to compare the objective value of this solution with the optimum one. Let  $\{x_1, x_2, \dots, x_t\}$  be the optimum subset.

(a) [5] Use local optimality to argue that:

$$d(x_1, q_2) + d(x_1, q_3) + \dots + d(x_1, q_t) \leq d(q_1, q_2) + d(q_1, q_3) + \dots + d(q_1, q_t).$$

(b) [8] Deduce that: [*Hint*: Use two inequalities of the form above.]

$$(t-1) \cdot d(x_1, x_2) \leq 2[d(q_1, q_2) + d(q_1, q_3) + \dots + d(q_1, q_t)].$$

(c) [10] Use this expression to argue that

$$\sum_{i,j} d(x_i, x_j) \leq 2 \sum_{i,j} d(q_i, q_j).$$

Note: this shows that the local optimum has an objective value at least  $1/2$  the global optimum.

Question 3: Set Cover Revisited ..... [24]

In class, we saw the set cover problem (phrased as picking the smallest set of people who cover a given set of skills). Formally, we have  $n$  people, and each person has a subset of  $m$  skills. Let the set of skills of the  $i$ th person be denoted by  $S_i$ , which is a subset of  $[m]$  (shorthand for  $\{1, 2, \dots, m\}$ ).

- (a) [8] Suppose that there is a set of seven (7) people whose skill sets optimally cover all of  $[m]$  (i.e., together, they possess all the skills). Now, suppose we run the greedy algorithm discussed in class until the set of people chosen covers at least 75% of the skills. How many people must we pick using the greedy algorithm to ensure this coverage?

**In class we proved the following theorem**

**Suppose there is an optimum solution that uses  $k$  people. Then the greedy algorithm does not use more than  $k \log_e m$  people**

- (b) [6] Consider the following “street surveillance” problem. We have a graph  $(V, E)$  with  $n$  nodes and  $m$  edges. We are allowed to place surveillance cameras at the nodes. Once placed, they can monitor all the edges incident to the node. The goal is to place as few cameras as possible, so as to monitor **all the edges** in the graph. Show how to cast the street surveillance problem as Set Cover.

**Instead of thinking of uncovered skills think of the edges as unsurveilled streets. Instead of getting the minimum number of people that cover all skills, we want to setup the minimum number of cameras such that each street is surveilled. Each node contains a subset of the unsurveilled streets (whichever edges are incident to that node.) With this problem setup we can use the exact same greedy logic as set coverage. Like set cover this solution won’t be optimal.**

- (c) [10] Let  $(V, E)$  be a graph as above, and suppose that the optimal solution for the street surveillance problem places  $k$  cameras (and is able to monitor all edges). Now consider the following “lazy” algorithm:

1. initialize  $S = \emptyset$
2. while there is an unmonitored edge  $\{i, j\}$ :  
     add both  $i, j$  to  $S$  and mark all their edges as monitored

Clearly (due to the while loop), the algorithm returns a set  $S$  that monitors all the edges. Prove that the set also satisfies  $|S| \leq 2k$  (recall that  $k$  is the number of nodes in the optimal solution).

MORAL. Even though the algorithm looks “dumber” than the greedy algorithm, it has a better approximation guarantee — 2 versus  $\log n$ .

*Hint.* Consider the edges  $\{i, j\}$  encountered when we run the algorithm. Could it be that the optimal set chooses *neither* of  $\{i, j\}$ ?

Question 4: Kruskal’s MST algorithm ..... [10]

Here is Kruskal’s greedy algorithm to find a minimum spanning tree of a weighted, undirected, connected graph  $G(V_G, E_G)$ , where  $V_G$  is the set of vertices with  $|V_G| = n$ , and  $E_G$  is the set of edges with  $|E_G| = m$ .

```

1: function KRUSKAL (  $G(V_G, E_G)$  )
2:   Sort  $E_G$  in monotonically increasing order of edge weight.
3:   Let graph  $T = (V_T, E_T)$  where  $V_T = V_G$  and  $E_T = \emptyset$ .
                                     ▷ In other words,  $T$  has all vertices of  $G$ , but no edges.
4:   for edge  $u-v$  in sorted  $E_G$  do
5:     if  $u-v$  does not complete a cycle in  $T$  then
6:       Add edge  $u-v$  to  $T$ .
7:     end if
8:   end for
9:   return  $T$ 
10: end function

```

Prove that, even though the for-loop at line 4 runs over all edges of  $G$ , KRUSKAL adds exactly  $n - 1$  edges to  $T$ . [Hint: *Think about the number of disconnected components in  $T$ .*] **To start this proof we**

can use the given hint. Before the for loop has executed all vertices have been added to our graph. When this happens there are exactly  $n$  disconnected components in our graph (One for each vertex).

When the first edge is added we connect two of the disconnected components. This leaves us with exactly  $n - 1$  disconnected components

For every edge after the first edge there are two cases. Case 1: We add an edge that comes from a vertex in a connected component and goes to a vertex outside of that connected component. In this case we are left with  $n - 2$  disconnected components. (You have one less disconnected component from the last step). Case 2: We add an edge with vertices that aren't in any connected components. In this case we also are left with  $n - 2$  disconnected components. Once we add an edge between 2 disconnected components they become 1 connected component.

That pattern continues as we iterate. From the pseudocode we can figure out what our stopping condition would be. The algorithm loops through all edges and adds the edge if it doesn't result in a cycle. Because we know that our graph began connected we can guarantee that the end result will also be connected. By definition of a tree we know that our final result will contain a single connected component and no disconnected components. Using what was proved in the paragraph above shows that we will add  $n - 1$  edges (each time we add an edge we are reducing the number of disconnected components by 1 and we reach termination when all disconnected components are connected).  $n, n - 1, n - 2, \dots, 1$

Because the problem asks us to show it is exactly  $n - 1$  edges we will show by contradiction why  $n - 2$  edges or  $n$  edges isn't a valid solution

For  $n - 2$  the contradiction is easy. Assume that we have added  $n - 2$  edges. This means by definition that we have exactly 2 disconnected components. When we finish. This fact can not be true because the for loop adds all edges that do not contain a cycle. This means that there must exist an edge between the 2 connected components that can be added that wouldn't result in a cycle.

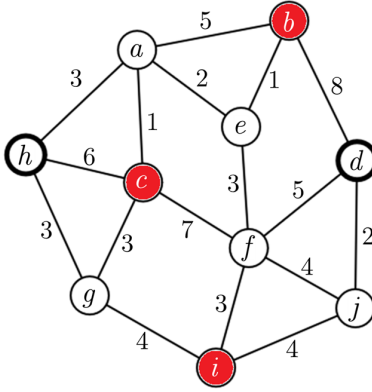
For  $n$  the contradiction is also easy. Assume that we have added  $n$  edges. This means by definition that we have added an edge after we only had a single connected component. This is a contradiction because any edge added after we have a single connected component must result in a cycle (we already have a path to every other edge) by the pigeonhole principle.

We have therefore proven that even though the for-loop at line 4 runs over all edges of  $G$  KRUSKAL adds exactly  $n - 1$  edges to  $T$

Question 5: Let's plan a road trip ..... [15]

Imagine you are planning a long road trip from your home  $h$  to your destination  $d$ . You have a roadmap in the form of a **simple, weighted, undirected graph**  $G(V, E)$ , and you need to plan your journey using it. Your trip is going to take several days, and there are a few vertices in the graph that have hotels where you can spend the night. Let  $H$  be the set of vertices with hotels, and  $H \subset V - \{h, d\}$ . The edge weights (represented as  $t_{pq}$  for edge  $p - q$ ) are positive integers indicating the **number of hours** required to travel along the edge. E.g. the figure below represents one such graph, where  $H = \{b, c, i\}$  and  $t_{ha} = 3$ ,  $t_{ab} = 5$ , and so on.

You can spend a maximum of 7 hours each day driving. After traveling **at most** 7 hours, you must reach a hotel to sleep. From that vertex, you can start driving for another 7 hours the next day. Given a graph  $G(V, E)$  and the subset  $H$ , give a **polynomial-time** (in  $m$  and  $n$ ) algorithm that finds out if it is possible to reach  $d$  (starting from  $h$ ) under the 7-hours-per-day constraint. There is no limit on how many days you take. E.g. in the graph shown above, it is possible to travel from  $h$  to  $d$  under the given constraints using many possible paths, such as  $h - g - i - j - d$  (2 days), or  $h - a - e - b - e - f - i - j - d$  (3 days). Your algorithm only needs to return yes or no (possibility of reaching  $d$ ), and outputting the number of days and the exact path is not necessary.



It is okay if you do not include a formal proof of correctness. Please give: i) a brief description of your problem-solving approach, ii) the pseudocode, and iii) analysis of the running time.

(i): When I was coming up with an algorithm I realized that we would start by check if  $d$  was reachable from  $h$  in  $\leq 7$  steps. If it wasn't our only other option would be to visit a hotel. From this hotel we would check if  $d$  was reachable in  $\leq 7$  steps. If it was not I would have to use another hotel.

Using this understanding we can create a reachability graph where  $n = \{H, h, d\}$  and  $m = (u,v)$  is an edge if there exists a path from  $u$  to  $v$ . Once this reachability graph is created we can run BFS to find the minimum number of stops in our reachability graph or DFS if we just want to check if a path exists

(ii): Note - This algorithm assumes that the number of hotels + source + dest is smaller than the number of nodes in the graph. If every node is either the source, dest, or a hotel this algorithm is no longer polynomial.

Note - This algorithm also assumes that there is a subroutine  $\text{Djikstras}(start, end)$  that returns true if there is a path from start to end that has a path cost of  $\leq 7$ . This subroutine is implemented exactly as Djikstras was in class but it doesn't add any edge weights to the connected component that have a cummulative cost higher than 7

Note - This algorithm assumes there is a subroutine  $\text{BFS}(nodes, edges, start, end)$  that returns true if there exists a path in the graph  $G = (nodes, edges)$  between start and end and returns false otherwise. This algorithm is implemented exactly how we saw BFS implemented in class

---

**Algorithm 1**  $\text{TripPlanning}(h, d, G, H)$

---

<pre> 1: <math>H \leftarrow \{H, h, d\}</math> 2: <math>N_r \leftarrow H</math> 3: <math>M_r \leftarrow \{\}</math> 4: <b>for</b> every node <math>N</math> in <math>N_r</math> <b>do</b> 5:   Call <math>\text{Djikstras}(N, X)</math> where <math>X</math> is every other node in <math>H</math> 6:   <b>if</b> path exists between <math>H</math> and <math>x</math> <b>then</b> 7:     Add edge <math>(N, X)</math> to <math>M_r</math> 8:   <b>end if</b> 9: <b>end for</b> 10: <b>return</b> <math>\text{BFS}(N_r, M_r, h, d)</math> </pre>	<pre> ▷ Add source and destination to H ▷ Nodes of reachability graph ▷ Edges of reachability graph </pre>
---	--

---

(iii):

Lines 1-3 are constant

The for loop on lines 4-9 take  $O(N^2 * \log n(m+n))$  where  $N$  is the number of hotels + source + destination. The  $N^2$  comes from the fact that we have to try all distinct pairs of the values in  $N$ . The other terms besides the  $N$  simply come from the algorithm of Dijkstra's discussed in class.

Line 10 takes  $O(N_r + M_r)$  using the usual BFS algorithm implemented in class

This analysis shows clearly that the overall runtime of the algorithm is polynomial like expected.

Question 6: All-Pairs Shortest Paths (APSP) ..... [23]

Given a directed graph  $G = (V, E)$  with non-negative edge lengths  $\{w_e\}$ , we define the *distance matrix*  $M$  as the  $n \times n$  matrix ( $n = |V|$  as usual) whose  $i, j$ 'th entry is the shortest path distance between vertices  $i$  and  $j$ . Given the graph (vertices, edges and lengths), the goal of the APSP problem is to find the matrix  $M$ .

In what follows, let  $G$  be an **unweighted, undirected** graph (all edge lengths are 1). Thus, in this case, shortest path from one vertex  $u$  to the rest of the vertices can be found via a simple BFS. (Thus the APSP problem can be solved in time  $O(n(m+n)) = O(n^3)$ .)

Let  $A$  denote the *adjacency matrix* of the graph, i.e., an  $n \times n$  matrix whose  $ij$ 'th entry is 1 if  $ij$  is an edge, and is 0 otherwise. Now, consider powers of this matrix  $A^k$  (defined by traditional matrix multiplication). Also, for convenience, define  $A^0 = I$  (identity matrix of size  $n \times n$ ).

- (a) [10] Prove that for any two vertices  $i, j$ , their distance in the graph  $d(i, j)$  is the smallest  $k \geq 0$  such that  $A^k(i, j) > 0$ .

**Claim:** For any two vertices  $i, j$  their distance in the graph  $d(i, j)$  is the smallest  $k \geq 0$  such that  $A^k(i, j) > 0$ . This claim is stating that the first  $k$ th matrix where  $i, j$  is zero represents the distance between the two vertices  $i, j$ .

To solve this problem it is important to understand what each  $A^k$  represents. When  $k = 1$   $A^k$  represents the connections that are one "hop" away from each node or directly connected by an edge. When we increment  $k$  to  $k + 1 = 2$   $A^k$  represents the connections that are 2 hops away. This is a known fact in graph theory for unweighted and undirected graphs. **Show an example of the idea of number of hops**

Using this fact we can easily see that the path with the smallest number of hops represents the shortest path. To find the smallest number of hops we can simply check which value of  $k$  yields a value  $> 0$ . The first time this shows up represents the smallest number of hops between  $(i, j)$  which also yields the smallest distance.

- (b) [8] The idea is to now use fast algorithms for computing matrix multiplications. Suppose there is an algorithm that can multiply two  $n \times n$  matrices in time  $O(n^{2.5})$ . Use this to prove that for any parameter  $k$ , in  $O(kn^{2.5})$  time, we can find  $d(i, j)$  for all pairs of vertices  $(i, j)$  such that  $d(i, j) \leq k$ . In other words, we can find all the *small* entries of the distance matrix. Let us see a different procedure that can handle the "big" entries.
- (c) [5] Let  $i, j$  be two vertices such that  $d(i, j) \geq k$ . Prove that if we sample  $(2 \ln n) \cdot \frac{n}{k}$  vertices of the graph uniformly at random, the probability of not sampling any vertex on the shortest path from  $i$  to  $j$  is  $\leq \frac{1}{n^2}$ . [Hint: You may find the inequality  $1 - x \leq e^{-x}$  helpful.]