Quiz 2

Instructions:

• Do not open this quiz booklet until directed to do so. Read all the instructions on this page.
• Write your name below and circle your recitation at the bottom of this page.
• Write your solutions in the space provided. If you need more space, write on the back of the sheet containing the problem. Pages will be separated for grading.
• You are allowed two one-sided, letter-sized sheets with your own notes. No calculators or programmable devices are permitted. No cell phones or other communication devices are permitted.

Advice:

• You have 120 minutes to earn a maximum of 120 points. Do not spend too much time on any single problem. Read them all first, and attack them in the order that allows you to make the most progress.
• When writing an algorithm, a clear description in English will suffice. Using pseudo-code is not required.
• Do not waste time rederiving facts that we have studied. Simply state and cite them.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Parts</th>
<th>Points</th>
<th>Grade</th>
<th>Grader</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td>14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>19</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>26</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>18</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>17</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>120</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Name: ____________________________

Circle your recitation:

R01 Brando Miranda 10AM
R02 Brando Miranda 11AM
R03 Parker Zhao 12PM
R04 Alex Jaffe 12PM
R05/R07 Danil Tyulmankov 1PM 2PM
R06 Peinan Chen 1PM
R08 Kevin Tian 2PM
R09 Allen Park 3PM
R10 Daniel Manesh 4PM
Problem 0. What is Your Name? [2 points] (2 parts)

(a) [1 point] Flip back to the cover page. Write your name and circle your recitation section.

(b) [1 point] Write your name on top of each page.
Problem 1. True or False [14 points] (7 parts)
For each of the following questions, circle either T (True) or F (False). There is no need to justify the answers. Each problem is worth 2 points.

(a) T F The rolling hash technique enables us to compute and compare hashes of two arbitrary strings, each of length \( l \), in \( O(1) \) time.
   Solution: False
   Remarks: The rolling hash technique applies only to situation in which we already have computed a hash of a string and want to compute a hash of the same string after removing or adding a letter to it.

(b) T F When using an adjacency matrix graph representation, relaxing all the edges in a graph \( G(V, E) \) can be done in \( O(E) \) time.
   Solution: False
   Remarks: In an adjacency matrix graph representation passing through all the edges of a graph takes always \( \Theta(V^2) \) time, even if \( E \ll V^2 \).

(c) T F Consider a modification to the table doubling procedure in which whenever the number of keys \( n \) is at least \( \frac{m}{2} \), where \( m \) is the current size of the table, we set its new size \( m' \) to be \( m' = 2m + \lfloor \sqrt{m} \rfloor \). If we never delete any elements from our table, then the resulting amortized overhead of this modified table doubling is still only \( O(1) \) per each hash table operation.
   Solution: True
   Remarks: The new size \( m' \) is still at least twice the size of the old table and at most three times that size. So, the \( O(1) \) per each hash table operation amortized analysis we covered still works.

(d) T F If \( G(V, E) \) is a directed graph in which all the edge weights are positive integers bounded by a fixed constant, then all shortest paths from a single source can be computed in \( O(V + E) \) time.
   Solution: True
   Remarks: Let \( C \) be the fixed constant bounding the largest weight. We can transform our graph into an unweighted one by substituting each original edge with a path of length equal to that edge’s weight. Clearly, the shortest path distances in such unweighted graph are the same as in the original graph and that unweighted graph has at most \( V' = V + C \cdot E \) vertices and \( E' = C \cdot E \) edges. So, we can compute shortest path distances in it (and thus in our original graph) using BFS in time
   \[
   O(V' + E') = O(V + C \cdot E + C \cdot E) = O(V + E).
   \]
(e) T F If we run Dijkstra’s algorithm on a graph in which there is a single negative-weight edge outgoing from the source $s$, then all the distances computed by this algorithm will be correct.

**Solution:** True

**Remarks:** As this negative-weight edge $e = (s, v)$ is outgoing from the sources and all other edges have non-negative length, the distance of this edge’s other endpoint $v$ from $s$ is equal to the (negative) weight $w_e$ of that edge $e$. So, the first round of relaxations of edges outgoing from $s$ will set $d[v] = w_e < 0$. Consequently, $v$ will be the first vertex (after $s$) pulled out of the priority queue $Q$ and its distance from $s$ will be computed correctly. As the edge $e$ will never take part in any further computations and all the remaining edges have non-negative weights, we know that the rest of distances will be computed correctly too.

(f) T F The union of a shortest path from $s$ to $u$ and a shortest path from $u$ to $t$ must also be a shortest path from $s$ to $t$.

**Solution:** False

**Remarks:** Consider the following graph:

![Graph](image)

(g) T F Assume that a data structure supports some operation in $O(T)$ amortized time. Consider now a situation in which we initialize the data structure and then make $k$ calls to that operation. The worst-case time needed to serve the first of these calls is $O(T)$.

**Solution:** True

**Remarks:** By definition of amortized complexity, for any $j \geq 1$, the (freshly initialized) data structure can support a sequence of $j$ calls in worst-case time $O(j \cdot T)$. However, as this guarantee has to hold for any $j$, it also holds for $j = 1$, which gives us the desired conclusion.
Problem 2. Ben Bitdiddle is an Ace Trader [19 points] (3 parts)

Ben Bitdiddle has landed a summer internship at a high-frequency trading firm Algorithmic Power (AP). Given his extensive 6.006 expertise, he has been put in charge of analyzing various communication networks the firm uses to send and receive trading information.

(a) [5 points] Ben’s first project requires him to implement the Bellman–Ford algorithm to find single-source shortest paths on graphs that may contain negative-weight cycles. Working too quickly without checking his work, Ben accidentally reversed the order of the inner and outer loops controlling the edge relaxation in the algorithm, so he relaxed each edge $V−1$ times before moving on to the next edge.

Draw a graph for which Ben’s algorithm would provide an incorrect set of shortest path lengths. Make sure that you specify the graph, the directions and weights of all the edges, as well as the order of edge relaxations. No further justification is necessary.

Solution:
Consider the following graph with the edge relaxation ordering of:

- $V−1$ relaxations of the edge $(s, t)$;
- $V−1$ relaxations of the edge $(u, t)$;
- $V−1$ relaxations of the edge $(s, u)$.

(After this execution the distances computed will be \(d[s] = 0\), \(d[u] = 1\), and \(d[t] = 3\), which is incorrect.)
(b) [6 points] Ben can’t find the error and correct the code he wrote in part (a), but he somehow discovers that by running this code multiple times without re-initialization, his results improve.

(i) What is the minimum number of times that Ben must run his defective code on a given graph $G = (V, E, w)$ to ensure that all shortest paths are correctly computed?

(ii) What is the worst-case asymptotic running time of Ben’s hacked version of his defective code to produce the correct result on a graph $G$?

Don’t forget to provide a concise justification.

**Solution:**

Ad (i): Note that performing $V - 1$ relaxations in a row is effectively just a single relaxation. So, Ben’s defective implementation of Bellman-Ford algorithm is essentially implementing one iteration of it (but with an $V - 1$ time overhead).

From the class we know that we need to execute $V - 1$ such iterations to guarantee that all the distances are correctly computed. (And we need one more iteration if we want to detect a negative-weight cycle, if it exists in $G$.)

Ad (ii): Given what we said above, in the worst case we need to execute Ben’s defective implementation of the Bellman-Ford algorithm $V - 1$ times. Also, each such execution takes $\Theta(V \cdot E)$ time to relax $V - 1$ times each one of $E$ edges in $G$.

So, the overall worst-case running time of the resulting algorithm is

$$(V - 1) \cdot \Theta(V \cdot E) = \Theta(V^2 \cdot E).$$
(c) [8 points] Ben’s next assignment is to compute the best route to support the traffic between the firm and the NASDAQ stock exchange. The underlying communication network is modeled as a directed graph $G = (V, E, w)$ with non-negative edge weights representing the average transit times along different network links. The AP headquarters corresponds to a vertex $s$ in that graph, and the NASDAQ stock exchange is a vertex $t$.

Ben needs now to compute the shortest $s \rightarrow t$ path in that communication network. Unfortunately, before he entered all the data from his paper diagram of that network, he spilled coffee on it and can no longer read the average transit time $w(e)$ of one of the network links $e$. This is especially regrettable because Ben remembered his supervisor telling him that the shortest $s \rightarrow t$ path is very likely to contain $e$. Ben would now like to reconstruct what the weight of $e$ would need to be in order to lie on such a shortest $s \rightarrow t$ path.

Help Ben by doing the following:

- Provide an efficient algorithm that will compute the largest edge weight $w^*$ that the edge $e$ can have while still being on a shortest $s \rightarrow t$ path in graph $G$.
- State the running time of your algorithm (no justification necessary).
- Argue, briefly, the correctness of your algorithm.

Solution:

Let $G'$ be the graph $G$ with the edge $e = (u, v)$ removed. Consider the following three shortest path distances in that graph:

- $\delta'(s, u)$, the distance from $s$ to $u$ in $G'$;
- $\delta'(v, t)$, the distance from $v$ to $t$ in $G'$;
- $\delta'(s, t)$, the distance from $s$ to $t$ in $G'$.

Observe that for $e$ to be on the shortest $s \rightarrow t$ path in the original graph $G$ we need to have that

$$\delta'(s, t) \geq \delta'(s, u) + w(e) + \delta'(v, t),$$

as this condition ensures exactly that the length of the shortest $s \rightarrow t$ path in $G$ that contains the edge $e$ is at most the length of the shortest $s \rightarrow t$ path in $G$ that does not contain $e$. Consequently, the maximum value of $w(e)$ that makes the above that condition satisfied is

$$w^* := \delta'(s, t) - \delta'(s, u) - \delta'(v, t).$$

(Note that $w^*$ might be negative, but in that case $e$ could not have been on a shortest $s \rightarrow t$ path in the original graph $G$, as all edge weights in $G$ were non-negative.)

In the light of the above, an algorithm that computes the three distances $\delta'(s, t)$, $\delta'(s, u)$, $\delta'(v, t)$ using three calls to Dijkstra’s algorithm, and returning $w^*$ as above, will correctly solve our problem. (Note that, in fact, we could compute first two of these distances in a single call to Dijkstra’s algorithm, but this does not affect the asymptotic running time of our algorithm.)
Also, the running time of this algorithm will be

\[ O(V + E) + O(V \log V + E \log V) + O(1) = O(V \log V + E \log V), \]

where the first term corresponds to constructing the graph \( G' \), the second accounts for our three calls to Dijkstra’s algorithm, and the final \( O(1) \) term is the time needed to compute \( w^* \) using the equation we derived.
Problem 3. Computing the “Longest” Shortest Path [26 points] (5 parts)

Prof. Madry enjoys sightseeing, so whenever he travels he always tries to choose a route that is as “round-about” as possible, provided it does not increase the total time needed to reach the destination.

To help Prof. Madry plan his trips, you are asked to design a fast algorithm for computing the “longest” shortest path. Specifically, consider a weighted directed graph \( G = (V, E, w) \), with all weights being positive, and an origin \( s \). The task is to compute, for each possible destination \( t \), the number of edges on the “longest” shortest \( s \leadsto t \) path, i.e., the shortest \( s \leadsto t \) path in \( G \) that maximizes the number of visited vertices among all the shortest \( s \leadsto t \) paths in \( G \).

(a) [5 points] Let us call an edge \( e = (u, v) \) tight if

\[
\delta(s, u) + w(u, v) = \delta(s, v),
\]

where \( \delta(s, v') \) is the shortest path distance from \( s \) to the vertex \( v' \).

Note that an edge is tight if and only if it is a part of some shortest path from \( s \) in \( G \).

Give an algorithm that computes the subgraph \( \hat{G} \) of \( G \) containing all the tight edges of \( G \) in \( O(E \log V + V \log V) \) time.

Solution:

To compute the subgraph \( \hat{G} \), let us just use Dijkstra’s algorithm to compute all the distances \( \delta(s, u) \), for all destinations \( u \). This takes \( O(E \log V + V \log V) \) time.

Then, we can identify all the tight edges in \( O(E) \) time by making a pass through all the edges of \( G \) and eliminating from it all the edges that do not satisfy the above tightness condition. The resulting subgraph will be exactly \( \hat{G} \).
(b) [6 points] Argue that, for any destination \( t \),

(i) every \( s \to t \) path in the subgraph \( \hat{G} \) is a shortest \( s \to t \) path in the graph \( G \);

(ii) every shortest \( s \to t \) path in \( G \) is an \( s \to t \) path in \( \hat{G} \).

*Hint:* Note that if an edge \( e = (u,v) \) is tight then \( w(u,v) = \delta(s,v) - \delta(s,u) \).

**Solution:**

To prove (i), note that if \( \hat{P} \) is an \( s \to t \) path in the subgraph \( \hat{G} \) then each of the edges \( e_1, \ldots, e_k \) on that path have to be tight. As a result, by the definition of tightness (see the hint), the weight \( w(\hat{P}) \) of that path in the graph \( G \) is exactly

\[
    w(\hat{P}) = \sum_{i=1}^{k} w(e_i) = \sum_{i=1}^{k} (\delta(s,v_i) - \delta(s,v_{i-1})) = \delta(s,v_k) - \delta(s,v_0) = \delta(s,t) - \delta(s,s) = \delta(s,t),
\]

where \( v_0 = s, v_1, \ldots, v_k = t \) are the consecutive vertices on the path \( \hat{P} \).

So, the weight of the path \( \hat{P} \) in \( G \) is exactly \( \delta(s,t) \). Thus, \( \hat{P} \) has to be a shortest \( s \to t \) path in \( G \).

To establish (ii), note that if \( P \) is a shortest \( s \to t \) path in \( G \) then each one of its edges has to be tight. Otherwise, this would violate the optimal substructure property of shortest paths. In other words, if we had some edge \( e = (u,v) \) on this path that is not tight, it would mean that the path consisting of that edge alone (which is a trivial subpath of the shortest \( s \to t \) path \( P \)) is not the shortest \( u \to v \) path in \( G \).

Consequently, all edges of \( P \) are present in the subgraph \( \hat{G} \) and thus \( P \) is indeed an \( s \to t \) path in \( \hat{G} \).
(c) [4 points] Prove that the subgraph $\hat{G}$ is acyclic.

**Solution:**

Assume for the sake of contradiction that there exists a cycle $C$ in the subgraph $\hat{G}$. Observe that as each edge of that cycle would need to be tight, by part (b) it would mean that this cycle is a shortest $v \sim v$ path in $G$, where $v$ is some vertex on that cycle. Consequently, it would mean that $C$ has a total weight of at most zero in $G$.

However, as all the edge weights in $G$ are positive, it is impossible to have a cycle in it of weight that is non-positive. So, such cycle could not have existed, which proves that $\hat{G}$ is indeed acyclic.
(d) [5 points] Give an algorithm that finds the length $l(t)$ of the longest $s \leadsto t$ path in $\hat{G}$ for all destinations $t$ in $O(V + E)$ time. (Note that the graph $\hat{G}$ is unweighted.)

Hint: Part (c) might be useful here.

Solution:

Let $\hat{G}'$ be the graph $\hat{G}$ in which each edge has a weight equal to $-1$. (Note we can create such a graph $\hat{G}'$ from $\hat{G}$ in $O(V + E)$ time.)

By part (c), we know that $\hat{G}'$ is acyclic (since $\hat{G}$ is such). So, the shortest path distances in $\hat{G}'$ are well-defined, i.e., there is no negative-length cycles.

Observe now that if $\delta'(s, t)$ is the shortest $s \leadsto t$ path distance in $\hat{G}'$ then $l(t) = -\delta'(s, t)$ is exactly the length of the longest $s \leadsto t$ path in $\hat{G}$.

Furthermore, as $\hat{G}'$ is acyclic, computing all the distances $\delta'(s, t)$, for all destination $t$, can be done in $O(V + E)$ time, by using the shortest path algorithm for DAGs that we presented in recitations.

Thus, indeed we can compute all the values $l(t)$ for all the destinations $t$ in $O(V + E)$ total time.
(e) [6 points] Consider now an algorithm that, first, uses part (a) to compute the graph $\hat{G}$ of all the tight edges in $G$ and, then, uses part (d) to find the length $l(t)$ of the longest $s \sim t$ path in that graph $\hat{G}$, for each destination $t$.

Prove that, for each destination $t$, the length $l(t)$ computed by the above algorithm is exactly the number of edges on the “longest” shortest $s \sim t$ path in $G$.

Solution:

Let us fix some destination $t$ and let $P_t$ be the “longest” shortest $s \sim t$ path in $G$, and let $k_t$ be the number of edges it contains.

Our goal is to prove that $k_t = l(t)$. We will do that by first arguing that $l(t) \geq k_t$ and then showing that $l(t) \leq k_t$.

To prove the first inequality note that, by part (b) (ii), the fact that $P_t$ is a shortest $s \sim t$ path in $G$ implies it is an $s \sim t$ path in the subgraph $\hat{G}$. So, by the correctness of the algorithm from part (d), we know that the length $l(t)$ of the longest $s \sim t$ path in $\hat{G}$ we compute has to be at least $k_t$. (Otherwise, $P_t$ would have a larger length in $\hat{G}$ than the longest $s \sim t$ path in that graph.)

Now to argue that we also have that $l(t) \leq k_t$, let us define $\hat{P}_t$ to be the longest $s \sim t$ path in $\hat{G}$. This means that the number of edges on path $\hat{P}_t$ is exactly $l(t)$.

By part (b) (i), we know that $\hat{P}_t$ has to be a shortest $s \sim t$ path in $G$ too. This, however, implies immediately our desired inequality $l(t) \leq k_t$, as $k_t$ is the maximum number of edges that any shortest $s \sim t$ path in $G$ can have.

Since our above reasoning is valid for any choice of the destination $t$, we can conclude that indeed the lengths $l(t)$ give us the number of edges on the “longest” shortest $s \sim t$ path in $G$, for all $t$. 
Problem 4. **Variants of Dijkstra’s Algorithm** [18 points]  (3 parts)

In addition to the standard Dijkstra’s algorithm, we discussed two other variants of that algorithm that sometimes might speed it up: bidirectional search and $A^*$ search. Still, these variants might perform differently on different graphs, and sometime work even worse than the standard Dijkstra’s algorithm. Your task will be to rank the effectiveness of each one of these three variants on the graphs provided below.

Specifically, for each graph below, give a ranking (by assigning ranks 1, 2, and 3) of these three variants of Dijkstra’s algorithm according to the number of attempted edge relaxations made when computing a single source, single target shortest path from $s$ to $t$ in that graph. (Rank 1 means the smallest number of such relaxations, and rank 3 means the largest number of them.) *No justification is necessary.*

*Whenever an algorithm needs to break ties to decide which edge to relax next, consider the worst case, i.e., tie breaking that will result in a maximum number of edge relaxations for that algorithm.*

Each of these graphs is undirected and each edge has unit weight.

Also, the potentials we use for $A^*$ search are given by the distance from $t$ in the horizontal direction only. That is, for each vertex $v$, its potential $\lambda_t(v)$ is equal to $|x(v) - x(t)|$, where $x(v)$ is the horizontal coordinate of vertex $v$. (This is almost the same potential used in the lecture and recitation notes, except that the vertical coordinate has been ignored.) To fix the scale, we let the distance between $s$ and $t$ along the horizontal axis be exactly 1. So, $\lambda_t(s) = 1$ and $\lambda_t(t) = 0$.

*Note: Partial credit will be given based on the correctness of the relative rankings.*

**(a) [6 points]** Dijkstra’s Alg. (2) Bidirectional Search (3) $A^*$ Search (1)

*Justification: Let us analyze the behavior of each one of these three variants on the provided graph.*

*Dijkstra’s Alg.: Note that the distance from $s$ to $t$ in this graph is exactly 4. So, the worst-case tie breaking in edge relaxation in that variant will lead to relaxing all the edges that are at a distance of at most 4 from $s$ in that graph. The figure below presents all these edges.*
There is exactly 8 of these edges, making Dijkstra’s algorithm be ranked second here.

**Bidirectional Search:** As the distance from s to t in this graph is exactly 4, in the worst-case, Bidirectional Search will terminate after relaxing all the edges that are at a distance at most 2 away from either s or t. The figure below presents all these edges.

There is exactly 16 of these edges, making Bidirectional Search be ranked third here.

**A* Search:** Recall that A* Search is equivalent to performing Dijkstra’s algorithm with respect to the modified weights $w^*$ given by the potentials $\lambda_t(v)$. Furthermore, by the “telescoping” property of that modified weights the length of any $u \sim v$ path with respect to them changes by an additive factor of exactly $\lambda_t(v) - \lambda_t(u)$. That is, it depends only on the difference of the potentials of its endpoints.

As the potential difference $\lambda_t(t) - \lambda_t(s)$ between t and s is exactly $-1$, this means that the length of the (unique) $s \sim t$ path with respect to modified weights is exactly 3 (instead of the original 4). Also, due to scaling, we know that no modified edge weight becomes negative.

Consequently, in the worst case, A* Search will relax all edges that are at a distance of at most 3 from s with respect to the modified weights. The figure below presents all these edges. (Note that the potential $\lambda_t(v')$ of the vertex $v'$ has to be strictly between 0 and 1, so the modified length of the $s \sim v'$ path is less than 3. On the other hand, the potential $\lambda_t(v'')$ of the vertex $v''$ is exactly $\lambda_t(s) = 1$, so the $s \sim v''$ path has a modified length of exactly 4.)
There is exactly 7 of these edges, making A* Search be ranked first here.
(b) [6 points]  
Dijkstra’s Alg. (3)  
Bidirectional Search (2)  
A* Search (1)

**Justification:** Let us analyze the behavior of each one of these three variants on the provided graph.

**Dijkstra’s Alg.:** Note that the distance from $s$ to $t$ in this graph is exactly 2. So, in the worst case, all the edges that are at a distance of at most 2 from $s$ in that graph will be relaxed. The figure below presents all these edges.

There is exactly 13 of these edges, making Dijkstra’s algorithm be ranked third here.

**Bidirectional Search:** As the distance from $s$ to $t$ in this graph is exactly 2, in the worst-case, Bidirectional Search will terminate after relaxing all the edges that are at a distance at most 1 away from either $s$ or $t$. The figure below presents all these edges.
There is exactly 10 of these edges, making Bidirectional Search be ranked second here. 

A* Search: By analogous reasoning to the one we employed in part (a), the modified length of the $s \sim t$ path in this graph is exactly 1 (instead of the original 2).

Consequently, in the worst case, A* Search will relax all edges that are at a distance of at most 1 from $s$ with respect to the modified weights. The figure below presents all these edges. (Note that the potentials $\lambda_t(v')$ and $\lambda_t(v'')$ of the vertices $v'$ and $v''$ have to be strictly larger than $\lambda_t(s) = 1$. So, the modified weights of the edges $(s, v')$ and $(s, v'')$ are strictly larger than 1.)

There is exactly 4 of these edges, making A* Search be ranked first here.

(c) [6 points] Dijkstra’s Alg. (3) Bidirectional Search (1) A* Search (2)

Justification: Let us analyze the behavior of each one of these three variants on the provided graph.

Dijkstra’s Alg.: Note that the distance from $s$ to $t$ in this graph is exactly 5. So, in the worst case, all the edges that are at a distance of at most 5 from $s$ in that graph will be relaxed. The figure below presents all these edges.
There is exactly 25 of these edges, making Dijkstra’s algorithm be ranked third here.

Bidirectional Search: As the distance from \( s \) to \( t \) in this graph is exactly 5, in the worst-case, Bidirectional Search will terminate after relaxing all the edges that are at a distance at most 3 away from either \( s \) or \( t \). The figure below presents all these edges.

There is exactly 17 of these edges, making Bidirectional Search be ranked first here.

\( A^* \) Search: By analogous reasoning to the one we employed in part (a), the modified length of the \( s \sim t \) path in this graph is exactly 4 (instead of the original 5).

Consequently, in the worst case, \( A^* \) Search will relax all edges that are at a distance of at most 4 from \( s \) with respect to the modified weights. The figure below presents all these edges. (Note that the modified weights of all the edges that do not belong to the
$s \sim t$ path are reduced. Also, the potentials $\lambda_t(v')$ and $\lambda_t(v'')$ of the vertices $v'$ and $v''$ are strictly between 0 and 1. This means the modified length of the $s \sim v'$ path is less than 4 but the modified length of the $s \sim v''$ is strictly larger than 4.

There is exactly 21 of these edges, making $A^*$ Search be ranked second here.
Problem 5. Ben Bitdiddle’s Adventures with Hashing [24 points] (4 parts)

Ben Bitdiddle learned about open addressing and is eager to give this new paradigm a try. At first, he wanted to use the linear probing technique, that is, to use a hash function $h_L$ in which the $i$-th probe is given by

$$h_L(k, i) = h(k) + i \pmod m,$$

where $h(k)$ is a “classic” hash function that satisfies the simple uniform hashing (SUHA) assumption.

However, he heard rumors that linear probing might perform poorly, so he decided to also consider applying the double hashing technique. Unfortunately, he did not realize that double hashing requires combining two different hash functions. He implemented instead a hash function $h_D$ in which the $i$-th probe is given by

$$h_D(k, i) = \left( h(k) + i \cdot (h(k) + 1) \right) \pmod m,$$

where the same “classic” hash function $h$ is used twice and the “+1” was introduced to deal with the case of $h(k) = 0$.

(a) [5 points] Recall that the function $h$ satisfies the simple uniform hashing assumption (SUHA). Does this imply that the hash function $h_D$ that Ben designed satisfies the uniform hashing assumption (UHA)? Justify your answer.

Solution: No, $h_D$ does not satisfy the uniform hashing assumption.

Recall that in order for $h_D$ to satisfy the uniform hashing assumption it must be the case that, for any key $k$, the probe sequence

$$h_D(k, 0), h_D(k, 1), \ldots, h_D(k, m - 1)$$

is a uniformly random permutation of the set $\{0, \ldots, m - 1\}$.

This means, in particular, that the hash function $h_D$ should be able to generate each one of the $m!$ such permutations with a non-zero probability. But, since there is only $m$ different values of $h(k)$, there is at most $m \ll m!$ different probe sequences that $h_D$ can generate for the key $k$.

Alternatively, one can notice that if for a given key $k$ it happens that $h(k) = m - 1$, which has to occur with non-zero probability by the fact that $h$ satisfies the simple uniform hashing assumption, then

$$h_D(k, i) = (h(k) + i \cdot (h(k) + 1)) \pmod m = (m - 1 + i \cdot m) \pmod m = m - 1,$$

for each $i$. So, the probe sequence generated by the hash function $h_D$ for the key $k$ is not even a permutation in that case.
(b) [5 points] Ben wants now to compare the performance of the hash functions \( h_L \) and \( h_D \). To this end, he constructs a hash table of size \( m \), with \( m \) even, that has the first half of its slots occupied. He inserts a new key \( k \) using the hash function \( h_L \) first.

Provide an asymptotic estimate (i.e., a \( \Theta \)-estimate) of the expected number of probes \( P_L \) needed to insert that new key \( k \) into that hash table if we use the hash function \( h_L \). Document your work.

**Solution:**

*Observe that, by the fact that \( h \) satisfies simple uniform hashing assumption, for any \( 0 \leq j \leq m - 1, \)

\[
\Pr[h(k) = j] = \frac{1}{m}.
\]

Also, if \( h(k) = j \), then the number of probes \( r_j \) needed to find an empty slot is exactly

\[
r_j = \max\left\{\frac{m}{2} - j - 1, 0\right\} + 1.
\]

Consequently, we have that the expected number of probes \( P_L \) is

\[
P_L = \sum_{j=0}^{m-1} \Pr[h(k) = j] \cdot r_j = \frac{1}{m} \sum_{j=0}^{m-1} \left(\max\left\{\frac{m}{2} - j - 1, 0\right\} + 1\right)
\]

\[
= \frac{1}{m} \left(m + \sum_{j=0}^{m-1} \left(\frac{m}{2} - j - 1\right)\right)
\]

\[
= \frac{1}{m} \left(m + \Theta(m^2)\right) = 1 + \Theta(m) = \Theta(m).
\]
(c) [6 points] Next, Ben wants to test the hash function \( h_D \) in the same scenario as \( h_L \) was tested in part (b). That is, he again considers a hash table of size \( m \), with \( m \) even, that has the first half of its slots occupied, and this time wants to insert a new key \( k \) into it using the hash function \( h_D \).

Provide an asymptotic estimate (i.e., a \( \Theta \)-estimate) of the expected number of probes \( P_D \) needed to insert that new key \( k \) into that hash table if we use the hash function \( h_D \).

Document your work.

**Hint:** Recall that \( \sum_{i=1}^{n} \frac{1}{i} = \Theta(\log n) \).

**Solution:**

Again, by the fact that \( h \) satisfies simple uniform hashing assumption, for any \( 0 \leq j \leq m - 1 \),

\[
\Pr [h(k) = j] = \frac{1}{m}.
\]

Also, if \( h(k) = j \) then the number of probes \( r'_j \) needed to find an empty slot is exactly

\[
r'_j = \max \left\{ \left\lceil \frac{m}{2} \cdot \frac{m - j - 1}{j + 1} \right\rceil, 0 \right\} + 1 = \max \left\{ \left\lceil \frac{m}{2} \cdot \frac{m - j - 1}{j + 1} \right\rceil, 0 \right\} + 1.
\]

Consequently, we have that the expected number of probes \( P_D \) is

\[
P_D = \sum_{j=0}^{m-1} \Pr [h(k) = j] \cdot r'_j = \frac{1}{m} \sum_{j=0}^{m-1} \left( \max \left\{ \left\lceil \frac{m}{2} \cdot \frac{m - j - 1}{j + 1} \right\rceil, 0 \right\} + 1 \right)
\]

\[
= \frac{1}{m} \left( m + \sum_{j=0}^{m-1} \left\lceil \frac{m}{2} \cdot \frac{m - j - 1}{j + 1} \right\rceil \right) = \frac{1}{m} \left( m + \sum_{j=0}^{m-1} \Theta \left( \frac{m}{j} \right) \right)
\]

\[
= \frac{1}{m} \left( m + \Theta (m \log m) \right) = 1 + \Theta(\log m) = \Theta(\log m).
\]
(d) [8 points] Ben starts to suspect that the hash function \( h_D \) is better than the hash function \( h_L \). However, he is not sure yet. To really confuse him, devise a scenario in which the hash function \( h_L \) performs much better than the hash function \( h_D \).

Specifically, devise an initial configuration of the occupied slots of the hash table such that the expected number \( P'_L \) of probes needed to insert a new key \( k \) into that hash table when using the hash function \( h_L \) is asymptotically smaller than the expected number \( P'_D \) of probes needed to insert \( k \) into the hash table using the hash function \( h_D \).

Remember to provide both a precise description of the initial configuration and asymptotic estimates of \( P'_L \) and \( P'_D \). Document your work.

**Solution:**

Consider an initial configuration in which every odd numbered slot is occupied. Then, \( h_L \) needs at most two probes to find an empty slot. That is, either it probes an unoccupied slot directly (which happens with probability \( \frac{1}{2} \)), or probes an occupied slot, but then the adjacent slot it will probe next has to be unoccupied. More precisely, we have that

\[
P'_L = \frac{1}{2} \cdot 1 + \frac{1}{2} \cdot 2 = \frac{3}{2} = \Theta(1).
\]

On the other hand, the hash function \( h_D \) also, with probability \( \frac{1}{2} \), probes an unoccupied slot directly. However, due to its flawed construction, if \( h_D \) probes an occupied slot then we must have that \( h(k) \) is odd and thus \( h(k) + 1 \) is even. As a result, \( h_D(k, i) \) is an odd number for all \( i \). Consequently, \( h_D \) will keep probing occupied location, ending up needing \( m \) probes total (to incorrectly conclude that the table is full). We thus have that

\[
P'_D = \frac{1}{2} \cdot 1 + \frac{1}{2} \cdot m = \frac{m + 1}{2} = \Theta(m).
\]
Problem 6. Escaping the Dungeon of an Evil Overlord [17 points] (3 parts)

You have been trapped by an Evil Overlord in his dungeon! You need to find an exit as soon as possible, before the overlord realizes that he left your cell unlocked.

The Evil Overlord’s dungeon can be modeled as an \( n \)-vertex tree, i.e., an undirected graph with no cycles, in which vertices correspond to rooms and edges to connections between these rooms. Your starting room is \( s \) and the exit room is \( t \).

You don’t know anything more about that dungeon graph except that, due to architectural feasibility constraints, there is at most \( B \) different vertices that are at a distance at most \( D \) from \( s \). We will refer to that property as being \( B \)-bounded. (Note that, a priori, the degree of each vertex can be arbitrary, as long as it does not make the graph be not \( B \)-bounded.)

Now, your job is to construct a strategy that, starting from the vertex \( s \), explores the dungeon graph and finds the room \( t \) containing the exit while trying to minimize the number of edges you traverse.

Note that whenever one travels from a room \( v \) to room \( v' \), one cannot simply “jump” there – one needs to follow the (unique) \( v \sim v' \) path in the dungeon and traverse all the edges of this path, possibly revisiting many of the already visited rooms.

(a) [5 points] Consider a strategy that corresponds to visiting the rooms in a DFS order. Show that such strategy might perform pretty poorly. That is, that it might require us to traverse \( \Omega(n) \) edges before finding the exit \( t \), even if the exit \( t \) is only at a distance \( D = 1 \) away from the initial room \( s \).

More precisely, specify first an \( n \)-vertex dungeon graph that is \( B \)-bounded, for any \( B \geq 3 \), with \( s \) and \( t \) marked and being within a \( D = 1 \) distance of each other. Then, present a DFS ordering of that graph’s vertices in which \( s \) is the first vertex visited but reaching \( t \) by following that DFS ordering requires \( \Omega(n) \) edge traversals. (Your example can be quite simple.)

Solution: Consider a dungeon graph that is a path of length \( n - 1 \), with \( t \) being the first vertex on this path and \( s \) being the second one. (See the figure below.)

Observe that \( s \) and \( t \) are within a distance of \( D = 1 \) of each other, as desired. Also, this graph is \( B \)-bounded for any \( B \geq 3 \), as there are exactly three vertices at a distance at most \( D = 1 \) from \( s \): \( t \), \( v_3 \), and \( s \) itself.

Finally, consider an ordering in which one goes to the “right” of \( s \), i.e., explores the vertices \( s, v_3, \ldots, v_n \), before backtracking and finally exploring \( t \). This is a valid DFS ordering and the number of edge traversals made before finding \( t \) is exactly

\[
2(n - 2) + 1 = 2n - 1 = \Omega(n),
\]

as desired.
(b) [6 points] Consider now a strategy that corresponds to visiting the rooms in a BFS order. Show that such a strategy might require us to traverse \( \Omega(B \cdot D) \) edges before finding the exit \( t \), for any setting of \( B \geq 2 \cdot D \).

To this end, for any given \( D \) and \( B \geq 2 \cdot D \), specify first an \( n \)-vertex \( B \)-bounded dungeon graph with \( s \) and \( t \) marked and being within a distance \( D \) of each other. (Note that your construction needs to be parametrized by \( D \) and \( B \).) Then, present a BFS ordering of that graph’s vertices in which \( s \) is the first vertex visited but reaching \( t \) by following that BFS ordering requires traversing \( \Omega(B \cdot D) \) edges.

**Hint:** Recall that you cannot “jump” between non-adjacent vertices during your exploration.

**Solution:** For a given \( D \) and \( B \geq 2 \cdot D \), let us define \( k = \lfloor \frac{B}{D} \rfloor \). Note that we have that \( k \geq 2 \). Consider now a graph that is a star graph of \( k \) paths \( P_1, \cdots, P_k \), with each of these paths being of length \( D - 1 \). Let \( s \) be the vertex at the center, i.e., \( s \) is a common endpoint of all these paths, and let \( t \) be the other endpoint of the path \( P_k \). (See the figure below.)

![Diagram of a star graph with paths and vertices labeled](image)

Clearly, \( s \) and \( t \) are at a distance of at most \( D \) away. Also, this graph is \( B \)-bounded since the total number of vertices at a distance at most \( D \) in this graph is

\[
k(D - 1) + 1 \leq B - k + 1 \leq B.
\]

Consider now an exploration ordering of vertices in which, in each phase \( i \), with \( 0 \leq i < D \), we explore vertices at distance \( i \) from \( s \), in order that always makes the vertices on path \( P_k \) be explored last.

Observe that this is a valid BFS ordering that visits \( s \) first and the vertex \( t \) last. Finally, note that exploring a given vertex at distance \( i \) from \( s \) on path \( P_j \), requires us to traverse \( 2i \) edges of the path \( P_j \). As a result, the total number of edge traversals made during that exploration is at least

\[
\sum_{i=0}^{D-1} k(2i) = 2k\Omega(D^2) = \Omega(B \cdot D),
\]

as desired.
(c) [6 points] Propose a strategy that, for any $D$ and $B \geq 2 \cdot D$, enables us to always find an exit in a $B$-bounded dungeon graph after at most $O(B)$ edge traversals, provided we know the distance $D$ between $s$ and $t$ in advance. Don’t forget to provide an analysis that establishes the worst-case $O(B)$ edge traversals bound.

**Solution:**

Consider a strategy in which we perform a DFS exploration starting from the vertex $s$ but we never explore vertices that are at a depth larger than $D$. That is, whenever we reached a vertex at depth $D$, we do not continue the exploration and instead backtrack. (Note that one can implement this modification by keeping a depth counter and updating it appropriately.)

Observe that this modification corresponds to performing a standard DFS walk in the subgraph of the dungeon graph consisting of all the vertices that are at a distance at most $D$ from $s$. As we know that $t$ is guaranteed to be within a distance $D$ from $s$, it is contained in that subgraph and thus is guaranteed to be found by this strategy.

Finally, note that DFS exploration (unlike BFS exploration) traverses each edge in the explored subgraph at most twice, even when it only has to move between adjacent vertices. Also, as the dungeon graph is a tree and has to be $B$-bounded, there is at most $B$ vertices, and thus at most $B$ edges, in that explored subgraph. So, the total number of edge traversals made by our exploration strategy is indeed at most $2B = O(B)$, as desired.