CSE 101 Homework 3

Kreshiv Chawla, Brian Masse, Taira Sakamoto, Emily Xie, Annabelle Coles

October 21, 2025

1. **Degrees of separation**

   In the seven degrees of Kevin Bacon party game, you try to connect the actor Kevin Bacon to a challenge actor by giving a sequence of movies and actors, so that Kevin bacon and the first actor in the list were in the first movie on the list, the first actor and second actor on the list were in the second movie, and so on, and the last actor on the list was the one you were challenged.

   We can make this an algorithmic problem as follows. We are given a list of movies, and for each movie, a list of actors that appeared in the movie. There are M movies total, and the total length of all the lists for all movies is L. We are given actors A and B and want to find a list of pairs $(movie_1, actor_1), ...(movie_k, actor_k = B)$ so that both A and $actor_1$ are in $movie_1$, and for $i = 2...k$, $actor_{i-1}$ and $actor_i$ are both in $movie_i$. We want to make the number of movies k as small as possible.

   Below, you will describe how to use a graph algorithm from class to solve the problem.

   (a) In graph G, the set of vertices will be the list of actors L. There will be L vertices total. Edges connect two vertices if both actors are casted in the same movie. Edges are undirected and non-weighted.

   (b) To create the graph, we will first create all the vertices from the list L. Then, iterate through each movie cast and add edges between all vertices, actors in the movie cast. To store the graph, we will use an adjacency matrix. To create the graph it will take $O(L^2)$ because for each vertex (actor) we need to check if is in the same movie as any other actors in list L.

   (c) Each vertex represents one actor, so for L number of unique actors, there will be L number of vertices.

   (d) Each edge represents a a vertex-vertex pair of actors that are in the same movie. Worst case, if all actors are casted in all movies, there will be at most L^2 edges, as there will be an edge between all vertices.

   (e) If there are k movies, and there exists a valid path $[A, B]$, the path length is at most $[A, B] \leq k$.

   (f) To find the shortest path $[A, B]$, we will use the Breadth First Search (BFS) algorithm. The inputs are graph G, actor name A and actor name B. Graph G has already been constructed. From the algorithm, we will append each actor name in the BFS traversal to a list, this will give the actor names in the shortest valid path.

   (g) The total time complexity of BFS is $O(|V|+|E|)$, in the worst case for our inputs, $|V| = O(L)$ and $|E| = O(L^2)$. So total time complexity is $O(L + L^2)) = O(L^2)$

2. **Smallest elements in a heap**

   Say you are given a binary min-heap of distinct integers, given as an array $A[1,..n]$ and an integer $1 \leq k \leq n$. Your goal is to output the k smallest elements stored in the heap. Give the most efficient algorithm you can for this problem. (7 points, clear description of algorithm. 7 points, correctness proof (hint: use a loop invariant.) 6 points, time analysis and efficiency . (Hint: use a second heap in your algorithm. My best time is $O(klogk)$.)

   Algorithm.x‘’

```
Initialize an empty min-heap $H$
push $(A[1], 1)$ into $H$                                  # Store (key, index) pair
for $t = 1$ to $k$ do
    $(x, i) \leftarrow$ pop-min$(H)$
    output $x$
    if $2i \leq n$ then
        push $(A[2i], 2i)$ into $H$                          # Left child
    end if
    if $2i + 1 \leq n$ then
        push $(A[2i+1], 2i+1)$ into $H$                      # Right child
    end if
end for
```

   Correctness. *Base case:* At first, only the root $(A[1], 1)$ is in H. Since it has no parent, the property holds.

   Inductive step: Assume the property holds before an iteration. When we remove the smallest element (x, i) from H, x is the smallest value not yet output (because all other elements in H are larger, and any nodes not in H have parents not yet output, so they must also be larger). We then add i's children (if any) into H. Their parent i has just been output, so the invariant still holds.

   Therefore, by induction, after k iterations, the k smallest elements are output in order.

   Time Analysis and Efficiency. Each iteration performs one pop and up to two push operations, each taking $O(\log k)$ time since the heap has size at most k.

   Therefore, the total time complexity is $O(k \log k)$, with extra space $O(k)$.

3. **Choosing which version of Dijkstra's algorithm to use**

   We saw that we could use either an array or a heap to implement the priority queue for Dijkstra's algorithm. For each of the types of graphs below, give the time complexity of Dijkstra's algorithm for both versions in terms of the number of vertices. Then say which one you would use for that type of graph. (5 points each, 2 for each version, and one for conclusion.)

   (a) A wheel graph, with a directed cycle of legth $n-1$ and a hub with edges to and from each of the vertices in the cycle.

   - $|V| = (n-1)+1 = n$
   - $|E| = (n-1)+2(n-1) = 3(n-1)$

   Thus,

   - Array $\in O(n^2+3(n-1)) \in O(n^2)$
   - Min-Heap $\in O(logn \cdot (4n-3)) \in O(n \cdot logn)$

   Thus, a min-heap is preferable.

   (b) A barbell graph, an undirected graph with two complete graphs on $n/2$ vertices each joined by a single edge.

   - $|V| = n$
   - $|E| = 2\cdot$ (Number of edges in connected graph) + $1 = 2 \cdot \frac{\frac{n}{2}!}{2!\cdot(\frac{n}{2}-2)!}$

   Array-implementation time complexity:

   $$\begin{aligned}&\in O(n^2+|E|)\\ &\in O(n^2+\frac{\frac{n}{2}!}{(\frac{n}{2}-2)!}) \in O(n^2+(\frac{n}{2})(\frac{n}{2}-1))\\ &\in O(n^2)\end{aligned}$$

   Min-Heap-implementation time complexity:

   $$O(log(n)(n+n^2)) \in O(n^2 \cdot logn)$$

   Thus, the array implementation is preferable

   (c) A grid graph, where each vertex (i,j) with $1 \le i,j \le \sqrt{n}$ is adjacent to $(i-1,j),(i+1,j),(i,j-1)$ and $(i,j+1)$, unless adding or subtracting 1 takes us out of the range.

   - $|V| = n \implies$ width & height of grid $= \sqrt{n}$
   - $|E| \le 4n$

   Array-implementation time complexity:

   $$\begin{aligned}&\in O(n^2+4n)\\ &\in O(n^2)\end{aligned}$$

Min-Heap-implementation time complexity:

$\in O(log(n) \cdot (n + 4n))$
$\in O(n \cdot log(n))$

Thus, the min-heap implementation is preferable

(d) A rook move graph, with vertices as above, but edges between any two vertices with either the same first coordinate or second coordinate.

- $|V| = n \implies$ width & height $(k) = \sqrt{n}$
- $|E|$:

Edges in column a given column j:

$$\begin{aligned} &= (k-1) + (k-2) + ...1 \\ &= \sum_{i=1}^{k}(k-i) = k^2 - \frac{k(k+1)}{2} \\ &= \frac{n - \sqrt{n}}{2} \end{aligned}$$

Edges in columns j = 1...$\sqrt{n}$ (Edges in rows i = 1...$\sqrt{n}$):

$$\begin{aligned} &= \frac{n^{\frac{3}{2}} - n}{2} \\ \implies & |E| = n^{\frac{3}{2}} - n \end{aligned}$$

Array-Implementation time complexity:

$\in O(n^2 + n^{\frac{3}{2}} - n)$
$\in O(n^2)$

Min-Heap-Implementation time complexity:

$\in O(logn \cdot (n + n^{\frac{3}{2}}))$
$\in O(n^{\frac{3}{2}} \cdot logn)$

Thus, the min-heap implementation is preferable

4. **Paths from a set of vertices**

Say you are given a set S of vertices in a directed graph with non-negative edge weights, and a single target vertex t. You want to find the shortest path from some $s \in S$ to t. Describe how to use or modify an algorithm from class to solve this problem efficiently. Be sure to describe exactly how you use or modify that algorithm, relate the correctness of that algorithm to the problem above, and give a time analysis of your entire algorithm in terms of the number of vertices and edges of the input graph.

```
def "shortestPath"(s, t):
    dist: dict[label: int] = []   // create a dictionary for initial distances
    prev: dict[label: label] = [] // create a dictionary for previous nodes

    for v in S: dist[v] = infinity
    dist[s] = 0

    F = createHeap(S)            // initialize the min-heap with all verticies,
        ordered by distance

    while (F not empty):
        v = F.removeMin()        // pull the vertex with the shortest path, O(logn)

        for u in neighborhood(v) and u not in F:

            // Use an array to store the location of labels in the min-heap
            // Such an array makes finding nodes O(1)
            if F.find(u) > F.find(v) + l(v, u):
                F.FindAndLower(u, F.find(v) + l(v, u))
                prev[u] = v

            if v == t:
                // Returning the previous allows the calling function to reconstruct
                // the path from s to t, by following prev[t] until reaching s.
                return prev
```

The above function works similar to the 'find minimum distances' implementation of Dijkstra's algorithm from class: Starting from the vertex s, the function searches for the shortest to every other node. Once it reaches t, it returns a dictionary that can be used to reconstruct the path:

- **Min-Heap:** Min-Heap is used to make the 'removeMin' function run in $O(logn)$ and the 'findAndLower' function run in $O(logn)$ time. Specifically, the 'find' function within 'findAndLower' is made to run in $O(1)$ by maintaining an array that maps a vertex to its index within the heap.
- **Return prev:** Prev is a map between vertexes and the previous vertex in the path from s.

Time Analysis:

- Each vertex is removed once it has been visited, meaning each vertex is visited at most once.

- 'FindAndLower' is run at most once per edge.

$$\begin{aligned} \implies \text{time complexity} &\in O(O(removeMin)|V| + O(FindAndLower)|E|) \\ &\in O(logn \cdot (|V| + |E|)) \end{aligned}$$

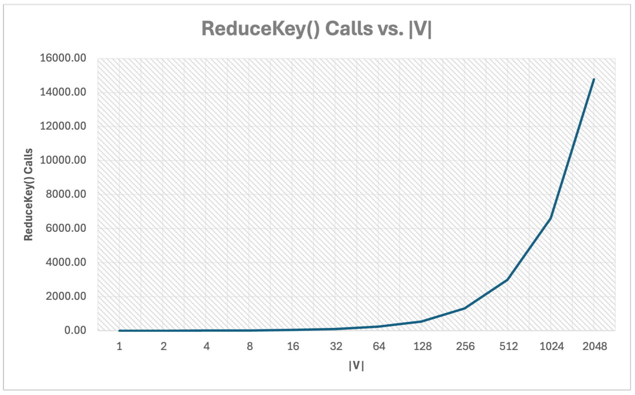
5. **Dijkstra experiment**

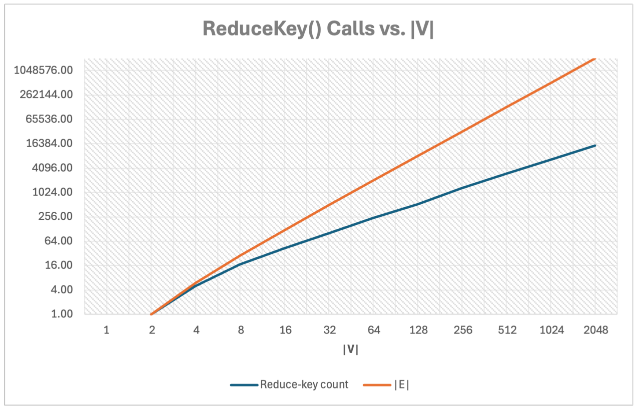
Implement Dijkstra's algorithm in any programming language (using data structures from libraries is OK), and modify it to count the number of times the decrease-key operation is used (just the number of times, not the total time spent on it.)

Test it by running it on complete simple directed graphs where each edge has a randomly chosen real weight in $[0, 1]$. Test it for many sizes of input, preferably say $|V| = 128$, 256, 512, 1024, ... and plot the number of decrease-key operations as a function of $|V|$. (Using a log-log scale is a good idea). Does this seem to match the theoretical worst-case bound? Can you think of a reason why it might not?

```
// adjacencyList stores a list of tuples (weight, label)
def djikstras(adjacencyList: list[list[tuple[int, int]]] ) -> int:

    vertex_count = len(adjacencyList)

    // Initialize the frontier with the first node (labeled 0) distance 0
    // And remaining verticies distance infinity
    // frontier is stored as (distance, label)
    frontier: list[tuple[int, int]] = [(0, 0)]
    explored: list[tuple[int, int]] = []

    count = 0

    for i in range(1, vertex_count):
        frontier.append( (math.inf, i) )

    heapq.heapify(frontier)

    while (len(frontier) != 0):
        current_vertex = heapq.heappop(frontier)
        heap_size = len(frontier)

        for edge in adjacencyList[current_vertex[1]]:
            label = edge[1]

            // find the label in the heap
            vertex_indicies = [i for i in range(0, heap_size) if frontier[i][1] ==
                label]
            if len(vertex_indicies) > 0:
                vertex_index = vertex_indicies[0]

                vertex_distance = frontier[vertex_index][0]
                current_distance = current_vertex[0]
                edge_length = edge[0]

                // update the distance of that vertex
                if vertex_distance > current_distance + edge_length:
                    frontier[vertex_index] = (current_distance + edge_length, label)
                    heapq.heapify(frontier)
                    count += 1

        explored.append(current_vertex)

    return count
```

Results





The actual number of calls to the ‘reduceKey()‘ function matches, asymptotically, the theoretical worst case bound. There are, however, consistently less actual calls to the ‘reduceKey()‘ function than the theoretical worst case number of calls.

They match asymptotically because the average case and the worst case number of calls of ‘reduceKey()‘ are the same using a min heap implementation. Specifically, both are proportional to the number of edges in the graph, $|E|$ ($|E| = \frac{|V|\cdot(|V|-1)}{2}$, for a complete graph).

In practice, the number of calls is less than the worse-case, maximum numbers of calls, since ‘reduceKey‘ is only called when a new shortest path is found, which is not true for every edge.