Homework Assignment 2

CS/CE 412/471 Algorithms: Design and Analysis, Spring 2025

4 problems, 100 points

1. Introduction to Epidemiology [15 points]

You are given a set of nodes representing villages and mountains. Some villages are initially infected, and others are healthy. Villages are connected by undirected paths, which define how the disease can potentially spread. There is a path between any two nodes, but the disease cannot cross a mountain. That is, a mountain is a special type of node through which the disease cannot enter or pass.

Each day, the disease spreads from an infected village to its directly connected healthy villages. We are interested to:

- Visualize the graph.
- Simulate the spread of the disease over time.
- Compute the minimum number of days required to infect all reachable healthy villages.
- Identify the healthy villages that cannot be infected (due to mountain barriers).

Input Format

The first line of the input contains two integers, n and m, the numbers of nodes and edges respectively, where $n-1 \leq m \leq \frac{n(n-1)}{2}$.

The next m lines each contain two integers, u and v, indicating an edge between vertex u and vertex v where $0leu, n < n$.

The next line contains two integers p and q, the number of mountains and initially infected villages respectively, where $(1 \leq p, q \leq n)$ and $2 \leq p+q \leq n$.

The next line contains p integers $a_1\ a_2\ a_3 \ldots a_p$ where each $0 \leq a_i < n$ indicates a mountain.

The next line contains q integers $b_1\ b_2\ b_3 \ldots b_q$ where each $0 \leq b_j < n$ indicates an initially infected village.

It is guaranteed that $a_i \neq b_j$ for all valid values of i and j,

Output Format

The first line of the output should contain a single integer denoting the minimum number of days to infect all reachable healthy villages. Output 0 if all reachable villages are already infected.

The second line of the output contains the nodes that represent the healthy villages that cannot be infected. Output 0 if all healthy villages can get infected.

Sample

Input	Output
7 6 1 2 2 3 3 4 4 5 5 6 8 7 1 1 4 1	2 5 6 7

The input describes a graph with $n = 7$ nodes and $m = 6$ edges. The next 6 lines describe the edges. There are $p = 1$ mountain nodes and $q = 1$ initially infected village nodes. The $p = 1$ nodes representing mountains are: 4. The $q = 1$ nodes representing initially infected villages are: 1. The output declares a minimum of 2 days for the maximal spread of the disease. On Day 1, the disease spreads from node 1 to node 2. On Day 2, it spreads from node 2 to node 3. It cannot spread further as it is blocked by the mountain at 4. The output declares that the villages $5, 6, 7$ do not get infected.

Tasks

(a) [5 points] Design a linear-time algorithm to simulate the spread. That is, ensure your solution runs in $\mathcal{O}(n+m)$ time, where n is the number of nodes and m is the number of edges. Describe below any considerations, e.g. appropriate data structures, to ensure the linear run time.

Solution:

GET-REACHABLE-NODES(n, G, *start_nodes*, *mountains*)

1: *reachable* $\leftarrow \emptyset$ {Set of reachable nodes}

```
2: Q ← start_nodes {Initialize queue with start nodes}
3: visited ← mountains {Mark mountains as visited}
4: while Q ≠ ∅ do
5:    node ← Q.DEQUEUE()
6:    if node ∉ visited then
7:       visited ← visited ∪ {node}
8:       reachable ← reachable ∪ {node}
9:       for each vertex v ∈ Adj[node] do
10:         if v ∉ visited and v ∉ mountains then
11:            Q.ENQUEUE(v)
12:         end if
13:      end for
14:   end if
15: end while
16: return reachable =0
```

DISEASE-SIMULATE(n, G, $mountains$, $infected$)

```
1: days ← 0 {Track days of spread}
2: visited ← mountains ∪ infected {Mark initial visited nodes}
3: newly_infected ← infected {Track current infections}
4: timeline ← [infected] {Store infection progression}
5: while newly_infected ≠ ∅ do
6:    current_infected ← ∅
7:    for each vertex u ∈ newly_infected do
8:       for each vertex v ∈ Adj[u] do
9:          if v ∉ visited and v ∉ mountains then
10:            current_infected ← current_infected ∪ {v}
11:            visited ← visited ∪ {v}
12:         end if
13:      end for
14:   end for
15:   if current_infected ≠ ∅ then
16:      days ← days + 1
17:      timeline ← timeline ∪ {current_infected}
18:   end if
19:   newly_infected ← current_infected
20: end while
21: reachable ← GET-REACHABLE-NODES(n, G, infected, mountains)
22: unreachable ← {1, 2, ..., n} − reachable − mountains
23: return (days, unreachable, timeline) =0
```

Time Complexity: The algorithm runs in $\Theta(n + m)$ time:

- GET-REACHABLE-NODES: $\Theta(n + m)$
  - Each vertex is processed at most once: $\Theta(n)$
  - Each edge is examined at most once: $\Theta(m)$
- DISEASE-SIMULATE: $\Theta(n + m)$

- Each vertex enters $newly\_infected$ at most once: $\Theta(n)$
- Each edge is examined exactly once: $\Theta(m)$
- Set operations (union, membership) take $\Theta(1)$

(b) [5 points] Submit files, `disease.in` and `disease.py`, where the former contains input and the latter reads the input and implements your algorithm on it to produce the output. You may design any reasonable input.

(c) [5 points] Submit an animation, `disease.mp4`, which simulates the spread of the disease on your graph from above. The last frame should include the output of your program.

2. Pakistan Trip Planning [25 points]

For this problem, we will study the road network of Pakistan to construct a graph modeling the roads between at least 12 cities, listed below, and implement an All-Pairs Shortest Path Algorithm (e.g., Floyd-Warshall) to find the shortest road distance between any pair of cities. You will write a program that visualizes the network, and interactively displays the shortest road distance between two input cities.

The cities include the capital cities of each province/region:

- Islamabad (Capital Territory)
- Lahore (Punjab)
- Karachi (Sindh)
- Peshawar (Khyber Pakhtunkhwa)
- Quetta (Balochistan)
- Muzaffarabad (Azad Jammu & Kashmir)
- Gilgit (Gilgit-Baltistan)
- At least five other major cities of your choice.

(a) [5 points] Submit a file, `roads.py`, that hard codes the graph, visualizes it, applies your chosen all-pairs shortest path algorithm on it and stores the distances, and handles interactive queries (case-insensitive and handling invalid input).

(b) [5 points] Include below any notes on your approach, cities used, and assumptions.

Solution:

Approach: We represented cities as nodes and direct roads as weighted edges in an undirected network. We precomputed the shortest pathways connecting all city pairs using the Floyd-Warshall method and then visualised the network with NetworkX, emphasising the chosen route.

Cities Used: Islamabad, Lahore, Karachi, Peshawar, Quetta, Muzaffarabad, Gilgit, Rawalpindi, Faisalabad, Multan, Sukkur, Hyderabad.

Assumptions: The distances are approximate, the network is undirected, and only a few key direct highways are modelled. The city placements are just used for visualisation, not for distance calculations.

Algorithm A: The Bucket Algorithm
Input: Graph $G = (V, E)$
Output: Bucket $B \subseteq V$ (a set of vertices)

1. Choose any vertex $x \in V$
2. Initialize Bucket $B \leftarrow x$
3. While there exists an edge $(u, v) \in E$ such that $u \in B$ and $v \notin B$:
   (a) Add vertex v to B

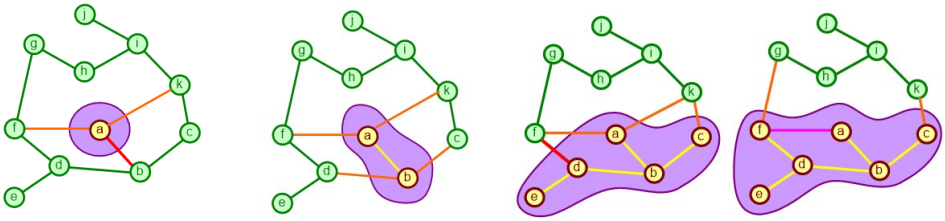


Figure 1: The bucket algorithm (top) and a visualization (bottom) of a run of the algorithm on an unweighted and undirected graph. The *bucket* is represented by the purple colored area. It expands from one step to the next.

(c) [5 points] Submit a short demo video (no longer than 5 minutes), `roads.mp4`, of a run of your program. The video should include the graph visualization, interaction with the user, and commentary by you including details of the algorithm, visualization, and interaction. Make sure that the visualization highlights any selected shortest route.

3. Greedy Bucket [35 points]

Figure 1 presents the Bucket Algorithm (Algorithm A) and illustrates a few steps of its dry run on an unweighted and undirected graph in which the bucket is represented by the purple colored area. The algorithm appears to useful to detect connected components in the graph. However, with slight modifications, it can do a lot more!

Figure 2 presents Algorithm B which modifies Algorithm A to output a tree containing connected vertices. Such a tree is called the *spanning tree*. Formally, "a spanning tree T of an undirected graph G is a subgraph that is a tree which includes all of the vertices of G. In general, a graph may have several spanning trees, but a graph that is not connected will not contain a spanning tree ... [rather a] spanning forest."[1]

(a) [5 points] Modify Algorithm B to operate on a *weighted* undirected graph and output its Minimum Spanning Tree i.e. a spanning tree that connects all the vertices with the least possible sum of edge weights. Call this Algorithm C.
Hint: be greedy in step 3b).

Solution:
Algorithm C: Greedy Bucket Algorithm for Minimum Spanning Tree

[1] `https://en.wikipedia.org/wiki/Spanning_tree`

Algorithm B: Bucket Algorithm with Spanning Tree
Input: Graph $G = (V, E)$ represented as an adjacency matrix
Output: Spanning tree T

1. Choose any vertex $x \in V$
2. Initialize Bucket $B \leftarrow x$, Tree $T \leftarrow \emptyset$
3. While there exists an edge $(u, v) \in E$ such that $u \in B$ and $v \notin B$:
   (a) Add vertex v to B
   (b) Add edge (u, v) to T
4. Return T

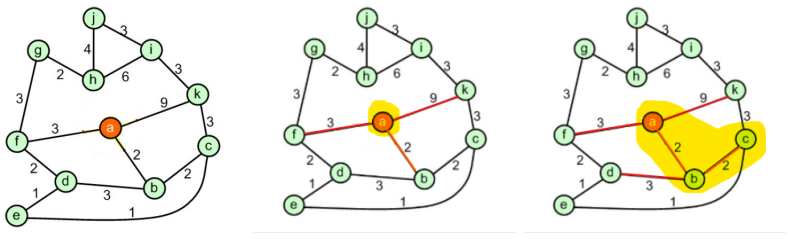
Figure 2: The bucket algorithm from Figure 1 is modified to compute a spanning tree.

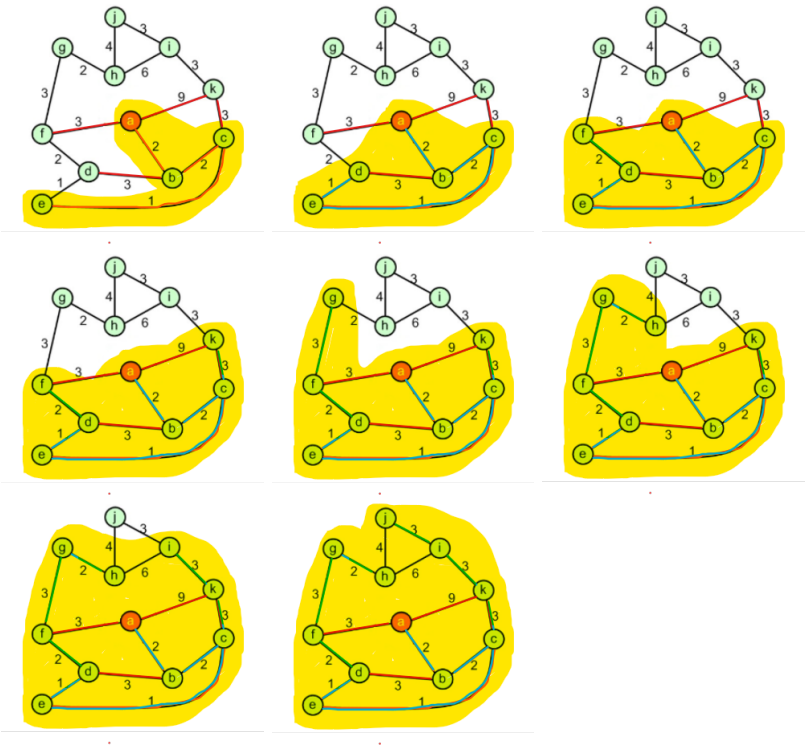
Input: Weighted undirected graph $G = (V, E)$ represented as an adjacency matrix
Output: Minimum spanning tree T

```
Choose any vertex x ∈ V
Initialize Bucket B ← {x}, Tree T ← ∅
while there exists an edge (u, v) ∈ E such that u ∈ B and v ∉ B do
    Select the edge (u, v) with the minimum weight among all such edges
    Add vertex v to B
    Add edge (u, v) to T
end while
return T =0
```

(b) [5 points] For Algorithm C that you developed above, visualize its run on the graph below starting from vertex a. Highlight the bucket and the minimum spanning tree edges at every step to clearly show how they grow (as shown in the sample dry run for Algorithm A).

Solution:





In the first three visualisations I was using orange colour to highlight MST edges, but it wasn't visible, so I switched to blue colour instead later.

(c) 5 points Derive the time complexity of Algorithm C.

Solution: At each step, we search for the minimum-weight edge between the current bucket and the rest. The search checks all edges from $u \in B$ to $v \notin B$.
Since there are $n-1$ steps in any spanning tree for n vertices. The selection process then runs for $n-1$ times. Finding this minimum weight requires scanning all edges so hence this process will take $O(m)$ time per step, where m is the number of edges. Since we are adding one vertex per step and there are $n-1$ steps so now the total time it takes is,

$$O(m \times n)$$

(d) 5 points Now let's design another algorithm, where now we *remove* certain edges frmo G such that the resulting graph is a Minimum Spanning Tree. Let's call this Algorithm D.
Note that while removing edges, we need to make sure the graph stays connected! To check connectivity use Algorithm A.

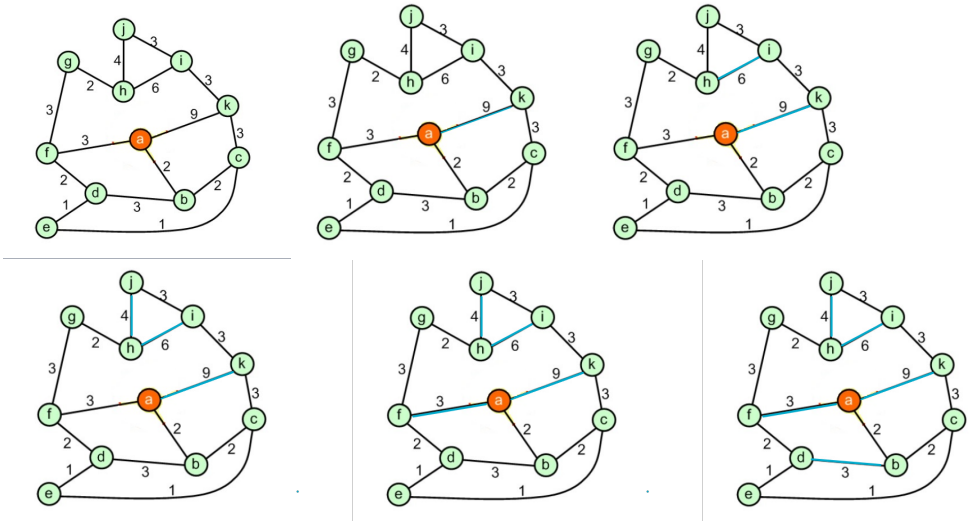
Solution:
Algorithm D: Edge-Removal Minimum Spanning Tree

Input: Weighted undirected connected graph $G = (V, E)$
Output: Minimum spanning tree T

1: Sort all edges E in **decreasing** order of weights
2: **for** each edge (u, v) in sorted order **do**
3: Tentatively remove edge (u, v) from G
4: Use Algorithm A (Bucket Algorithm) to check if G is still connected
5: **if** G is disconnected **then**
6: Add edge (u, v) back to G
7: **end if**
8: **end for**
9: **return** G (the remaining edges form a minimum spanning tree) =0

(e) 5 points For Algorithm D that you developed above, visualize its run on the graph below starting from vertex a. Highlight the edges you are deleting at every step to clearly show how the graph shrinks into an MST.

Solution:



The blue highlighted edges can be removed ensuring connectivity of graph.

For Algorithm D that you've designed in part c), dry run it on the graph below.

(f) 5 points Derive the time complexity of Algorithm D.

Solution: Sorting m edges on the basis of their weight takes $O(mlogm)$. After entering the loop, for each of the edges m we remove the edge tentatively and check its connectivity using `Algorithm A`. If we look at `Algorithm A` it is a simple `BFS/DFS` which takes a time complexity of $O(m+n)$, where n is the number of vertices and m is the number of edges.
So since for each edge we check it's connectivity using `Algorithm A` the total complexity then will be,

$$m \times O(n+m) = O(mn + m^2)$$

(g) [5 points] Modify Algorithm B to output its *Shortest Path Spanning Tree*. Unlike Minimum Spanning Tree (which minimizes the total weight of the tree), the Shortest Path Spanning Tree focuses on minimizing the distance from the root/starting vertex to each vertex.
Hint: be greedy in step 3b). Call this Algorithm E.

Solution:

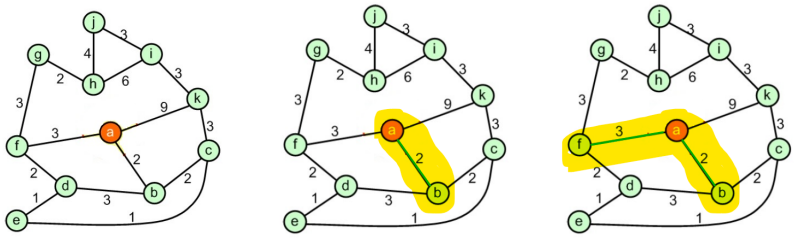
Algorithm E: Bucket Algorithm for Shortest Path Spanning Tree (Bellman-Ford style)

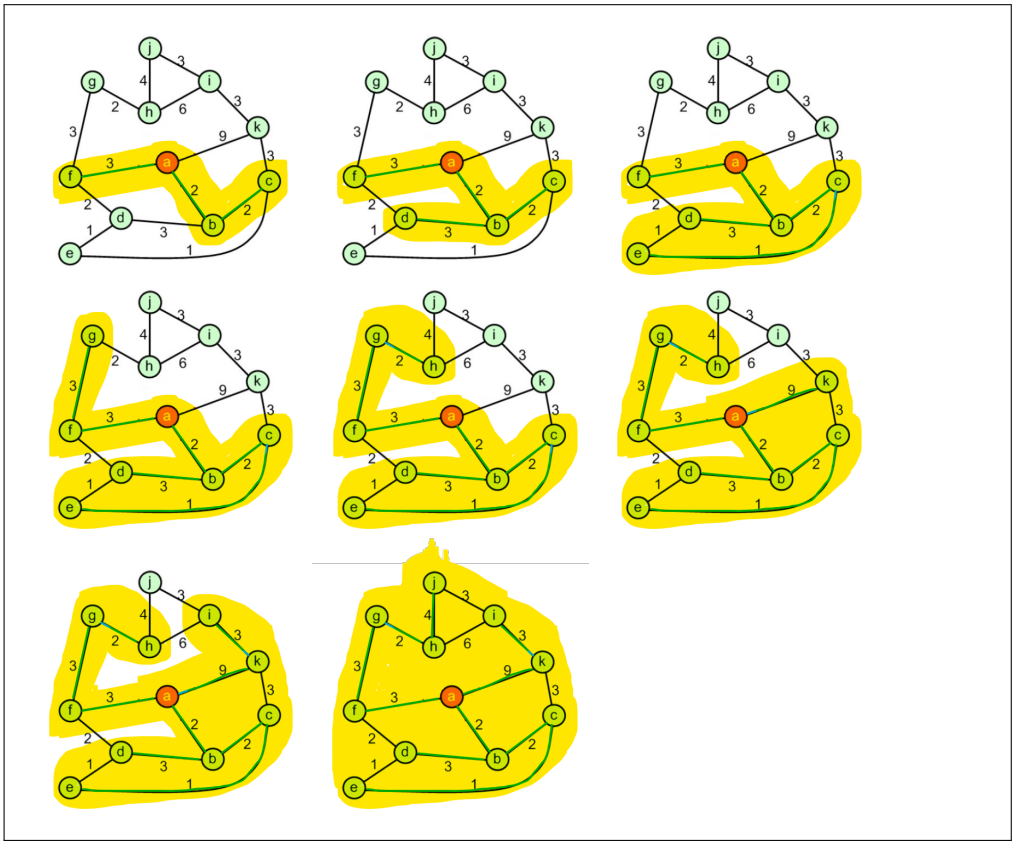
Input: Weighted graph $G = (V, E)$ represented as an adjacency matrix or list, starting vertex s
Output: Shortest Path Spanning Tree T

Initialize Tree $T \leftarrow \emptyset$
For all $v \in V$, set $dist[v] \leftarrow \infty$; set $dist[s] \leftarrow 0$
For all $v \in V$, set $parent[v] \leftarrow$ `None`
for $i \leftarrow 1$ to $|V| - 1$ **do**
  for each edge $(u, v) \in E$ **do**
    if $dist[u] + w(u, v) < dist[v]$ **then**
      $dist[v] \leftarrow dist[u] + w(u, v)$
      $parent[v] \leftarrow u$
    end if
    if $dist[v] + w(v, u) < dist[u]$ **then**
      $dist[u] \leftarrow dist[v] + w(v, u)$
      $parent[u] \leftarrow v$
    end if
  end for
end for
for each vertex $v \in V$ where $parent[v] \neq$ `None` **do**
  Add edge $(parent[v], v)$ to T
end for
return T =0

(h) For Algorithm E that you developed above, visualize its run on the graph below starting from vertex a. Highlight the bucket and the shortest path spanning tree edges at every step to clearly show how they grow (as shown in the sample dry run for Algorithm A).

Solution:





(i) Derive the time complexity of Algorithm E.

Solution:

Because we are using the Bellman-Ford method, we relax all edges for $V - 1$ iterations, where V represents the number of vertices.

In each iteration, we examine every edge in the graph. Relaxing one edge takes constant time $O(1)$. Since there are E edges, each full pass over all edges takes $O(E)$ time.

As we perform $V - 1$ such passes, the total time complexity becomes:

$$O(VE)$$

Thus, the overall time complexity of Algorithm E is:

$$\boxed{O(VE)}$$

where V is the number of vertices and E is the number of edges.

4. Getting into Agritech [25 points]

You have deployed a drone to inspect your vast farmlands. A drone's optimal flight distance between recharges is 1000m. That is flying xm induces stress damage (battery overcharge

or motor overheat) equal to $(1000 - x)^2$.

You are given a list of distances (in m) to charging stations at increasing distances from the starting point. The distance are given in sorted order. That is, the drone starts at distance 0 and encounters n charging stations at distances of $a_1 < a_2 < \cdots < a_n$ where each a_i is measured from the starting point. The drone can only stop and recharge at a charging station. It may stop at the charging stations in any order and need not stop at every station but must conclude at the final charging station (at distance a_n) which also houses the data center where the drone's recorded data is copied and analyzed.

As the resident computer scientist, it has come to you to plan the drone's journey.

(a) [5 points] Provide a recursive definition of the minimum stress damage. Make sure to indicate the base case. Clearly define any notation that you introduce.

Solution:
Let the array of charging station distances (in meters) be,

$$d = [0, a_1, a_2, \ldots, a_n]$$

where $a_1 < a_2 < \cdots < a_n$ and $d[0] = 0$ is the starting point. Now we'll define $S(i)$ to be the minimum stress damage incurred to reach station i from the starting point. The stress damage incurred while flying directly from station j to station i is,

$$\text{cost}(j, i) = (1000 - (d[i] - d[j]))^2$$

Then, the recursive formula for computing minimum stress is,

$$S(i) = \begin{cases} 0 & \text{if } i = 0 \\ \min\limits_{0 \le j < i} \left[S(j) + (1000 - (d[i] - d[j]))^2 \right] & \text{if } i > 0 \end{cases}$$

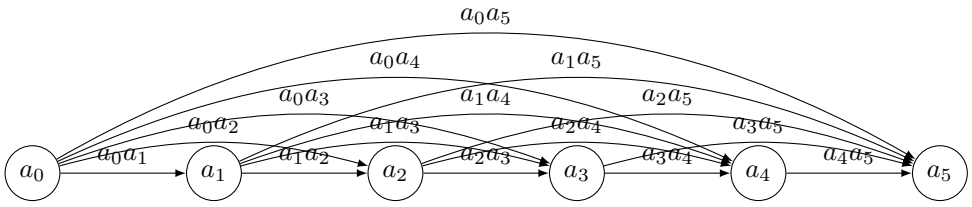
where,

- i is the current charging station index
- $d[i]$ is the distance of station i from start
- $(d[i] - d[j])$ is the distance between stations j and i

Final Goal: Calculate $S(n)$, which is the minimum total stress required to reach the last charging station at $d[n] = a_n$.

(b) [5 points] Include a below a visualization of the problem subgraph for $n = 5$.

Solution:



Explanation: Each node $dp(a_i)$ denotes the minimal stress damage required to reach station a_i. A directed edge exists between $dp(a_j)$ and $dp(a_i)$ if the drone can move straight from station a_j to station a_i (with $j < i$). The graph integrates overlapping subproblems instead of duplicating them, resulting in a compact subgraph rather than a tree structure.
To compute $dp(a_i)$, we consider all possible previous stations a_j $(0 \leq j < i)$ and select the one that minimizes the total accumulated stress, according to the formula,

$$dp(i) = \min_{0 \leq j < i} \left(dp(j) + (1000 - (a_i - a_j))^2 \right)$$

Thus, for each station a_i, we efficiently reuse solutions to previous subproblems without recalculating them, following the directed edges of the subgraph.

(c) [5 points] Discuss the time complexity of the solution.

Solution: To compute $S(i)$ we will consider all previous stations j where $0 \leq j < i$ which means that for each i we will examine i previous stations. For example,

For $i = 1$: we examine 1 previous station
For $i = 2$: we examine 2 previous stations
For $i = 3$: we examine 3 previous stations
$\vdots$
For $i = n$: we examine n previous stations

So the total work done is,

$$\sum_{i=1}^{n} O(i) = O(1 + 2 + \cdots + n) = O\left(\frac{n(n+1)}{2}\right) = O(n^2)$$

Therefore, the overall time complexity is $O(n^2)$, where n is the number of charging stations.

(d) [5 points] Provide a bottom-up implementation (pseudocode) to compute the minimum stress damage.

Solution:
Algorithm: Minimum-Stress-Damage

Input: Array $d[0..n]$ of charging station distances in increasing order (with $d[0] = 0$)
Output: Minimum total stress damage to reach station $d[n]$

```
1:  S[0] ← 0 {Base case: no stress at start}
2:  for i ← 1 to n do
3:      S[i] ← ∞
4:      for j ← 0 to i − 1 do
5:          d ← d[i] − d[j]
6:          stress ← (1000 − d)^2
7:          S[i] ← min(S[i], S[j] + stress)
8:      end for
9:  end for
10: return  S[n] =0
```

(e) [5 points] Show how the pseudocode can be modified to also compute the sequence of visited charging stations.

Solution:
Algorithm: Minimum-Stress-Damage-With-Path
Input: Array $d[0..n]$ of charging station distances in increasing order (with $d[0] = 0$)
Output: Minimum total stress damage and sequence of visited stations

```
0: procedure MIN-STRESS-DAMAGE-WITH-PATH(d[0 . . . n])
0:     Let S[0 . . . n] be a new array
0:     Let prev[0 . . . n] be a new array
0:     S[0] ← 0
0:     prev[0] ← −1
0:     for i ← 1 to n do
0:         S[i] ← ∞
0:         for j ← 0 to i − 1 do
0:             d ← d[i] − d[j]
0:             stress ← (1000 − d)^2
0:             total ← S[j] + stress
0:             if total < S[i] then
0:                 S[i] ← total
0:                 prev[i] ← j
0:             end if
0:         end for
0:     end for
0:     {Reconstruct path}
0:     path ← empty list
0:     current ← n
0:     while current ≠ −1 do
0:         Prepend current to path
0:         current ← prev[current]
0:     end while
0:     return (S[n], path)
0: end procedure=0
```