CS/ECE 374 P26

Jiawei Tang, Pengxu Zheng

TOTAL POINTS

75 / 100

QUESTION 1

1 26.A. 20 / 35

- **- 0 pts** Correct
- **- 20 pts** Implementing a completely correct O(nlogn+m) solution using Dijkstra's and adding weights on edges.
- **- 2 pts** Algorithm runtime is not stated/Incorrect.
- **- 5 pts** Algorithm is on the right track, but has minor mistakes
- **- 10 pts** In case of Solution 1 in the official solution, not explaining why shortest path in G' translates to minimizing dangerous vertices in graph.
- ✓ **- 15 pts Algorithm is on the right track, but has major mistakes**
- **- 26.25 pts** IDK
- **- 35 pts** Totally Incorrect / Blank

Based on the way this algorithm is described, we would need to run BFS on all the copied graphs. Worst case is n copied graphs and so BFS on n graphs with n vertices each is n^2 runtime.

QUESTION 2

2 26.B. 55 / 65

- **- 0 pts** Correct
- ✓ **- 10 pts Incorrect / Missing Runtime Analysis**
- **- 20 pts** Runtime worst than O(n^2logn+mn) but not exponential
- **- 10 pts** Algorithm on the right track, but has minor mistakes
- **- 20 pts** Algorithm on the right track, but has major mistakes
- **- 48.75 pts** IDK
- **- 65 pts** Totally Incorrect / Blank Solution

gradescope

HW9 Solution

CS/ECE 374: Algorithms & Models of Computation, Spring 2019 Version: **1.0**

Submitted by:

- **≪Pengxu Zheng≫**: **≪pzheng5≫**
- **≪Jiawei Tang≫**: **≪jiaweit2≫**

26

Solution:

26.A.
Model:

For part A, we modify the graph to conduct BFS to find the path that minimizes the total number of dangerous vertices in the path. We start with making copies ($G_{1...n}$) of the given graph G as many as the number of dangerous vertices in G. The number of copies should be bounded by n at the maximum.

Path Finding:

To find the path, we begin with vertex s and conduct BFS(s) on the first copy of the graph. The BFS algorithm should be modified slightly as follows: when searching for adjacent vertices of vertex u, if a dangerous vertex d is discovered, then add a corresponding edge from u (assuming we are currently on graph G_i) to d in Adj(u) on G_{i+1}, and let BFS continue on G_{i+1}.

Path Storage and Path sorting:

Once vertex t is reached in a path, the algorithm terminates and the path should be stored in a search tree. We can use an int variable to keep track of the number of "transitions" in the copies of G for each path, as each transition represent a necessary pass on a dangerous vertex. We then combine the number of transitions and the corresponding search tree into a search table, sort in terms of the number of transitions, and return the path with the least number of transitions as our final result.

Since we used BFS as our search algorithm, the run time should be bounded by $O(n + m)$ time. Given that $m \geq n$, We conclude that our algorithm runs in O(m) time since $O(n+m) \leq O(2m) = O(m)$.

26.B.

For part B, we use the same graph model. For path finding algorithms, we replace BFS with Dijkstra's algorithm to count for edge weights. Similarly, we still let the algorithm to "transit" through copies of G whenever a dangerous vertex is discovered during the search of Adj(u). At the end of the algorithm from part A, the search table made of int values of number of dangerous vertices and search trees of paths will be made. We sort the search table in terms of the number of dangerous vertices as usual (in increasing number of dangerous vertices passed). Since Dijkstra's algorithm already helped us to keep track of the total edge weights traversed, we apply the constraint W such that dist(s, t) computed by Dijkstra's algorithm do not exceed W. We check the sorted search table with W, and return the first path whose dist(s, t) does not exceed W that also has the smallest number of dangerous vertices passed. In terms of run time, Dijkstra's algorithm can run in $O(m + n * log(n))$ time using Fibonacci Heaps. Since the dominant term of the run time of the algorithm is bounded by Dijkstra's algorithm, and the dominant term of the run time of Dijkstra's algorithm is $O(n * log(n))$, we conclude that our algorithm runs in $O(n * log(n))$ time.

1 26.A. **20 / 35**

- **- 0 pts** Correct
- **- 20 pts** Implementing a completely correct O(nlogn+m) solution using Dijkstra's and adding weights on edges.
- **- 2 pts** Algorithm runtime is not stated/Incorrect.
- **- 5 pts** Algorithm is on the right track, but has minor mistakes
- **- 10 pts** In case of Solution 1 in the official solution, not explaining why shortest path in G' translates to minimizing dangerous vertices in graph.
- ✓ **- 15 pts** **Algorithm is on the right track, but has major mistakes**
- **- 26.25 pts** IDK
- **- 35 pts** Totally Incorrect / Blank

Based on the way this algorithm is described, we would need to run BFS on all the copied graphs. Worst case is n copied graphs and so BFS on n graphs with n vertices each is n^2 runtime.

gradescope

HW9 Solution

CS/ECE 374: Algorithms & Models of Computation, Spring 2019 Version: **1.0**

Submitted by:

- **≪Pengxu Zheng≫**: **≪pzheng5≫**
- **≪Jiawei Tang≫**: **≪jiaweit2≫**

26

Solution:

26.A.
Model:

For part A, we modify the graph to conduct BFS to find the path that minimizes the total number of dangerous vertices in the path. We start with making copies ($G_{1...n}$) of the given graph G as many as the number of dangerous vertices in G. The number of copies should be bounded by n at the maximum.

Path Finding:

To find the path, we begin with vertex s and conduct BFS(s) on the first copy of the graph. The BFS algorithm should be modified slightly as follows: when searching for adjacent vertices of vertex u, if a dangerous vertex d is discovered, then add a corresponding edge from u (assuming we are currently on graph G_i) to d in Adj(u) on G_{i+1}, and let BFS continue on G_{i+1}.

Path Storage and Path sorting:

Once vertex t is reached in a path, the algorithm terminates and the path should be stored in a search tree. We can use an int variable to keep track of the number of "transitions" in the copies of G for each path, as each transition represent a necessary pass on a dangerous vertex. We then combine the number of transitions and the corresponding search tree into a search table, sort in terms of the number of transitions, and return the path with the least number of transitions as our final result.

Since we used BFS as our search algorithm, the run time should be bounded by $O(n + m)$ time. Given that $m \geq n$, We conclude that our algorithm runs in O(m) time since $O(n+m) \leq O(2m) = O(m)$.

26.B.

For part B, we use the same graph model. For path finding algorithms, we replace BFS with Dijkstra's algorithm to count for edge weights. Similarly, we still let the algorithm to "transit" through copies of G whenever a dangerous vertex is discovered during the search of Adj(u). At the end of the algorithm from part A, the search table made of int values of number of dangerous vertices and search trees of paths will be made. We sort the search table in terms of the number of dangerous vertices as usual (in increasing number of dangerous vertices passed). Since Dijkstra's algorithm already helped us to keep track of the total edge weights traversed, we apply the constraint W such that dist(s, t) computed by Dijkstra's algorithm do not exceed W. We check the sorted search table with W, and return the first path whose dist(s, t) does not exceed W that also has the smallest number of dangerous vertices passed. In terms of run time, Dijkstra's algorithm can run in $O(m + n * log(n))$ time using Fibonacci Heaps. Since the dominant term of the run time of the algorithm is bounded by Dijkstra's algorithm, and the dominant term of the run time of Dijkstra's algorithm is $O(n * log(n))$, we conclude that our algorithm runs in $O(n * log(n))$ time.

2 26.B. **55 / 65**

- **- 0 pts** Correct
- ✓ **- 10 pts Incorrect / Missing Runtime Analysis**
- **- 20 pts** Runtime worst than O(n^2logn+mn) but not exponential
- **- 10 pts** Algorithm on the right track, but has minor mistakes
- **- 20 pts** Algorithm on the right track, but has major mistakes
- **- 48.75 pts** IDK
- **- 65 pts** Totally Incorrect / Blank Solution

gradescope