Project 2: Public Bicycle Management

Tan QiYe

2016-11-28

Contents

1 Introduction

1.1 Concepts Description

1.1.1 Graph

In computer science, and more specifically in graph theory, a graph is a structure amounting to a set of objects in which some pairs of the objects are in some sense "related." The objects correspond to mathematical abstractions called vertices (also called nodes or points) and each of the related pairs of vertices is called an edge (also called an arc or line). Typically, a graph is depicted in diagrammatic form as a set of dots for the vertices, joined by lines or curves for the edges. Graphs are one of the objects of study in discrete mathematics.

1.1.1.1 Definition of graph

In the most common sense of the term, a graph is an ordered pair $G = (V, E)$ comprising a set V of vertices, nodes or points together with a set E of edges, arcs or lines, which are 2-element subsets of V (i.e., an edge is associated with two vertices, and the association takes the form of the unordered pair of the vertices). To avoid ambiguity, this type of graph may be described precisely as undirected and simple.

The vertices which belong to an edge are called the ends or end vertices of the edge. A vertex may exist in a graph and not belong to an edge.

V and E are usually taken to be finite, and many of the well-known results are not true (or are rather different) for infinite graphs because many of the arguments fail in the infinite case. Moreover, V is often assumed to be non-empty, but E is allowed to be the empty set. The order of a graph is $|V|$, its number of vertices. The size of a graph is $|E|$, its number of edges. The degree or valency of a vertex is the number of edges that connect to it, where an edge that connects to the vertex at both ends (a loop) is counted twice. For an edge x, y, graph theorists usually use the somewhat shorter notation xy.

1.1.1.2 Types of graph

As stated above, in different contexts it may be useful to refine the term graph with different degrees of generality. Whenever it is necessary to draw a strict distinction, the following terms are used. Most commonly, in modern texts in graph theory, unless stated otherwise, graph means "ndirected simple finite graph".

- **Undirected graph:** An undirected graph is a graph in which edges have no orientation. The edge (x, y) is identical to the edge (x, y), i.e., they are not ordered pairs, but sets x, y (or 2-multisets) of vertices. The maximum number of edges in an undirected graph without a loop is $n(n-1)/2$.
- **Directed graph:** A directed graph or digraph is a graph in which edges have orientations. It is written as an ordered pair $G = (V, A)$ (sometimes $G = (V, E)$) with

- V a set whose elements are called vertices, nodes, or points;
- A a set of ordered pairs of vertices, called arrows, directed edges (sometimes simply edges with the corresponding set named E instead of A), directed arcs, or directed lines.

An arrow (x, y) is considered to be directed from x to y; y is called the head and x is called the tail of the arrow; y is said to be a direct successor of x and x is said to be a direct predecessor of y. If a path leads from x to y, then y is said to be a successor of x and reachable from x, and x is said to be a predecessor of y. The arrow (y, x) is called the inverted arrow of (x, y).

A directed graph G is called symmetric if, for every arrow in G, the corresponding inverted arrow also belongs to G. A symmetric loopless directed graph $G = (V, A)$ is equivalent to a simple undirected graph $G = (V, E)$, where the pairs of inverse arrows in A correspond one-to-one with the edges in E; thus the number of edges in G is $|E| = |A|/2$, that is half the number of arrows in G.

- **Weighted graph:** A weighted graph is a graph in which a number (the weight) is assigned to each edge. Such weights might represent for example costs, lengths or capacities, depending on the problem at hand. Some authors call such a graph a network. Weighted correlation networks can be defined by soft-thresholding the pairwise correlations among variables (e.g. gene measurements). Such graphs arise in many contexts, for example in shortest path problems such as the traveling salesman problem.

1.1.1.3 Properties of graphs

If two edges of a graph share a common vertex, they are called adjacent. Two arrows of a directed graph are called consecutive if the head of the first one is the tail of the second one. Similarly, two vertices are called adjacent if they share a common edge (consecutive if the first one is the tail and the second one is the head of an arrow), in which case the common edge is said to join the two vertices. An edge and a vertex on that edge are called incident.

The graph with only one vertex and no edges is called the trivial graph. A graph with only vertices and no edges is known as an edgeless graph. The graph with no vertices and no edges is sometimes called the null graph or empty graph, but the terminology is not consistent and not all mathematicians allow this object.

Normally, the vertices of a graph, by their nature as elements of a set, are distinguishable. This kind of graph may be called vertex-labeled. However, for many questions it is better to treat vertices as indistinguishable. (Of course, the vertices may be still distinguishable by the properties of the graph itself, e.g., by the numbers of incident edges.) The same remarks apply to edges, so graphs with labeled edges are called edge-labeled. Graphs with labels attached to edges or vertices are more generally designated as labeled. Consequently, graphs in which vertices are indistinguishable and edges are indistinguishable are called unlabeled. (Note that in the literature, the term labeled may apply

to other kinds of labeling, besides that which serves only to distinguish different vertices or edges.)

1.1.2 Shortest Path Problem

The shortest path problem can be defined for Graph whether undirected graph, Directed graph, or Mixed graph. It is defined here for undirected graphs; for directed graphs the definition of path requires that consecutive vertices be connected by an appropriate directed edge.

Two vertices are adjacent when they are both incident to a common edge. A Path in an undirected graph is a sequence]of vertices $P = (v_1, v_2, \ldots, v_n) \in V \times V \times \cdots \times V$ such that v_i is adjacent to v_{i+1} for $1 \leq i < n$. Such a path P is called a path of length $n - 1$ from v_1 to v_n. (The v_i are variables; their numbering here relates to their position in the sequence and needs not to relate to any canonical labeling of the vertices.)

Let $e_{i,j}$ be the edge incident to both v_i and v_j. Given a Function weight function $f : E \to \mathbb{R}$, and an undirected (simple) graph G, the shortest path from v to v' is the path $P = (v_1, v_2, \ldots, v_n)$ (where $v_1 = v$ and $v_n = v'$) that over all possible n minimizes the sum $\sum_{i=1}^{n-1} f(e_{i,i+1})$. When each edge in the graph has unit weight or $f : E \to \{1\}$, this is equivalent to finding the path with fewest edges.

The problem is also sometimes called the single-pair shortest path problem, to distinguish it from the following variations:

- The single-source shortest path problem, in which we have to find shortest paths from a source vertex v to all other vertices in the graph.
- The single-destination shortest path problem, in which we have to find shortest paths from all vertices in the directed graph to a single destination vertex v. This can be reduced to the single-source shortest path problem by reversing the arcs in the directed graph.
- The all-pairs shortest path problem, in which we have to find shortest paths between every pair of vertices v and v' in the graph.

These generalizations have significantly more efficient algorithms than the simplistic approach of running a single-pair shortest path algorithm on all relevant pairs of vertices.

1.2 Project Description

In order to make Hangzhou City a mordern tourism city, the government provides a public bike service through which tourists can rent a bike at any station and return it to any other stations in the city. However, there is a problem on how to manage the bikes in different stations by collecting or sending bikes to adjust the condition of stations through Public Bike Management Center (PBMC).

Now we assume that if a station is exactly half-full, it is said to be in perfect condition, while if a station is full or empty, it is called problem station. Therefore, if a problem station is reported, PBMC will send some bikes to there or collect some bikes from there. Meanwhile, along the way, every station will be adjusted to the perfect condition. For the purpose of saving resources, we need to find the shortest path and if there are more than one shortest path, we should choose the one which requires the least number of bikes sent from PBMC.

According to this assumption, our tasks are:
(1) Find the shortest path;
(2) Calculate the number of bikes that need to be sent;
(3) Calculate the number of bikes that need to be returned;
(4) Compare various paths and choose the one that requires the least resources.

1.3 Input & Output

1.3.1 Input Specification

For each input, the first line contains 4 numbers:

- C_{max} : The maximum capacity of each station which is always an even number and no more than 100;
- N : The total number of stations which is less than or equal to 500;
- S_p : The index of the problem station;
- M : The number of roads.

By default, we number the stations from 1 to N, represented by S_i (i=1,$\cdots$,N), and PBMC is represented by the vertex 0. After that, the second line contains N non-negative numbers C_i (i=1,$\cdots$,N), which represent the current number of bikes at S_i repectively. Then the rest of input have M lines, each of which contains 3 numbers: S_i , S_j, and T_{ij}. It means that it will cost the time T_{ij} to move between stations S_i and S_j. Besides, All the numbers in a line are separated by a space.

1.3.2 Output Specification

For each input, output just occupy one line. In this line, the first number is that how many bikes that PBMC must send. Then after printing a space, the output should print the path in the format: 0->S1->$\cdots$->Sp. Finally, after printing another space, the output will print the number of bikes which should be taken back from S_p after the adjustment complete.

If there are more than one shortest path, we should choose the one that requires the least number of bikes that should be taken back to PBMC. The judge's data guarantee that such a path is unique.

1.3.3 Sample Input

```
8 3 2 5
6 0 7
0 1 2
0 2 1
1 2 3
1 3 2
2 3 3
```

1.3.4 Sample Output

```
4 0->2 0
```

2 Algorithm Specification

We firstly use The Shortest Path Faster Algorithm (SPFA) to find the shortest path and restore all the shortest paths. Secondly, we use Depth-First Search (DFS) to finally choose the correct path and calculate numbers.

2.1 Data structure

In order to represent graph structure, we use adjacency list to store the information of abstract graph structure.An adjacency list representation for a graph associates each vertex in the graph with the collection of its neighboring vertices or edges. Each list describes the set of neighbors of a vertex in the graph. This is one of several commonly used representations of graphs for use in computer programs.

First of all, we create an array to store every vertex by default index. What's more, each element of this array is the head of a list, which contains all the neighbour vertices of the start vertex(the head of list). In addition to the index of neighbour vertex, time variable is needed in the struct, so as to represent the weighted graph to deal with the problem.

Code 1: Adjacency List

```
1 typedef struct __Road{
2     int time;
3     int next_point;
4     struct __Road* next_road;
5 }Road;
6
7 // g_roads[i] for all roads that started with station i
8 Road* g_roads[MAX + 1];
```

Each element of g_roads[ ] is the head of a list whose nodes are "struct ___Road". Road is used to store the roads(or edges) information for a certain start (the head of the list).

By using Adjacency List, the space complexity to store information is $O(|V| + |E|)$, which is much smaller than Adjacency matrix and Incidence matrix. Meanwhile, both the time complexity of adding vertex or edge are $O(1)$.

2.2 The Shortest Path Faster Algorithm (SPFA)

2.2.1 General

SPFA is an improvement of the Bellman-Ford algorithm which can compute single-source shortest paths in a weighted directed graph. The algorithm is believed to work well on random sparse graphs and is particularly suitable for graphs that contain negative-weight edges. However, the worst-case complexity of SPFA is the same as that of Bellman–Ford, so for graphs with nonnegative edge weights Dijkstra's algorithm is preferred.

The basic idea of SPFA is that each vertex is used as a candidate to relax its adjacent vertices. Once a vertex is relaxed then put that vertex into the queue and the vertexs in the queue are the potential candidate vertexs. The procedure goes on until there is no vertexs can be relaxed.

2.2.2 Proof of Correctness

The correctness of the algorithm can be shown by induction. The precise statement shown by induction is:

Lemma. After i repetitions of **for** loop:

- If Distance(u) is not infinity, it is equal to the length of some path from s to u;
- If there is a path from s to u with at most i edges, then Distance(u) is at most the length of the shortest path from s to u with at most i edges.

Proof. For the base case of induction, consider $i = 0$ and the moment before **for** loop is executed for the first time. Then, for the source vertex, $source.distance = 0$, which is correct. For other vertices u, $u.distance = infinity$, which is also correct because there is no path from *source* to u with 0 edges.

For the inductive case, we first prove the first part. Consider a moment when a vertex's distance is updated by $v.distance := u.distance + uv.weight$. By inductive assumption, $u.distance$ is the length of some path from *source* to u. Then $u.distance + uv.weight$ is the length of the path from *source*to v that follows the path from *source* to u and then goes to v.

For the second part, consider the shortest path from *source* to u with at most i edges. Let v be the last vertex before u on this path. Then, the part of the path from *source* to v is the shortest path from *source* to v with at most $i - 1$ edges. By inductive assumption, $v.distance$ after $i - 1$ iterations is at most the length of this path. Therefore, $uv.weight + v.distance$ is at most the length of the path from s to u. In the i^{th} iteration, $u.distance$ gets compared with

$uv.weight + v.distance$, and is set equal to it if $uv.weight + v.distance$ was smaller. Therefore, after i iteration, $u.distance$ is at most the length of the shortest path from *source* to u that uses at most i edges.

If there are no negative-weight cycles, then every shortest path visits each vertex at most once, so at step 3 no further improvements can be made. Conversely, suppose no improvement can be made. Then for any cycle with vertices $v[0],\cdots,v[k-1]$:

$$v[i].distance \leq v[i-1 \pmod{k}].distance + v[i-1 \pmod{k}]v[i].weight$$

Summing around the cycle, the $v[i].distance$ and $v[i-1 \pmod{k}].distance$ terms cancel, leaving :

$$0 \leq \text{sum from 1 to k of } v[i-1 \pmod{k}]v[i].weight$$

I.e., every cycle has nonnegative weight

2.2.3 Implementation in Pseudo Code

Code 2: SPFA in Pseudo Code

```
1  function exp(S: source vertex)
2      Queue Q;
3      Enqueue(S, Q);
4      while !IsEmpty(Q) do begin
5          u := Dequeue(Q);
6          for each v ∈ u.AdjacentList do begin
7              tmp := disk[v];
8              relax(u, v);
9              if tmp > disk[v] and v is not in Q do begin
10                 Enqueue(v, Q);
11             end;
12         end;
13     end;
```

2.2.4 Implementation in C

Code 3: SPFA in C

```
1  void spfa(int src){
2      int now_station = src;
3      int now_temp_queue[MAX * 4];
4      int front = 0;
5      int rear = 0;
6      now_temp_queue[front] = 0;
7      front++;
8
9      while(front != rear){
10         now_station = now_temp_queue[rear];
11         rear++;
12         Road* i = g_roads[now_station];
```

```
13          for(; i != NULL; i = i->next_road){
14              int temp = g_paths[i->next_point].time;
15              int now = g_paths[now_station].time + i->time;
16              if(temp > now){
17                  g_paths[i->next_point].last = now_station;
18                  g_paths[i->next_point].time = now;
19                  now_temp_queue[front] = i->next_point;
20                  front++;
21              }
22          }
23      }
24  }
```

2.3 Using Depth-First Search (DFS) to select path

2.3.1 General

Depth-first search (DFS) is an algorithm for traversing or searching graph data structures. One starts at the source and explores as far as possible along each edge before backtracking. Therefore, we can collect the resources that every shortest path needs and compare them to find the correct one.

The time and space analysis of DFS differs according to its application area. In theoretical computer science, DFS is typically used to traverse an entire graph, and takes time $\Theta(|V| + |E|)$, linear in the size of the graph. In these applications it also uses space $O(|V|)$ in the worst case to store the stack of vertices on the current search path as well as the set of already-visited vertices. Thus, in this setting, the time and space bounds are the same as for breadth-first search and the choice of which of these two algorithms to use depends less on their complexity and more on the different properties of the vertex orderings the two algorithms produce.

For applications of DFS in relation to specific domains, such as searching for solutions in artificial intelligence or web-crawling, the graph to be traversed is often either too large to visit in its entirety or infinite (DFS may suffer from non-termination). In such cases, search is only performed to a limited depth; due to limited resources, such as memory or disk space, one typically does not use data structures to keep track of the set of all previously visited vertices. When search is performed to a limited depth, the time is still linear in terms of the number of expanded vertices and edges (although this number is not the same as the size of the entire graph because some vertices may be searched more than once and others not at all) but the space complexity of this variant of DFS is only proportional to the depth limit, and as a result, is much smaller than the space needed for searching to the same depth using breadth-first search. For such applications, DFS also lends itself much better to heuristic methods for choosing a likely-looking branch. When an appropriate depth limit is not known a priori, iterative deepening depth-first search applies DFS repeatedly with a sequence of increasing limits. In the artificial intelligence mode of analysis, with a branching factor greater than one, iterative deepening increases the

running time by only a constant factor over the case in which the correct depth limit is known due to the geometric growth of the number of nodes per level.

DFS may also be used to collect a sample of graph nodes. However, incomplete DFS, similarly to incomplete BFS, is biased towards nodes of high degree.

2.3.2 Implementation in Pseudo Code

Code 4: DFS in Pseudo Code

```
1  var
2      route[MAX + 1] := record paths;
3  function DFS(G := Graph, V := source vertex)
4      route[index] := V;
5      for E ∈ V.AdjacentList do begin
6          if (G.Mark[E] == UNVISITED)
7              DFS(G, E);
8      Calculate(G, route);
9  }
```

2.3.3 Implementation in C

Code 5: SelectPath in C

```
1  void selectPath(int point){
2      static int top = 0;
3      static int route[MAX + 1];
4
5      if(point != 0){
6          route[top] = point;
7          top++;
8          int tempTop = top;
9          for(int i = 0; i <= g_paths[point].top; i++){
10             selectPath(g_paths[point].last[i]);
11             top = tempTop;
12         }
13     }
14
15     if(point == 0){
16         int send = 0;
17         int ret = 0;
18         int temp = 0;
19         for(int i = top − 1; i >= 0; i−−){
20             temp += g_cMax / 2 − g_bikes[route[i;
21             if(temp > send){
22                 send = temp;
23             }
24         }
25         ret = send − temp;
26         int case = (send == finalSend && ret < finalRet);
```

```
27          if(send < finalSend || case){
28              finalRoute[0] = 0;
29              for(int i = 1; i <= top; i++){
30                  finalRoute[i] = route[top - i];
31              }
32              finalSend = send;
33              finalRet = ret;
34          }
35      }
36  }
```

2.4 Dijkstra's algorithm (another choice)

2.4.1 General

Dijkstra's algorithm is another algorithm for finding the shortest paths between nodes in a graph, which we can use to replace SPFA.

For a given source node in the graph, the algorithm finds the shortest path between that node and every other. It can also be used for finding the shortest paths from a single node to a single destination node by stopping the algorithm once the shortest path to the destination node has been determined. For example, if the nodes of the graph represent cities and edge path costs represent driving distances between pairs of cities connected by a direct road, Dijkstra's algorithm can be used to find the shortest route between one city and all other cities.

Dijkstra's original algorithm does not use a min-priority queue and runs in time $O(|V|^2)$ (where $|V|$ is the number of nodes). The idea of this algorithm is also given in Leyzorek et al. 1957. The implementation based on a min-priority queue implemented by a Fibonacci heap can run in $O(|E| + |V| \log |V|)$ (where $|E|$ is the number of edges). This is asymptotically the fastest known single-source shortest-path algorithm for arbitrary directed graphs with unbounded non-negative weights.

2.4.2 Proof of Correctness

Proof is by induction on the number of visited nodes.

Invariant hypothesis: For each visited node u, dist[u] is the shortest distance from *source* to u; and for each unvisited v, dist[v] is the shortest distance via visited nodes only from *source* to v (if such a path exists, otherwise infinity; note we do not assume dist[v] is the actual shortest distance for un-visited nodes).

The base case is when there is just one visited node, namely the initial node source, and the hypothesis is trivial.

Assume the hypothesis for $n - 1$ visited nodes. Now we choose an edge uv where v has the least dist[v] of any unvisited node and the edge uv is such that dist[v] = dist[u] + length[u,v]. dist[v] must be the shortest distance from

source to v because if there were a shorter path, and if w was the first unvisited node on that path then by hypothesis dist[w] < dist[v] creating a contradiction. Similarly if there was a shorter path to v without using unvisited nodes then dist[v] would have been less than dist[u] + length[u,v].

After processing v it will still be true that for each unvisited node w, dist[w] is the shortest distance from source to w using visited nodes only, since if there were a shorter path which doesn't visit v we would have found it previously, and if there is a shorter path using v we update it when processing v .

2.4.3 Implementation in Pseudo Code

Code 6: Dijkstra in Pseudo Code

```
1  function Dijkstra(Graph, source):
2
3      create vertex set Q
4
5      for each vertex v in Graph:
6          dist[v] ← INFINITY
7          prev[v] ← UNDEFINED
8          add v to Q
9
10     dist[source] ← 0
11
12     while Q is not empty:
13         u ← vertex in Q with min dist[u]
14         remove u from Q
15
16         for each neighbor v of u:           .
17             alt ← dist[u] + length(u, v)
18             if alt < dist[v]:
19                 dist[v] ← alt
20                 prev[v] ← u
21
22     return dist[], prev[]
```

2.4.4 Implementation in C

Code 7: Dijkstra in C

```
1  void dijkstra(int src){
2      int now_station = src;
3      int visited[MAX + 1];
4      for(int i = 0; i <= g_N; i++){
5          visited[i] = 0;
6      }
7
8      //initialize the heap
9      priority_queue heap;
```

```
10      heap = (priority_queue)malloc(sizeof(struct __Heap));
11      heap->size = 0;
12      heap->capacity = MAX + 1;
13      heap->stations[0] = 0;
14      heap->positions[0] = 0;
15
16      insert(now_station, heap);
17      visited[src] = 1;
18
19      while(heap->size != 0){
20          now_station = delete_min(heap);
21
22          //now the now_station is poped from the heap
23          if(visited[now_station]==1){
24              continue;
25          }
26          visited[now_station] = 1;
27
28          //traverse all the road of the now_station
29          for(Road* i = g_roads[now_station];
30              i != NULL;
31              i = i->next_road){
32
33              int top = g_paths[i->next_point].top;
34
35              int now_time = g_paths[now_station].time + i->time;
36
37              if(g_paths[i->next_point].time > now_time){
38                  top = 0;
39                  g_paths[i->next_point].top = 0;
40                  g_paths[i->next_point].last[top] = now_station;
41                  g_paths[i->next_point].time = now_time;
42                  if(visited[i->next_point] == 0
43                      ||
44                      visited[i->next_point]==1){
45
46                      insert(i->next_point, heap);
47                      visited[i->next_point] = 1;
48                  }
49              }
50
51              else if(g_paths[i->next_point].time == now_time){
52                  g_paths[i->next_point].top++;
53                  top++;
54                  g_paths[i->next_point].last[top] = now_station;
55              }
56          }
57      }
58  }
```

3 Test Results

3.1 Test Cases

In this project, we used the following cases to test the correctness and the performance of our program:

3.1.1 Problem Sample

this test case is provided by the original problem, which is used for checking the correctness of the program in general cases.

```
10 3 3 5
6 7 0
0 1 1
0 2 1
0 3 3
1 3 1
2 3 1
```

3.1.2 Smallest Boundary Sample

this test case is generated manually, which is used for check the vulnerability and stablity of the problem when facing special situation general cases.

```
1 1 1 1
1
0 1 1
```

3.1.3 Largest Line Sample

this test case is generated by python script, which is used for check the vulnerability and stablity of the problem when facing largest output, all of the point is in the answer sequence.
This test case is too larger, we only provide part of the input in the report, you can find the full case in the same folder.

```
100 500 498 500
76 64 52 17 27 ...
0 1 5
1 2 42
2 3 73
3 4 11
...
```

3.1.4 Largest Random-Value Sample

this test case is generated by python script, the Random value of point provide a simulation of reality, which will happen in a large probability.
This test case is too larger, we only provide part of the input in the report, you can find the full case in the same folder.

```
100 500 6 250000
60 67 84 26 73 ...
189 460 85
55 68 47
132 483 28
183 467 75
...
```

3.1.5 Largest Same-Value Sample

this test case is generated by python script, the random-value case provide a large input of our shortest path algorithm, but it cannot provide enough shortest path to test the speed of our depth-first-search algorithm, so we provide this case to ensure the depth-first-search part of our problem can work correctly and fast.
This test case is too larger, we only provide part of the input in the report, you can find the full case in the same folder.

```
100 500 60 500000
65 46 26 33 33 ...
115 371 1
99 291 1
222 469 1
198 317 1
...
```

3.1.6 Sepcial Sample

this test case is generated manually, which contains a special graph and which have plenty of shortest path , and all they are cross together, which is a hard to work out the result of depth-first-search.
This test case is too larger, we only provide part of the input in the report, you can find the full case in the same folder.

```
50 61 61 244
16 14 24 24 46 ...
0 1 1
0 2 1
1 3 1
2 3 1
...
```

3.2 Test Script

we use the following script to test the output of our program.

```
#!/usr/bin/python2
import commands
for i in range(1,7):
```

```
    (status, output) = commands
        .getstatusoutput('./a.out < p2.test%d.in' % i)
    print "Test Case%d : %s" % (i, output)
```

and we use the following script to test the running time of our program, which makes use of the time module of python.

```
#!/usr/bin/python2
import commands
import time
for i in range(1,7):
    start =  int(round(time.time() * 1000))
    (status, output) = commands
        .getstatusoutput('./a.out < p2.test%d.in' % i)
    end = int(round(time.time() * 1000))
    print "Test Case%d : %d ms" % (i, end - start)
```

3.3 Test Output

According our test, our program passed all test case correctly and quickly, here is the output of each test case generated by our program.

```
Test Case 1: 3 0->2->3 0
Test Case 2: 0 0->1 1
Test Case 3: 348 0->1->2->3->4 ... 47
Test Case 4: 0 0->95->167->6 102
Test Case 5: 22 0->60 0
Test Case 6: 18 0->1 ... 13
```

3.4 Running Time

After the validation of the correctness of our program, we also test the running time of our program, through the test script written in python, we got the running time of each case

```
Test Case1 : 6 ms
Test Case2 : 8 ms
Test Case3 : 7 ms
Test Case4 : 216 ms
Test Case5 : 375 ms
Test Case6 : 2071 ms
```

All of the result is tested on the following computer:
Hardware: Inter Core-i5 4230m / 8G DDR3 / SSD
System: Arch Linux x86_64, kernel: 4.8.8-2
Compiler: gcc version 6.2.1 20160830 (GCC)
Compile Command: gcc -O2 -std=c99

3.5 Conclusion

After our test, it can prove that our program can work out the correct result of each cases we provided, and it can work out at a acceptable time, so we can confirm that, in most cases, out program can work out the result correctly and efficient.

4 Analysis and Comments

We first use theoretical analysis to deduce the growth of function of a certain algorithm, and then do validation in terms of the implementation in C code. Finally, we make comparison between the result from analysis and the result from performance test, in order to make it be accurate.

4.1 Dijkstra Algorithm

4.1.1 Time complexity

Bounds of the running time of Dijkstra's algorithm on a graph with edges E and vertices V can be expressed as a function of the number of edges, denoted $|E|$, and the number of vertices , denoted $|V|$, using Big O notation. How tight a bound is possible depends on the way the vertex set Q is implemented. In the following, upper bounds can be simplified because $|E| = O(|V|^2)$ for any graph, but that simplification disregards the fact that in some problems, other upper bounds on $|E|$ may hold.

For any implementation of the vertex set Q, the running time is in $O(|E| \cdot T_{\mathrm{dk}} + |V| \cdot T_{\mathrm{em}})$, where T_{dk} and T_{em} are the complexities of the "decrease-key" and "extract-minimum" operations in Q, respectively . The simplest implementation of Dijkstra's algorithm stores the vertex set Q as an ordinary linked list or array, and extract-minimum is simply a linear search through all vertices in Q. In this case, the running time is $O(|E| + |V|^2) = O(|V|^2)$.

For sparse graphs, that is, graphs with far fewer than $|V|^2$ edges, Dijkstra's algorithm can be implemented more efficiently by storing the graph in the form of adjacency lists and using a self-balancing binary search tree, binary heap, pairing heap, or Fibonacci heap as a priority queue to implement extracting minimum efficiently. To perform decrease-key steps in a binary heap efficiently, it is necessary to use an auxiliary data structure that maps each vertex to its position in the heap, and to keep this structure up to date as the priority queue Q changes. With a self-balancing binary search tree or binary heap, the algorithm requires $\Theta((|E| + |V|) \log |V|)$ time in the worst case; for connected graphs this time bound can be simplified to $\Theta(|E| \log |V|)$. The Fibonacci heap improves this to $O(|E| + |V| \log |V|)$.

When using binary heaps, the time complexity is lower than the worst-case , assuming edge costs are drawn independently from a common probability distribution, the expected number of "decrease-key" operations is bounded by $O(|V| \log(|E|/|V|))$,giving a total running time of $O(|E| + |V| \log \frac{|E|}{|V|} \log |V|)$.

4.1.2 Space complexity

For naive dijkstra algorithm, the space complexity is simply $O(V)$, which is used to storage the distance and the visit flag.

When we use the heap optimization, the space consumption will improves, but the heap still occupy the space which is linear relation about the the total number of points, which is also $O(V)$, so the total function of growth of the space consumption of dijkstra algorithm itself is $O(V)$.

In fact, when we think of the space consumption of a shortest path algorithm, we cannot ignore the space that the graph itself occupied, so when we use the Adjacency matrix to storage the graqh, we got the space Complexity $O(V^2)$, and when we use the adjacency list, the space complexity is $O(E)$. so the total space complexity of dijkstra algorithm using the adjacency list is $O(E + V)$

4.2 Bellman-Ford Algorithm

the Shortest Path Faster Algorithm is a optimization of Bellman-Ford Algorithm, to be more general, we will analyse the time complexity of the Bellman-Ford Algorithm

4.2.1 Time Complexity

Bellman–Ford is based on the principle of Relaxation, in which an approximation to the correct distance is gradually replaced by more accurate values until eventually reaching the optimum solution. In both algorithms, the approximate distance to each vertex is always an overestimate of the true distance, and is replaced by the minimum of its old value with the length of a newly found path.

Different from Dijkstra's algorithm uses a priority queue to Greedy algorithm select the closest vertex that has not yet been processed, and performs this relaxation process on all of its outgoing edges;

by contrast, the Bellman–Ford algorithm simply relaxes all the edges, and does this $|V|-1$ times, where $|V|$ is the number of vertices in the graph. In each of these repetitions, the number of vertices with correctly calculated distances grows, from which it follows that eventually all vertices will have their correct distances.

This method allows the Bellman–Ford algorithm to be applied to a wider class of inputs than Dijkstra. Bellman–Ford runs in $O(|V| \cdot |E|)$ Big O notation, where $|V|$ and $|E|$ are the number of vertices and edges respectively.

4.2.2 Space Complexity

the analysis of space complexity of Bellman-Ford algorithm is simliar to the dijkstra Algorithm, so here is just the Conclusion, the space complexity of Bellman–Ford itself is $O(V)$, and the total complexity of Bellman–Ford algorithm is $O(V + E)$.

4.2.3 Optimization in SPFA

the SPFA algorithm using the queue Optimization to decrease the count of relaxes, but in fact, under the worst circumstance, the SPFA algorithm will fallback to the general time consumption of naive Bellman–Ford Algorithm,

so we can regard the time Complexity of Bellman–Ford algorithm, which is $O(|V| \cdot |E|)$ using the Adjacency list.

The space Complexity of SPFA is same as the Bellman–Ford algorithm.

4.3 Depth First Search

4.3.1 Time Complexity

The Depth First Search problem is a well-know P-complete Problem, which have a high time Complexity and space complexity, The Time complexity of DFS differs according to its application area. In theoretical computer science, DFS is typically used to traverse an entire graph, and takes time $(|V| + |E|)$, linear in the size of the graph.

4.3.2 Space Complexity

Depth first search also uses space $O(|V|)$ in the worst case to store the stack of vertices on the current search path as well as the set of already-visited vertices.

4.4 Validation

In order to verify the Conclusion from the theoretical analysis of time complexity, we using some test script to test our program, in terms of the variation of the input data size, we can get a N-T relation graph, by the analysis the figure, we can verify the Conclusion from the theoretical deduction.

4.4.1 Test Script

Here is our test script in python, for improving the accuracy of time, we ignore the input and output time, and intgrate the time statistic funcion into the original program.

```
#!/usr/bin/python2
import commands
import math
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
testN = [ 10 * i for i in range(1,50)]
testM = [ 50 * i * i for i in range(1,50)]
avg = 3 # test count
def test(n,m):
    sum = 0
    for i in range(0, avg):
        (status, output) = commands.
            getstatusoutput('python2 p2.in.py %d %d %d %d > test.in'
            % (100,n,m,100) ) # the generate data program
        (status, output) = commands.
            getstatusoutput('./a.out < test.in')
        # the std out is the running time
        sum += int(output)
```

```
        return sum / avg

ret = map(lambda i: test(testN[i],testM[i]),range(len(testN)))
plt.plot([500*i*i*i for i in range(1,50)],ret)
plt.savefig('result.png') # draw and output the figure
```

4.4.2 Validation Result

when we using the random generated data (see Chapter 3), the count of shortest path is limited, so the major consumption of our program is the shortest path algorithm.After running the test script, we can get the following result figure

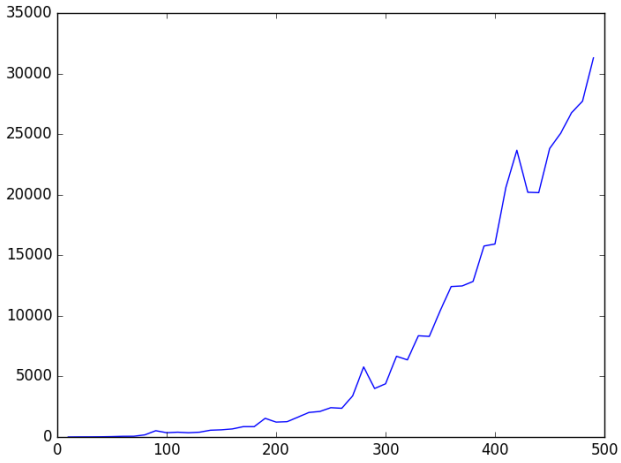

Figure 1: Performance on Random Data

and we can get the square root of the time, we can get a better figure to analysis.

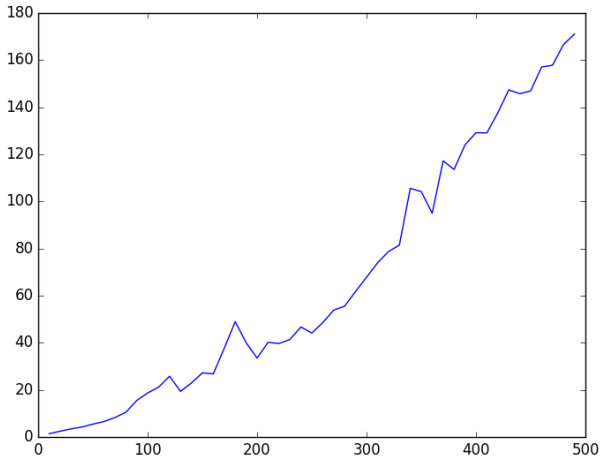

Figure 2: Performance on Random Data (Root)

we can found that, the result of our $N - \sqrt{T}$ function is similar to the linear function, which can prove that our program nearly has a time complexity of $O(N^2)$, and from the theoretical analysis, we can found that the time complexity of dijkstra algorithm is $O((|E| + |V|) \log |V|)$., which is equal to $O((N + N^2) \log N) = O(N^2)$, which is satisify the result of our theoretical analysis.

4.5 Comments

In general, the test result quite matches our expectation. Our Program perfectly finished the requirement of the given problem, and work out the result correctly and efficient, according to the theoretical analysis, we can prove the time Complexity and the space consumption of out program is acceptable, which is required by the online judge system, and the result of the validation shows that the result of our theoretical analysis is correct at a large probability. In details, the Implementation of the shortest path algoirthm by us shows a result that the dijkstra algoirthm with heap optimization is Generally fast than the Bellman–Ford or SPFA algoirthm, so it is a great idea to use dijkstra in the conventional situation. As to the depth first search algoritm, although it have an exponential time complexity, but in reality, the count of shortest path is limited, so we may not face the circumstance that much shortest path exists, and the time consumption of shortest path algoritm will be the major factor of the running time of our program.

5 Appendix

5.1 Source Code

Code 8: Source Code

```c
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

#define  MAX  500           // the max number of stations

/*Path is used to store the information about the path
from PBMC(0) to other stations.*/
typedef struct __Path{
    int last[MAX + 1];      //the last points
    int time;               //the mininal time
    int top;                //the number of last points
}Path;

/*Road is used to store the roads(or edges) information
for a certain start*/
typedef struct __Road{
    int time;               //the time
    /*the end, while the start will be stored as the index
    of g_roads*/
    int next_point;
    //the pointer for another road for that certaion start
    struct __Road* next_road;
}Road;

//same as cMax, N, sP, M in the description in pta
int g_cMax, g_N, g_sP, g_M;
/*the x numbers of bikes that are sent and returned of
the number of last points(top)*/
int g_finalSend = INT_MAX, g_finalRet = INT_MAX;
//the bikes that each station stores
int g_bikes[MAX + 1];
//the current shortest path to station i
Path g_paths[MAX + 1];
/*g_roads[i] for all roads that started with station i,
this is a Adjacency List */
Road* g_roads[MAX + 1];
//all the station in the required path
int g_finalRoute[MAX + 1];

//add a road into the adjacency list
void add_road(int a, int b, int t);
//Shortest Path Faster Algorithm
void spfa(int src);
/*used to select the required path among the shortest
roads, using dfs*/
void selectPath(int point);

/*The Shortest Path Faster Algorithm (SPFA) is an improv-
ement of the Bellman-Ford algorithm which computes single-
```

```
source shortest paths in a weighted directed graph.*/

/*the basic idea of spfa is that each vertex is used as a
candidate to relax its adjacent vertices. Once  a vertex
is relaxed then put that vertex into the queue and the
vertexs in the queue are the potential candidate vertexs.
The procedure goes on until there is no vertexs can be rel-
axed.*/

void spfa(int src){
    int now_station = src;
    int now_temp_queue[MAX * 4];
    int front = 0;
    int rear = 0;
    now_temp_queue[front] = 0;
    front++;

    while(front != rear){
        now_station = now_temp_queue[rear];
        rear++;
        /* use now_station as the candidate vertices to relax
        vertices in its Adjacency List */
        for(Road* i = g_roads[now_station];
            i != NULL;
            i = i->next_road){
            /* top acts as a reference to
            g_paths[i->next_point].top,the number of the last
            point of i on its paths to PBMC */
            int top = g_paths[i->next_point].top;
            //once the vertex can be relaxed
            if(g_paths[i->next_point].time
               >
               g_paths[now_station].time+i->time){
                //anbandon all the previous paths
                top = 0;
                g_paths[i->next_point].top = 0;
                g_paths[i->next_point].last[top] = now_station;
                g_paths[i->next_point].time
                 =
                g_paths[now_station].time + i->time;
                now_temp_queue[front] = i->next_point;
                front++;
            }
            //once the vertex has a same time-cost path
            else if(g_paths[i->next_point].time
                      ==
                      g_paths[now_station].time + i->time){
                g_paths[i->next_point].top++;
                top++;
                g_paths[i->next_point].last[top] = now_station;
```

```
            }
        }

    }
}

//this is a dfs way to selectPaht
void selectPath(int point){

    //route is used to store the present route.
    //top is the bound.
    static int top = 0;
    static int route[MAX + 1];

    /*while point is not PBMC, add the point into the route,
    notice that the route is reverse*/
    if(point != 0){
        route[top] = point;
        top++;
        int tempTop = top;
        for(int i = 0; i <= g_paths[point].top; i++){
            selectPath(g_paths[point].last[i]);
            top = tempTop;
        }
    }

    /*calculate the send number and ret number of the present
    route, it starts at (top -1) beacuse the route is reverse*/
    if(point == 0){
        int send = 0;
        int ret = 0;
        int temp = 0;
        /*start from (top-1) because the route is reverse.
        temp is for the bikes needed in each passing by
        station. (if it is negative, it represents the bike
        we get). The max of temp along the route is what we
        must bring from the start.*/
        for(int i = top - 1; i >= 0; i--){
            temp += g_cMax / 2 - g_bikes[route[i;
            if(temp > send){
                send = temp;
            }
        }
        /*ret for the number of bikes that we should return.
        when the circulation above is down, the temp should
        be what we get(when negative) or need (when positive)
        from the along stations without send any bikes from
        the start.so the return number is (send - temp).*/
        ret = send - temp;

```

```
151         /*while the current route is "better" than the current
152         final route, replace the final route with the current
153         route.*/
154         if(send < g_finalSend
155            ||
156            (send == g_finalSend && ret < g_finalRet)){
157             g_finalRoute[0] = 0;
158             for(int i = 1; i <= top; i++){
159                //notice that the current route is still reverse
160                g_finalRoute[i] = route[top - i];
161             }
162             g_finalSend = send;
163             g_finalRet = ret;
164         }
165     }
166 }
167
168 //add a road to an Adjacency List
169 void add_road(int a, int b, int t){
170     Road* road_a = (Road*)malloc(sizeof(Road));
171     road_a->next_point = b;
172     road_a->time = t;
173     road_a->next_road = g_roads[a];
174     g_roads[a] = road_a;
175 }
176
177 //the main routine
178 int main(int argc, char const *argv[]){
179     int a, b, t;
180
181     scanf("%d %d %d %d", &g_cMax, &g_N, &g_sP, &g_M);
182     for(int i = 1; i <= g_N; ++i){
183         scanf("%d", &g_bikes[i]);
184     }
185
186     //read roads into g_roads
187     for(int i = 0; i < g_M; i++){
188         scanf("%d %d %d", &a, &b, &t);
189         add_road(a, b, t);
190         add_road(b, a, t);
191     }
192
193     //initialize all the paths
194     for(int i = 1; i <= g_N; i++){
195         g_paths[i].time = INT_MAX;
196     }
197
198     //use spfa, set the start as 0
199     spfa(0);
200
```

```
201     selectPath(g_sP);
202
203     //output the result
204     printf("%d ", g_finalSend);
205     for(int i = 0; i < g_N + 1; ++i){
206         if(g_finalRoute[i] == g_sP){
207             printf("%d ", g_finalRoute[i]);
208             break;
209         }
210         else{
211             printf("%d->", g_finalRoute[i]);
212         }
213     }
214     printf("%d", g_finalRet);
215     return 0;
216 }
```

5.2 Declaration

We hereby declare that all the work done in this project titled "Public Bike Management" is of our independent effort as a group.

5.3 Duty Assignments

Programmer: Duan Fuzheng
Tester: Zhang Chengyi
Report Writer: Tan Qiye