Algorithmic Problem Solving, BSc (Spring 2023) - Group F

Gergo Gyori[1], Jacob Andreas Sneding Rohde[2], and Markus Sibbesen[3]

[1]gegy@itu.dk
[2]jacro@itu.dk
[3]mksi@itu.dk

May 2023

1 Main

1.1 Problem Description

You invited your friends for dinner in the evening and want to impress them with a multi-course menu. Luckily, you are in possession of a fully automated robot kitchen. The guests arrive at 18, and you get home from work t minutes before that.

You have found the perfect recipe, consisting of several distinct steps, such as chopping onions, kneading dough, preheating up oven etc. Some of these steps depend on others being completed; you can't mix all the chopped vegetables, if you haven't chopped them yet. Your fully automatic kitchen can, however, work on an unlimited amount of steps simultaneously.

You're not sure if you're gonna be able to make it in time with all those steps, but luckily your kitchen comes with a speed dial, which can be turned down to reduce the time taken to complete certain steps. Now some steps, like waiting for the dough to rise, cannot be sped up, but shaping the dough can.

The input consists an integer $1 \leq n \leq 1000$, the amount of steps in the recipe. The next n lines consist of the steps in the recipe with each line being of the form

$$N : T : V : D$$

where N is the name of the step (an integer $0 \leq N < n$), T (an integer $0 \leq T \leq 10000$) is the base time in minutes to complete the step, V is either "V", meaning variable, or "C", meaning constant, denoting whether the step can be sped up by turning the dial, and D is a space separated list of all the steps this step depends on (max 10).

The next $1 \leq m \leq 100$ lines consist of test cases each of which is an integer t, representing the amount of minutes before the guests arrive that the kitchen is turned on. For each of these, output the precise value the dial needs to have in order for the recipe to be completed the moment the guests arrive. The dial takes the floating point value d, $0.0 \leq d \leq 1000.0$, where $d = 245.5$ means each variable step takes 24.55% as long as the base time. The absolute difference between your answer and the correct value needs to be < 0.01. If the slowest time $d = 1000.0$ is already less than the time limit, output 1000, and if the fastest time $d = 0.0$ is not fast enough, output "Impossible".

```
10
8:50:V:9 0 5
9:40:V:6 5 3 1 2
6:16:V:5 3
0:33:C:5 3 1 7
5:43:V:4 3 1 2
4:85:C:3 1
3:42:V:1
1:1:C:
7:15:C:
2:100:C:
240
```

```
806.282722513089
```

Sample 1: input (left) and output (right)

```
5
0:17:V:
4:2:V:2 3 1
1:14:V:0
3:3:C:
2:8:C:3
13
19
4
11
```

```
393.93939393939394
575.7575757575758
Impossible
0
```

Sample 2: input (left) and output (right)

1.2 Algorithm

The steps of the recipe form a directed acyclic graph (DAG). The Algorithm for solving the problem consists of 4 steps:

- Topologically sorting the graph.
- Finding the longest path (the critical path)
- Binary searching through different dial values, corresponding to different critical paths, to find a dial value that results in the correct critical path length.
- Computing the precise value analytically once a range is identified that 1) contains the true value, and 2) has the same critical path.

1.2.1 Topological Sort

For each node, we maintain a count of its incoming edges. We also maintain a list (queue) of nodes with 0 incoming edges. Until the list is empty, we repeatedly:

- Pop an element from the list, adding it to the topological sort
- Loop through its outgoing edges, subtracting 1 from the destination node's count of incoming edges
- Add destination nodes whose count has reached 0 to the list.

1.2.2 Critical Path

For each node in the topological sort, we compute the longest path from the beginning to it, defined by the sum of the `time` attributes of the nodes that the path touches. This is done in a bottom up dynamic programming manner, by recognizing that the length of the longest path to a node will be the maximum of the longest paths to its dependencies (the nodes at the source of its incoming edges) added to its own `time` value.

Since we have a topological sort, it can be done bottom up by looping through each node in the topological ordering, computing the above mentioned longest paths.

By recording the dependency node with the maximum longest path for each node, the critical path can be found by working backwards from the node in the graph with the maximum longest path to it.

1.2.3 Binary Search and Analytical Solution

To find the exact dial value corresponding to the intended time to complete, we binary search through the different possible values. We recognize that the relationship between dial and time is a piece wise linear function (figure 3), where each linear section corresponds to a particular critical path. Because of this, when we have identified two dial/time pairs, for which the correct times lies between, with the same critical path, an analytical solution is possible. When binary searching, if the current step's critical path is the same as the one from the previous step and the correct target time is between this step's computed time and the one from the previous step, the analytical solution is computed

$$d_c = \frac{(t_c - t)}{(t_p - t)/(d_p - d)} + d,$$

where (d_c, t_c) are the correct dial and time value, (d, t) are the current dial and time values and (d_p, t_p) are the previous dial and time value.

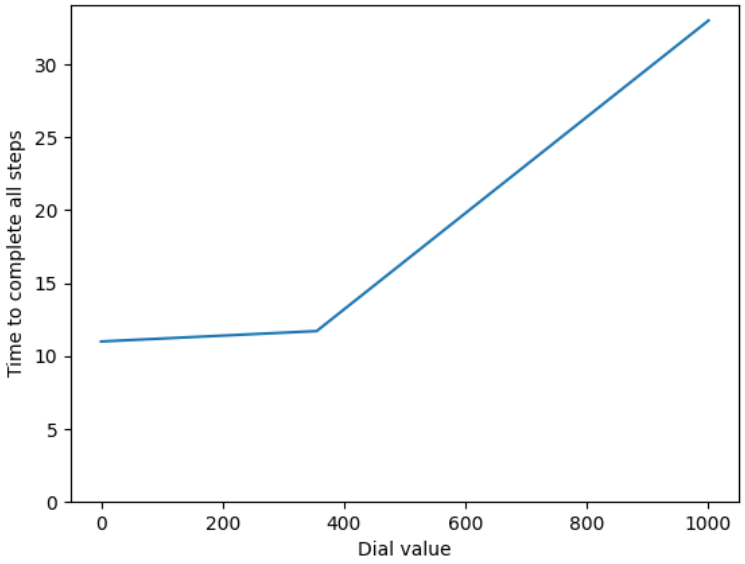

Figure 3: The piece-wise linear function describing the relationship between dial value and time to complete. This particular graph corresponds to sample input 2.

1.3 Input Cases and Wrong Solutions

There are two correct solutions, one solution that is too slow, and one solution that is wrong.

1.3.1 Slow Solution

The slow solution uses DFS to traverse the network from all the nodes without dependencies, adding together the `time` attributes of the nodes touched on the path, similar to the critical path method above. It then returns the highest of these sums to get the length of the longest path. This will give the correct answer if it is allowed to revisit already visited nodes. This is however very slow. Consider the case where the graph is arranged in layers of 2 nodes, each node depending in both of the previous layer's nodes. Here the time-complexity is $O(2^{\frac{n}{2}})$. We added an input of this arrangement to catch slow solutions of this manner.

1.3.2 Wrong Solutions

The wrong solution similarly uses DFS, but without revisiting any nodes. This is faster ($\sim O(n + n * d)$, where d is the amount of dependencies for each node), but is not guaranteed to yield the correct longest path. We randomly generated large inputs, which in practice always catch this mistaken implementation.

1.3.3 Correct Solution

The two correct implementations use topological sorting as described above. Using the analytical implementation is on average ~ 2 times faster than the implementation that doesn't take the analytical shortcut but instead binary searches until it has found

a satisfactory dial value. We did not find that this amount of speed up was enough to mandate the analytical approach, since arbitrary implementation details may have similar effects on the completion time.

1.3.4 Generation of Inputs

To generate the input we have a function that takes n (number of vertices), `max_time`, `num_test_cases`, and `max_connections` (max in-degree).

We iteratively create n nodes, each of which with between 0 and `max_connections` dependencies from the nodes already created, and give the node a random time between 1 and `max_time` and a random variability. We generate test cases corresponding to `num_test_cases`, by calculating the critical path lengths for dial value 0 and 1000, respectively, and picking a random value in that range, interspersing impossibly low values randomly in-between.

1.3.5 Choice of Model Parameters

While we did not achieve a working implementation ourselves, we believe there might be a way to compute the dial and time values of all the "breaks" in the function on figure 3, during the topological sorting. To incentivize users to try this approach, we put many test cases (100) for the same graph topology, since the "breaks" can be reused to quickly find an analytical solution, leading to a significant speedup.

The remaining limits were chosen in a more empirical manner. The maximum time for each step (10000 minutes) is arbitrary, and should not affect the time complexity. The maximum number of nodes and the maximum number of dependencies for each node were limited to 1000 and 10, respectively, to make the accepted solution complete in about 1 second

2 3 other problems

2.1 1. Minimum Spanning Tree

Link

The input consists of several test cases. Each test case starts with a line with two non-negative integers, $1 \leq n \leq 20\,000$ and $0 \leq m \leq 30\,000$, separated by a single space, where n is the numbers of nodes in the graph and m is the number of edges. Nodes are numbered from to $n-1$. Then follow m lines, each line consisting of three (space-separated) integers u, v, and indicating that there is an edge between u and v in the graph with weight $-20\,000 \leq w \leq 20\,000$. Edges are undirected. For every test case, if there is no minimum spanning tree, the output is the word *Impossible* on a line of its own. If there is a minimum spanning tree, then first output is a single line with the cost of a minimum spanning tree. On the following lines output is the edges of a minimum spanning tree. Each edge is represented on a separate line as a pair of numbers, x and y (the endpoints of the edge) separated by a space. The edges should be output so that $x < y$ and should be listed in the lexicographic order on pairs of integers. If there is more than one minimum spanning tree for a given graph, then any one of them will do.

After reading the input, the edges are sorted based on their weight in ascending order using the *sort*() method. This ensures that edges with the minimum weights are considered first during the construction of the minimum spanning tree. To keep track of *disjoint* sets of nodes, the Union-Find algorithm is employed. A disjoint

array disjoint is initialized, where each node is initially its own parent (-1) indicates a node is the root of its set).

The solution then proceeds to iterate over the sorted edges and attempts to add each edge to the minimum spanning tree. It checks if the two endpoints of the edge belong to different sets using the $find()$ function. If they have the same parent, adding the edge would create a cycle, so the edge is skipped. Otherwise, the two sets are joined using the $join()$ function, and the weight of the added edge is accumulated.

After considering all the edges, the solution checks if the resulting minimum spanning tree is connected by counting the number of disjoint sets (represented by -1 in the disjoint array). If there is more than one disjoint set, it means the minimum spanning tree is not possible, and "$Impossible$" is printed.

If the minimum spanning tree is connected, the total weight of the tree is printed. Then, the edges of the minimum spanning tree are printed in lexicographic order, sorted based on the endpoints of each edge.

The solution works within the time limit of the problem because it utilizes an efficient sorting algorithm ($sort()$ method) to sort the edges based on their weights. The Union-Find algorithm efficiently handles the operations of finding the parent and joining sets, ensuring efficient construction of the minimum spanning tree. The time complexity of the sorting algorithm is $O(m * logm)$, where m is the number of edges, while the Union-Find operations have a time complexity of $O(log(n))$. This combination of efficient algorithms and data structures allows your solution to handle large input sizes effectively.

In terms of worst-case inputs, the time complexity can be affected by the number of nodes and edges. If the number of nodes is large and the graph is highly connected with many edges, the time complexity can approach $O(n^2)$ or $O(m^2)$.

2.2 2. Knigs of the Forest

Link

Every year a tournament is held and the strongest moose wins. The size of the tournament pool is constant, so every year a moose is added and the winner of the year exits. The problem is to output the year that Karl-Älgtav wins, or output "unknown" if data provided is not enough.

First line contains k ($1 \leq k \leq 10^5$) and n ($1 \leq n \leq 10^5$), with k is size of tournament pool and n is the number of years for which sufficient information is supplied for determining when Karl-Älgtav wins.
Next line is y ($2011 \leq y \leq 2011 + n - 1$) and p ($0 \leq p \leq 2^{31} - 1$), with y being year Karl-Älgtav entered the tournament and p is his strength.
Then the following (n+k-2) lines provide the other tournament contestants in the same format as for Karl-Älgtav. k moose enter in 2011 and the rest enter on unique years

The code implements a solution using a priority queue to store and retrieve the strengths of moose, allowing for efficient retrieval of the moose with the highest strength (or lowest negated strength) during the tournament. Having the strengths of the moose stored in a list sorted by the year the moose is added to the tournament makes it possible to iterate through the list, add the new moose to the priority queue and retrieve the highest priority element to check if it is Karl. If Karl is found on the

i'th iteration, he is the winner of the i'th year since the annual tournament started in 2011.

The priority queue used uses the binary heap data structure, with insertion and deletion time O(log z), where z is size of the heap. When the PQ is constructed n deletions occur, with the size staying constant, so O(n * log k). k and n have max value of 100'000, so that means 100'000 * log (100'000) = 1'151'293.
The construction of the PQ is stricly smaller than that, because all insertions will be insertions into a PQ that has fewer than k elements, and can thus be ignored.
The amount of operations are thus of a lower order of magnitude than the 100 million to 1 billion that a normal computer can do in a second, and the time limit of 1 second is thus not exceeded.

2.3 3. Distributing Ballot Boxes

Link

The input contains at most 3 testcases. The first line of each test case contains the integers $N \leq (1 \leq N \leq 500\,000)$, the number of cities, and B $(N \leq B \leq 2\,000\,000)$, the number of ballot boxes. Each of the following N lines contains an integer $a_i, (1 \leq a_i \leq 5\,000\,000)$, indicating the population of the i^{th} city. For each case, the program should output a single integer, the maximum number of people assigned to one box in the most efficient assignment.

The solution employs a binary search approach to solve the problem. It starts by initializing a middle value of population of the biggest city in the array. Considering that each city should have at least one ballot box, the sum of all distributed ballot boxes is initialized by adding the middle value to the population of each city. If the number of available ballot boxes is lower than the distributed ballot boxes, the low variable is updated with the current middle value to continue the search. Otherwise, the high value is updated for the next middle value.

The solution works effectively because it utilizes binary search, which efficiently narrows down the search space to find the optimal value between the minimum and maximum numbers. Regardless of the input size, the binary search algorithm guarantees a logarithmic time complexity as it reduces the search space by half in each iteration.

Considering worst-case inputs, the time complexity depends on the value of the largest city population. The maximum value acts as an upper bound for the binary search, which ensures that the algorithm searches within a feasible range even for larger inputs.