Cost Constrainted Shortest Paths

CS 351

1 Description

Imagine the following situation. You are trying to travel across the country (from start city s to destination city d), but have a limited budget. To travel between two cities there may be many paths and different choices will have different tradeoffs between cost and time – e.g., taking a bus may be cheap but take a long time; renting a car may be fast, but expensive.

Essentially, what you have then is a doubly weighted graph. One weight is the *cost* c and the other is the *traversal time* t. We will assume that all *cost* and *traversal time* values are **non-negative integers**.

Your goal then is to determine the fastest way to get to your destination within your budget.

2 Algorithm Sketch

This section gives an overview of the main ideas behind an algorithm solving this problem. Part of your job will be to foramlize it into an actual implemtation.

The algorithm has a Dijkstra-like sturture with some important differences.

- In Dijkstra, we store a label `d[v]` at each vertex. For our problem we will have to store a *list* of candidate paths to v and the "signature" for each such path (i.e., the total cost of the path and the total traversal time of the path).
- For a vertex v, we only need to store "non-dominated" path signatures. A path is *dominated* if some other path is better in one of the dimensions and better (less than) than or equal in the other (i.e., there is no point in using the first path). We will call this set `S[v]`
- In a manner similar to Dijkstra, we will expand in non-decreasing order of *path cost.*
- Because of this order of expansion and the fact that `S[v]` contains only non-dominated signatures, we can order the set in *strictly increasing order* of cost and *strictly decreasing order of traversal time.*
- An invariant will be that `S[v]` will never have anything deleted from it. In other words, no path to v discovered in the future will make any of the signatures in `S[v]` dominated. The next bullet will clarify this.
- In addition to maintaining `S[v]`, we also maintain a binary heap which keeps track of the "frontier". Details follow.
  - The heap contains data elements of the form $<(c,t),v>$ which indidates that we have discovered a path from the source to v which has cost c and traversal time t.
  - Some of these path signatures sitting in the heap may turn out to be sub-optimal (dominated by some other path).
  - The heap (a min-heap) is ordered first by cost and second by time. This preserves the property that path signatures are added to the `S[]` sets in non-decreasing order of c. By secondarily, we mean that time is used as a tie-breaker in the heap (i.e., it is lexicographic).

- Unlike Dijkstra's algorithm, we will never do a `DecreaseKey()` operation. This is becaue for a particular vertex, we may have many candidate paths (signatures) sitting in the heap. When we see a new path, we just insert it.
- Also unlike Dijkstra's algorithm, when we remove a candidate signature $< (c, t), v >$ from the heap by a `DeleteMin()`, we need to make sure that it is not dominated by a previously found solution (i.e., one already in `S[v]`). **Important:** recall that `S[v]` is ordered in increasing order of c and decreasing order of t and by our invariant, all of the signatures are non-dominated; now, we have just removed $< (c, t), v >$ from the heap. When is this solution dominated by a solution already in `S[v]`? We already know that c from the newly extracted solution is at least as large as all c's in `S[v]`. Do any of them have smaller t-values? Well, the last entry in `S[v]` has the smallest t-value in `S[v]`, so if the new t-value from the heap is greater than any entry in `S[v]`, it must be greater than the *last* t-value in `S[v]`. What does this mean? You can check if the extracted solution is dominated by a prior solution by one comparison with the last entry in `S[v]`. If it is not dominated, you append it; if it is dominated, you discard it.
- Only when you have discovered a new non-dominated path as above, do you "expand" this solution and add new signatures to the heap. This is done in the natural way by examining edges leaving the vertex and "augmenting" the current path cost and time with those of the edge. Before inserting this new candidate into the heap, you might as well do a check to make sure it is not already dominated (as above). The algorithm still works if you don't do this check.

The above are the key ideas of the algorithm. Read it multiple times if it helps.

Recall that in the formulation, you have a budget. The above algorithm is the same regardless of budget, but when it is done, you examine the non-dominated solutions at the destination and pick the fastest one within your budget. (You can use the budget to make the algorithm a bit faster by stopping early when possible, but this is not required).

3 Pencil and Paper

Before starting your program, you will do the following exercises/problems.

(1) Construct an example instance of the problem with at least 5 vertices and at least 3 non-dominated paths from the specified source to the specified destination. Some of the paths should overlap (i.e., share common subpaths). Simulate the algorithm by hand. You need not simulate the individual heap operations, just maintain the contents of the heap clearly.

(2) Recall that cost and time values are non-negative integers.

- Let the sum of all edge costs be $C = \sum_{e \in E)} c(e)$
- Let the sum of all edge times be $T = \sum_{e \in E)} t(e)$
- Let $M = \max(C, T)$
- Using M, give an upper bound on $|S[v]|$. Recall that all cost values will be distinct (so will time values). Your bound will be pretty crude.
- Using this bound, give an upper bound on the number of heap operations (insertiona and deletions).
- Use this result to give an upper bound on the overall runtime of the algorithm.

4 Program Specifics

Your executable will be called `cpath` (for constrained paths) and will have four command line parameters.

The usage is "`cpath <file> <s> <d> <budget>`". So you are given a file contains the graph, the source vertex s (an int), the destination vertex d (an int) and a budget (also an int).

You will provide a makefile which produces the executable `cpath`).

File Format

The first line of the file will contain the number of vertices. Vertices are always implicitly numbered $0..|V|-1$.

After this there is a sequence of edges in the form "`u v c t`" which says there is an edge from u to v with cost c and traversal time t. For instance "`5 2 10 5` indicates that there is an edge from vertex 5 to vertex 2 with cost of 10 units and traversal time of 5 units.

Assumptions:

- You may assume that there is exactly one edge per line to simplify input parsing.
- Cost and time values are given as non-negative integers.

Sample Input File:

```
7
0 1 2 4
1 3 2 1
0 4 4 2
4 5 3 2
1 5 4 1
1 2 3 2
3 2 2 1
2 6 2 3
5 6 4 1
```

Output

After your program runs it will report the fastest cost-feasible path (if no feasible path exists, the program reports so). You should give the cost and traversal time of the path as well as the path itself (as a sequence of vertices). The format is up to you – just keep it simple and clear.

Things to remember

You can/should use the STL priority queue since:

- we don't need the `decrese_key` operation and
- we want to use "time" as a tie-breaker in the heap ordering and the heap you worked on in p2 would need to be modified to support this feature.