Program 2: Cost Constrainted Shortest Paths
CS 401

Early Submission (10% bonus): Wed April 28 by 11:59PM

On-time submissions: by 11:59PM Friday April 30

Late submissions (15% deduction): by 11:59PM, Sunday May 2

NOTE: pay close attention to the command line usage requirements for your submission!! Emphasis on "requirements"!!!.

Teams

First, you are allowed to work in teams of up to two for this assignment. (You get to choose your team).

You are not *required* to have a partner; the assignment is very much doable by one person.

Description

Imagine the following situation. You are planning a trip across the country (from start city s to destination city d). Your degrees of freedom in selecting a particular path are captured in a graph where vertices represent locations (cities, towns, arbitrary latitude/longitude coordinates) and edges represent "links" between locations – e.g., you might be able to take a bus from vertex u to vertes v.

Unlike the traditional shortest paths problem solved by, for example, Dijkstra's algorithm (where each edge (u, v) has a single weight $w(u, v)$, each edge (u, v) in our problem has *two* weights:

- a non-negative integer *cost* **c** and
- a non-negative integer *traversal time* **t**.

(**COMMENT**: the weights don't *have* to be integers for the algorithm you will develop and implement. But a runtime analysis alluded to in the last section of this handout is based on them being integers; the given graph class allows them to be floats for example).

Now suppose you have only so much money you are willing/able to spend on your trip – your budget. Your goal then is to determine the fastest way to get to your destination within your budget.

In order to solve this problem, you will compute a "tradeoff curve" of candidate paths from s to t (and analagous tradeoff curves for all other vertices in the graph).

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 single label `d[v]` for each vertex `v`. For our problem we will 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 (smaller) 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). In other words, we store the *tradeoff curve* of options we might consider for reaching destination `v` from the source. We will call this list `P[v]`

  Ordering: In practice, these lists will be arranged in *strictly* **increasing** *order of cost and strictly* **decreasing** *order of time.*

- 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 `P[v]` contains only non-dominated signatures, we can order (and build) the list in strictly increasing order of cost and strictly decreasing order of traversal time as stated above.

- An invariant will be that `P[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 `P[v]` dominated. (Said another way, once an option/signature is added to `P[v]` it is "permanent"). The next bullet will clarify this property.

- In addition to maintaining `P[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 with cost c and traversal time t.

  – 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 `P[]` lists 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 `decrease_key()` 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 (unless we can discard it at the moment we discover it). As a result, a simple priority-queue (heap) implementation that just supports `insert` and `delete_min` is sufficient.

  – Also unlike Dijkstra's algorithm, when we remove a candidate signature $<(c, t), v>$ from the heap by a `delete_min()`, we need to make sure that it is not dominated by a previously found solution (i.e., one already in `P[v]`). **Important:** recall that `P[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 `P[v]`? We already know that c from the newly extracted solution is at least as large as all c's in `P[v]`. Do any of them have smaller t-values? Well, the last entry in `P[v]` has the smallest t-value in `P[v]`, so if the new t-value from the heap is greater than any entry in `P[v]`, it must be greater than the *last* t-value in `P[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 `P[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.

Pencil and Paper

Before starting your program, you will do the following exercise. In the same folder as this handout, you will find a "worksheet." The worksheet includes a drawing of a graph which is also given as `g2.g` in the file format described in Program Specifics.

Your task is to simulate the algorithm on this graph with source vertex 0. (A supplemental lecture will be posted which performs the first few steps). Your simulation will of course also have to represent the dynamic contents of the heap (simply as a collection of `<(c,t),v>` options; when simulating a `delete_min` simply x-out or somehow mark-off the entry removed).

Program Specifics

You are allowed to pick a programming language of your choice to implement your solution provided it isn't anything too exotic (e.g., should at least be available on the CS servers). However, you have been provided a C++ graph class which you are welcome to use and modify as you desire.

Regardless of language chosen, your program must follow the following conventions for command line parameters.

REQUIREMENT: Your program/executable *must* be called `cpath` (for constrained paths) and will have four command line parameters.

The usage is "`cpath <file> <s> <d> <budget>`".

(Or perhaps something like "`java 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).

Again, following this command-line usage rule is a reqirement! If your program does not follow this rule, you will receive a (perhaps substantial) deduction.

File Format

A file representing a graph is given as 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.

The edges appear one-per-line.

You may assume that vertices have integer IDs `0..|V|-1`.

Assumptions:

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

Sample Input File:
Below are the contents of the graph file `g2.g` (as used in the worksheet).

```
0 1 2 4
1 3 2 1
0 4 4 2
4 5 5 1
1 5 4 1
1 2 3 2
3 2 2 1
2 6 2 3
5 6 4 1
4 1 6 1
1 4 1 5
5 2 6 1
3 6 10 0
```

Given Graph Class

You have been given a C++ graph class in the file `Graph.h` in the `src` folder. You are free to use and modify it as you see fit; you can use it as a model for your own graph data structures; or you can simply ignore it.

The given graph class parses a generalization of the format described above format where vertices are just strings. Thus, when that class reads a graph with an input line like "`5 2 10 5`", it interprets the vertices as having names `"5"` and `"2"` as strings. Internally, the graph class determines integer vertex IDs in `0..|V|-1` (so algorithms internally do not need to incur the overhead of fiddling with string vertex IDs). As you may expect, since the class expects arbitrary strings as user specified vertex names, if a vertex name happens to be something like `"5"`, the internally assigned vertex ID associated with this vertex is not necessarily going to be the integer 5.

Output

After your program runs it will do two things.

First: you will report the entire tradeoff curve of non-dominated paths from the specified source to the destination. This is simply a list of cost, time pairs in increasing order of cost and decreasing order of time (one per line with column labels, reasonable formatting, etc.). (You do not print the *actual* corresponding paths at this point.

Second: you 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 starting with the source vertex and ending with the destination). The exact format is up to you – just keep it simple and clear.

Submission

You will collect the following into a folder which you will then zip and submit via blackboard.

- Your worksheets from the warmup simulation exercise (one submission per team) as a pdf

- Your source code and related files
  - All of your source files and support files (e.g., a makefile).
  - A `README` file which must include: the team members; any instructions or details regarding compilation and running of your program. Make it easy on us!
- Each team will make only one blackboard submission (through one of the team member's account).

Tips and Discussion

The `priority_queue` class in the C++ STL will probably be useful for C++ programmers.

One thing you will have to figure out: how to perform "extraction" of a selected path as a post-process. As is so often the case, some additional bookkeeping information stored along with each "option" can make your life easier. Feel free to expand upon the basic `(c,t)` pairs stored in the option lists in `P[]` as you see fit.

Runtime and Complexity: FYI, it turns out that this problem is NP-hard. However (much like the subset sum algorithm you know so well), this algorithm has *pseudo-polynomial* runtime: the runtime is polynomial in the size of the graph *and* the numerical values of the problem instance (the edge cost and time labels). Analogously, recall that the runtime of the subset sum dynamic programming algorithm is polynomial in the number of given integers and `T`

The runtime analysis of this algorithm however isn't quite so simple. In short, based on the numerical values of the edge cost and time labels, we can derive upper bounds on the length of the option lists in `P[]` and the size of the heap: for example, since all of the costs (and times) in option list `P[v]` for some vertex `v` must be distinct and correspond to *simple* paths, we can derive an upper bound on its length. Unfortunately (or perhaps fortunatley), the resulting runtime bound is typically very pessimistic vs. observed behavior in practice.