COMPSCI220S1C2013 Assignment 4

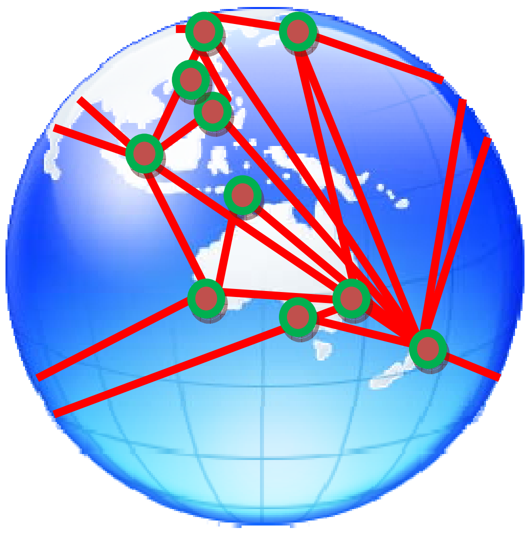
due 27 May 9:00 pm

One of the purposes that graph traversal is often used for is to search for a solution in an optimisation problem. This assignment is designed to let you solve such a problem and see the difference that the choice between DFS and BFS can make in a graph search with weighted edges.

Travel trouble

The general scenario is this: You have worked hard for your degree and now it's time to go and see the world. There's just one problem: As a poor student, you haven't got a lot of money. You want to get from Auckland to your destinations of choice as cheaply as possible. So you sit down and come up with a list of 50 cities all over the world that you'd like to go to (Auckland is the 51st place, so to speak, but you're already there). You know you won't be able to visit all of them but you'd like to know what your cheapest option is to get to each city, and which cities you'll be passing through on the way so you can stop off there.

This is where your headache starts. You look for the cheapest airfares between the cities involved. For some pairs of cities, there are none, but after a lot of work you end up with no less than 363 fares between pairs of the 51 cities on your list. You quickly realise that there are many ways of getting to most destinations: A few cities are directly reachable from Auckland, a few are reachable via a number of intermediate cities on your list to stop in. So how will you be able to find the cheapest way to go to each city?



So many routes! And that's just a few of them…

What you need is a solution. You sit down with a map and start drawing the possible ways to go. You draw lines between cities and the cities as big fat dots. You give up soon because you run out of ink.

Then you remember that sometime during your studies, you took this class called COMPSCI220. You remember seeing something quite similar there. The lecturer called it a “graph” and went on about “vertices” and “edges”. You remember being taught something about graph traversal. Could this help?

So you look at your old 220 notes and, sure enough, there’s something about DFS and BFS (for the purposes of this assignment, ignore that there are also other algorithms such as Dijkstra’s, Bellman-Ford or Floyd’s). So you identify the cities as the vertices in a graph, and the routes between to directly connected cities as the edges (we’ll assume here for a moment that the fare between city A and B is the same as between B and A, so we have edges, not arcs, and our graph is not a digraph). The cost of each airfare is the weight of the respective edge. You also figure out that the cost to reach a city via a particular path of routes is the sum of the airfares for these routes, i.e., the sum of the edge weights of the edges that make up the path.

Trouble is, you can’t quite figure out which algorithm is best for the job. So you decide to implement all three types and play around a bit.

Setting up the environment

This assignment assumes that you will be programming in Java. You will need to download **a personal template** for the algorithm program in the form of a Java class named **`TraversalProgram.java`**, as well as three auxiliary classes: **`City.java`**, which implements the class that represents cities (vertices), **`Route.java`**, the class that represents routes (edges), and **`Keyboard.java`**, a class you will probably remember from COMPSCI101. There is also a class **`TraversalApplication.java`** that represents the application itself – very much like you would have structured an application in COMPSCI101. For your convenience, there are also three classes that implement a stack and a queue of City objects: **`CityStore.java`**, **`CityStack.java`** and **`CityQueue.java`**. Have a look at them and the methods they provide, and use them as it fits your application.

The template and the zip file with the other classes are available from:

`http://www.cs.auckland.ac.nz/courses/compsci220s1c/assignments/2013S1C/a4/download/download.php`

You can compile the application as is and you will get core functionality: The program will print the list of cities it has stored, along with the outward routes and the associated cost for each city.

For your convenience, there is also a fully functional sample solution that uses data for hypothetical AUID 1234567 and that shows you what your output ought to look like. You may wish to start your assignment by downloading the template file for AUID 1234567 and using it to implement a solution that produces the same results as the sample solution, and then continue with your own template in each case.

You will work on `TraversalProgram.java` **only**, and this will also be the only file you submit. You will implement all your tasks as methods in this class. `TraversalProgram.java` contains a personalised set of cities and airfares that will let you develop and test your algorithms. Keep a backup of this file after each task is complete.

You must not change the `City.java` and `Route.java` files or the `CityStore.java`, `CityStack.java` and `CityQueue.java` files (and none of them should be submitted). Do not create additional class files.

Storing the graph

As the graph is relatively sparse, `TraversalProgram.java` (in conjunction with `City.java` and `Route.java`) stores the graph in a data structure very similar to an adjacency list: An array called `cities` represents the set of vertices as `City` objects, and each `City` object contains a (private) array named `routes` that represents the edges as `Route` objects. The only reason why we're using `Route` objects here rather than just the index of the destination in the `cities` array is that the edges have a weight as well as an endpoint, and the objects give us a way of storing this information.

All properties of cities, including the routes (edges), can be accessed via accessors and mutators. The `City` class makes mutators available that return the routes out of the city in the order they were added (`getRoutes()`) or as sorted by airfare (cheapest first). The same applies to the properties of routes. If there is no mutator for a property, you do not need it. Do not create such mutators yourself – any `City.java` or `Route.java` file you submit will be discarded and will not be used for marking.

You must not change this storage arrangement – it is provided for you. However, beyond this, the design of storage and data structures is your own!

Convenience methods: `TraversalProgram.java` makes a menu structure available in its `start()` method. This includes code to reset the cities to their "virgin" state (i.e., delete any stored costs or paths to the city from a previous search that are stored in the `City` object) and a method `printCitiesAndRoutes()` which will print the final result of your algorithms. Do not change the `start()` method.

Counting your steps

One step is defined as the inspection of a route (edge) to a destination vertex to see whether the vertex is white or not (count ONE step each time you inspect the edge, regardless of whether the vertex at the far end turns out to be white or not).

Save the number of steps in the `steps` object variable.

In the DFS and BFS algorithms that use the "cheapest edge first" strategy, you will need to pre-sort the routes. This is done implicitly by the `initWhiteCities()` method, whose return value is the number of comparisons taken by the simple in-place bubblesort here (a $O(n^2)$ algorithm, but justified in its use here because the lists of routes are generally quite short). Store this value in the **`sortSteps`** variable for the applicable algorithms only. This value should then also be the starting value for **`steps`**.

First task: DFS – pure depth first search [20 marks]

Implement the search as a depth-first traversal algorithm. To do so, add a method to do the DFS to your personalised template and add any other code you need as methods to **`TraversalProgram.java`**. Do not discriminate between routes on the basis of airfares just yet.

As you implement and try out your DFS algorithm, you should notice that it's not giving you optimal results: The way to get to some cities is pretty roundabout. As a result, the final airfare for many cities will be quite astronomical.

The reason for this behaviour is that depth first algorithms take you quickly far away from the root node. In a graph like ours, this often means that a vertex close to the root is reached well before the algorithm returns to the root, and so many of the root's edges may not get used. In fact, you may well find that only one of the dozen or so routes out of Auckland is actually used by DFS!

How many steps to expect? Every edge gets inspected in both directions, and only the first bullet point above applies here. So: twice the number of edges.

OK, so you think about how this might be fixed. The next solution you try is to get the price down by modifying your DFS algorithm to always look first at the cheapest airfare out of a city (lowest weight edge).

Second task: DFS with cheapest edge first [5 marks]

To implement this, use the **`getSortedRoutes()`** accessor rather than **`getRoutes()`** when you're looking at a city, which returns an array with routes sorted by airfare. The route with the lowest airfare is in the element with the lowest index in the array that is returned. You can either write a dedicated method for this task or modify the method you wrote for the first task so it can do both tasks depending on a parameter you pass.

As you implement and test this, you will notice that the total airfares to most destinations are now mostly lower, but still quite steep, and the number of stops decreases has also decreased a little, but not a lot. That's because most cheap edges are between cities in the same region, and we move away from the root a little more slowly than before, so we get to process more vertices closer to the root earlier on. However, you may still find that the algorithm uses only one route out of Auckland!

The number of steps will be the same as for pure DFS, plus the number of steps required for sorting. In principle, the "cheapest edge first" approach is one of many variants of a priority first search (as is pure DFS itself, where the highest priority value is always with the next reachable vertex that hasn't been pushed onto the stack yet.

Third task: DFS with path optimisation [10 marks]

You decide to try another tack: Use plain DFS, but with a twist. Whenever you try out an edge and discover a destination city at its end that has already got a path to it and a "price tag" attached, check whether you can reach it cheaper via the current edge. If so, update the price and path to that city. At this point, you also need to ensure that any other cities you have previously reached via this city are updated. You can do this by pushing the city back onto the stack. Again you may implement this via a separate method or via an amended version that can handle all DFS tasks.

As you implement and test this, you will find that you now get optimal cheapest paths indeed – all of a sudden, the OE looks affordable, and you don't have to change planes that often either! The joy only lasts until you look at the number of steps required: This is now much higher – expect between 50 and 80 times the number of steps as for pure DFS.

Why? Previously, we put each vertex on the stack exactly once, and so the total number of steps was limited by twice the total number of edges, because we could traverse each edge in two directions.

How bad is this for real decision making? If the number of edges is roughly $O(n^2)$, the indegree/outdegree of each vertex is O(n). If we arrive from a random direction (as we practically do with DFS), the probability that we initially arrive at a vertex via the cheapest path thus goes to 0 as the number of vertices and edges increases. The number of times that we will need to put the vertex back onto the stack is thus roughly proportional to the number of possible paths to it…which could be huge, conceivably up to O(n!) if the network is really dense. So this solution is not exactly scaleable!

Food for thought (to clarify understanding, not for submission): In this algorithm, is it possible for the same vertex to be in two positions on the stack at any one time?

Fourth task: DFS with cheapest edge first and path optimisation [5 marks]

You decide to launch one last-ditch effort on behalf DFS: You try to use a combination of cheapest edge first and your path optimisation technique. You hope that you'll arrive via the cheapest path first more often and that this will reduce the number of times you need to put a city back onto the stack.

Is this hope justified? Comment in your code above the method that implements this option.

Fifth task: BFS – pure breadth-first search [20 marks]

You now turn your attention to BFS. Implement a plain simple BFS that visits neighbour vertices in the order in which the routes were added.

As you implement this and try it out, you should notice that the number of steps will again be twice the number of edges, because you will be exploring each edge from either end. However, now you will notice that the airfares have come down considerably, and the suggested paths take you into all sorts of directions out of Auckland. But can we do better?

The number of steps should come as no surprise: twice the number of edges once again.

Sixth task: BFS with cheapest edge first [5 marks]

Again use the **`getSortedRoutes()`** accessor rather than **`getRoutes()`** when you implement this. You should notice that some paths change and some airfares get a little cheaper compared to plain BFS, but that the number of steps remains the same except for number of additional steps we now spend sorting. This number is $O(n^2)$ and so totally dominates the search.

Comment above the method you use to implement this task whether this strategy delivers optimal results.

Seventh task: BFS with path optimisation [15 marks]

The motivation here is again that of the third task, except that the underlying algorithm here is BFS, not DFS, so we put destinations that we can find cheaper back into the processing queue.

Food for thought again: In this algorithm, is it possible for the same vertex to be in two positions in the queue at any one time? If so, we'd need to prevent this from happening, of course, otherwise we'd process the same vertex twice.

As you implement and test this and compare with the third task, you should notice that while some of the paths may be different, the airfares to each city will be the same. However, this time, the number of steps will be much smaller. Why? Because BFS explores the immediate neighbours first, it is more likely to pick a good path early on, and as a result the city involved does not have to be reprocessed so frequently.

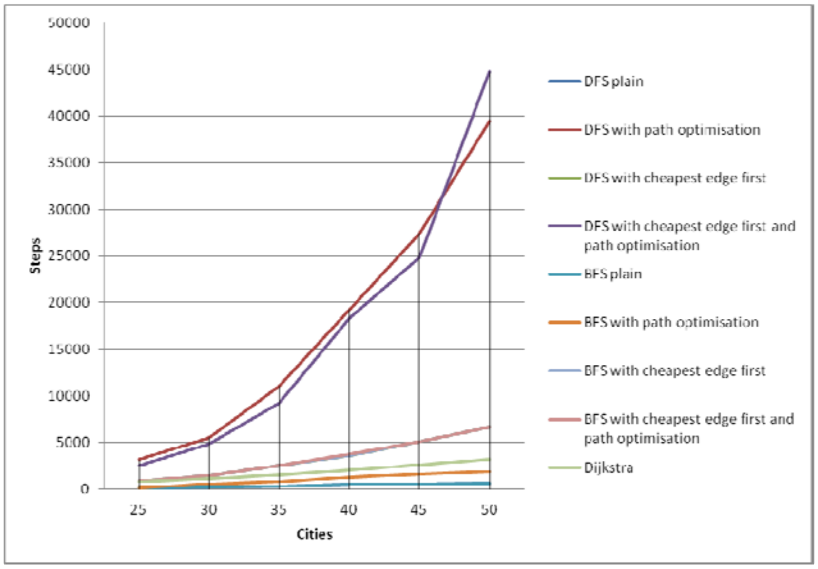
Take-home wisdom: If you have any reason for having to process vertices more than once in a graph traversal, make sure that the probability of this happening is minimised.

Eighth task: BFS with cheapest edge first and path optimization [5 marks]

Now that we are already finding optimal fares, the question is just whether we can get the number of steps down that it takes to find them. Can we? Implement this algorithm and comment on what you see, and why you think you may be seeing this result.

Tenth task: Evaluate the results [15 marks]

OK, by "evaluate" I don't mean "tell me what the cheapest ticket is". Instead, run your completed algorithms for 25, 30, 35, 40, 45, and 50 cities and their associated routes respectively and count the steps for each case. Enter the steps into a table (e.g., in MS Excel) and plot them as a function of the number of cities for each algorithm. You should get something vaguely like this:



Number of steps for AUID 1234567

Note that your data may result in slightly different curves – format is not important here, the overall behaviour of the algorithm is. Include your diagram in a PDF file which you submit along with your assignment (the PDF file must also include your name, UPI, and student ID number, and should be named **`steps.pdf`**). You don't need to include a result for Dijkstra's algorithm (I've shown this in the diagram above for comparison).

Getting started

Download the sample application and familiarise yourself with the way it works. Then look at the sample classes (you'll mainly use **`City.java`**, **`Route.java`**, **`CityStack.java`** and **`CityQueue.java`**) to see which methods they offer. Then download our template, save it as your **`TraversalProgram.java`** and start implementing the tasks one by one.

You can test that you're in the right ballpark by downloading a template for AUID 1234567 and programming there – the output produced by this should be exactly the same as in the sample application. **Note however that this solution will attract no mark – you must submit a class file with your own personalised data in it!**

Getting help

Read the Q&A below, if that doesn't help, ask the tutor or e-mail me: ulrich@cs.auckland.ac.nz

Submission

Submit your **`TraversalProgram.java`** and your **`steps.pdf`** via the assignment dropbox:

https://adb.auckland.ac.nz/

Please ensure that the file you submit reflects your own personal city and fare data (i.e., **do not accidentally submit something for AUID 1234567**). Late submission after 8.30 pm on the 19th will attract a penalty unless Sonny or I have given you an extension beforehand – and "beforehand" means applied for no later than the 18th, not half an hour before deadline ;-)

For completeness we are also required to point out the University's academic honesty policy to you:

http://www.auckland.ac.nz/uoa/cs-academic-honesty/

Questions and Answers

Q: *I'd like to use some Java data structures other than you have provided, from a common package. Is this OK?* **A:** Please don't – there is no guarantee that the marker's environment will have your package installed.

Q: *Can I use my own Java classes?* **A:** No. Please use the ones that are provided.

Q: *I would like to make a change to the City class. Is this OK?* **A:** No, please use all classes except `TraversalProgram` as provided.

Q: *Can I rename* `TraversalProgram`*?* **A:** For your own testing, yes, but once you submit your version must be called `TraversalProgram` and must be in a file called `TraversalProgram.java`.

Q: *Where do I put my own code?* **A:** There are two places you need to modify: You need to add methods for your graph traversal to `TraversalProgram.java` below line 57 in the file (where it says "PUT YOUR OWN METHODS HERE"). You will also need to add code to line 132ff. in the `start()` method of the `TraversalProgram` class to call these methods (where it says "PUT YOUR METHOD CALL HERE"). You do not need to write any other code or add other variables to the class itself.

Q: *Should there be anything else in the PDF file other than the plot with the curves for the number of steps for the various algorithms?* **A:** No, but if you would like to include any explanations for the marker then that is OK.

Q: *What if I count steps other than the ones you have asked for?* **A:** Please don't as this will cause problems with marking.

Q: *Why do I need to download a personalised template?* **A:** This is so we can ensure nobody copies anyone else's work. Your personalised template contains your personal graph.

Q: *The numbers I get for the steps are a little different from your sample solution. Is this a problem?* **A:** In most cases this is a "problem" brought about by the fact that the graph for your AUID is different from the one for the default AUID 1234567. A better indicator is to check whether the growth in the number of steps as the number of cities grows is similar to that in the sample application. If it's not, you have a real problem. One thing to watch out for is whether you have included the number of steps for sorting edges in the total step count for those algorithms that use "cheapest edge first".

Q: *Will I get a deduction in the last step if I haven't implemented some of the algorithms?* **A:** No, as long as you (a) make it clear in your code (e.g., by means of a printout "to be implemented" or somesuch) that you haven't got the code. In this case, you may substitute the number of steps from the sample application (which uses the default AUID 1234567). Plain copies of my plot are not acceptable, though.

Q: *Do we need to reproduce the information in the final path printout precisely, i.e., does the format of the path information have to match the sample 100%?* **A:** No, but the essential information needs to be there for each destination: total fare, cities along the path, and fares between cities.

Q: *Does our program have to find exactly the same results as the sample program you've provided (in terms of cost)?* In the case of your own graph, the answer is no – but the relative orders of magnitude should be similar. In the case of the graph for the sample AUID, there could be a number of reasons if the number of steps isn't the same as mine. If your number if steps is higher, then the most likely reason is that you always resubmit a city whose cost you are updating with a cheaper fare back into the BFS queue - even if it's already in the queue and would already be processed again regardless. This would lead to a small reduction in marks. If your number of steps is smaller, then either you're not counting right, or you've come up with a better algorithm than me, in which case you should be able to replicate mine & explain why yours gets away with fewer steps ;-)