MATH1058 coursework [40 marks]

Dave Waddington

April 10, 2019

1 Implementation

1.1 [5 marks] Pseudocode for the $O(n^3)$ version

Report the pseudocode (in this case the actual code) of your implementation "`dijkstra1`".

```
1   def dijkstra1(successors, s):
2     S = []
3     S.append(s)
4     n = len(successors)
5     L = dict()
6     L[s] = 0
7     P = dict()
8     P[s] = '-'
9     predesessor = s
10    while len(S)<n:
11        shortestDistance = numpy.Infinity
12        nextNode = s
13        # because of the way I have written this if I am given a non traversable
14        # graph it may return some infinite distances....
15        for i in S:
16            for j, length_ij in successors[i].items():
17                if L[i]+length_ij<shortestDistance and j not in S:
18                    shortestDistance=L[i]+length_ij
19                    nextNode=j
20                    predesessor = i
21        L[nextNode] = shortestDistance
22        P[nextNode] = predesessor
23        if nextNode not in S:
24            S.append(nextNode)
25    return L,P
```

1.2 [5 marks] Show that your implementation indeed has complexity $O(n^3)$

Proceeding as we did in the lectures, analysing the complexity of each line/block of lines. This is all done assuming that writing/removing etc. takes $O(1)$,

- Lines 2 to 9 take $O(1)$.
- Each time the block of lines 10 to 24 is repeated:
  - Lines 13 and 14 are $O(1)$.
  - Each time it is repeated, line 15 initiates a new block, due to line 10, it is repeated $O(n)$ times. (the logic of this comes from the lectures as a node is added to S on each iteration, starting with 1 node and ending with n nodes)
    - Line 16 we say is repeated a n times as it is repeated for all in the list of explored nodes, which will always be less than or equal to n, closer, the more dense the graph is in terms of nodes.
    - All inside the block held by line 16 (17 to 20) is repeated enough times to find all the successors of the node its checking at the time, the number of $len(successors[i].items()) <= n$ this block of $O(1)$ lines takes on $O(b)$ for b tending to n as the graph becomes more dense.
  - Lines 21 to 23 are also $O(1)$
- Finally just returning the lists

Overall, we have a complexity of:

$$O(1) + O(n)\big(O(1) + O(m)(O(b))\big) = O(nmb) = O(n^3)$$

for dense graphs.

1.3 [5 marks] Pseudocode for the $O(n^2)$ version

Report the pseudocode(in this case the actual code) of your implementation "`dijkstra2`". This is the main thread of the method, however within this I used another method called `minimize()` which I will add below, I wrote this as separate method to improve the readability of the code, as having it inside I believe would clutter the method (and more importantly confuse me when trying to write it to the spec of the pseudo from the lectures).

```
def dijkstra2(successors, s):
    S = []
    S.append(s)
    n = len(successors)
    L = dict()
    L[s] = 0
    P = dict()
    P[s] = '-'
    unexploredList = []
    #WRITE ME
    for i in range (0, n):
        if (i!=s):
            if (i in successors[s].keys()):
                L[i]=successors[s][i]
            else:
                L[i]=numpy.Infinity
            unexploredList.append(i)
            P[i]=s
    while len(S)<n:
        key,value = minimize(unexploredList, L)
        S.append(key)
        if key in unexploredList:
            unexploredList.remove(key)
        for j in unexploredList:
            if j in successors[key].keys():
                length = L[key]+successors[key].get(j)
                if (length<L[j]):
                    L[j] = L[key]+successors[key].get(j)
                    P[j]= key
    return L,P
```

Next is the method `minimize()`, used to grab the smallest valency node in the list of explored nodes, so that I know which to use for adding new nodes to the set of explored ones

```
def minimize(unexplored, all):
    shortest = numpy.Infinity
    node = 0
    for key in unexplored:
        value = all.get(key)
        if value<shortest:
            shortest=value
            node = key
    return node,shortest
```

Just as a note, I would have split this into more methods as is my general coding style, but after writing the one and then realising how it would look in the report to be jumping around different methods in the efficiency analysis and so decided not to.

1.4 [5 marks] Show that your implementation indeed has complexity $O(n^2)$

Proceed as we did in the lectures, analysing the complexity of each line/block of lines.

- Like the previous, lines 2 to 9 are $O(1)$.
- The block of lines 11-to-18 is repeated n times as given by Line 9, the block is made up from $O(1)$ code lines so this takes a total of $O(n)$
- The next block of lines 19 to 24 encompassed by the while statement on line 19 is repeated n times as 1 node is added to the explored list S each iteration.
  - Inside the block first we have lines 20 to 23 of $O(1)$
  - Next is a block (24 to 29) that repeats m where on the first iteration of the lop on line 19 $m = n$, and for every iteration afterwards it is 1 less, this means it adds $O(m) < O(n)$ in total
    - The little block of (25 to 29) is $O(1)$

Overall, we have a complexity of:

$$O(1) + O(n)\big(O(1) + O(n) + O(m)\big) = O(n^2).$$

2 Experimental results and analysis

2.1 [5 marks] Empirical computing times

Report the empirical computing times you have computed in Excel for `dijkstra1` and `dijkstra2` (y_1 and y_2) in a table. Report in an Excel line chart the values of y_1 versus n and y_2 versus n.

Table 1: Table of the computing times on my latest run (this doesn't automatically update like the excel sheet, if you wish to run the program yourself and get new data check the README.txt
This table shows the computing times for graphs given the node count for `dijsktra1()` and `dijkstra2()`

Node count	Dijkstra 1	Dijkstra 2
8	7.22963E-05	9.36296E-05
16	0.000438914	0.000189235
16	0.000274963	0.00032079
24	0.001270123	0.000380049
24	0.000683852	0.000340938
32	0.002972049	0.00067437
32	0.001468049	0.000544
48	0.018410272	0.001216
48	0.006371951	0.001303704
64	0.018998914	0.002200099
72	0.044270222	0.004164741
96	0.073347951	0.00828642

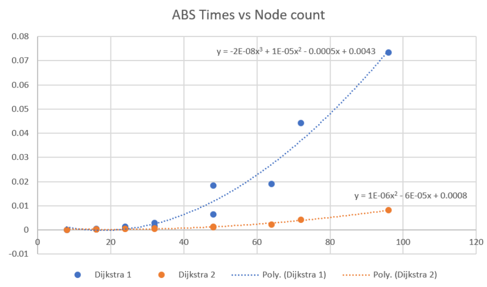

Figure 1: The line chart from excel that shows the nodes verses time taken to traverse the graph

2.2 [5 marks] Logarithms of the computing times

Report the logarithms of the empirical computing times you have computed in Excel for `dijkstra1` and `dijkstra2` ($\log(y_1)$ and $\log(y_2)$) in a table. Report in an Excel line chart the values of $\log(y_1)$ versus $\log(n)$ and $\log(y_2)$ versus $\log(n)$.

Table 2: Table of natural logs of the previous table

$ln(Nodes)$	$ln(D_1)$	$ln(D_2)$
2.079441542	-9.534737657	-9.276163669
2.772588722	-7.73120802	-8.572523212
2.772588722	-8.19887415	-8.04472347
3.17805383	-6.668641173	-7.875209359
3.17805383	-7.287769255	-7.983809119
3.465735903	-5.818503536	-7.301731087
3.465735903	-6.52382071	-7.516561311
3.871201011	-3.994846531	-6.712188495
3.871201011	-5.055849637	-6.642546062
4.158883083	-3.963373481	-6.119253026
4.276666119	-3.117443013	-5.481101252
4.564348191	-2.612540715	-4.793137279

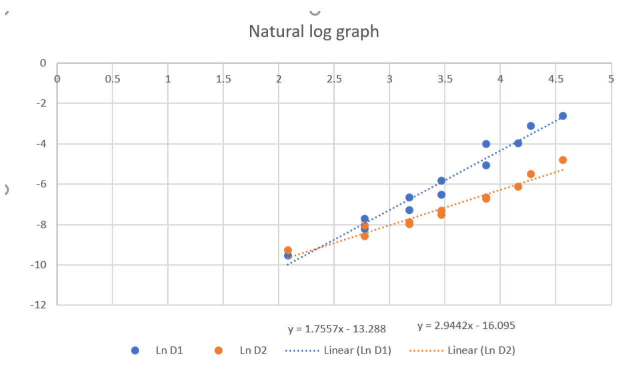

Figure 2: Figure showing the logs of times taken for the 2 dijkstra algorithms against the log of the number of nodes.

2.3 [5 marks] Estimated complexities

Report the computational complexities of `dijkstra1` and `dijkstra2` (y'_1 and y'_2) you have estimated in Excel via linear/affine regression as two functions of n.

As given by the coursework specification, for $i = 1{,}2$:

$$y_i = c_i n^{k_i} \tag{2.1}$$

And so this gives us y'_i as:

$$y'_i = ln(c_i) + k_i n' \tag{2.2}$$

Where $y'_i = ln(y_i)$ and $n'_i = ln(n_i)$ With the empirical evidence found in the excel sheet I have the following for y'_1 and y'_2

Table 3: Empirical values for $ln(c_i)$ and k_i.

i	k_i	$ln(c_i)$	c_i
1	2.944153824	-16.09540476	1.02295E-07
2	1.755667604	-13.28753332	1.6955E-06

2.4 [5 marks] Speedup

Report the estimated speedup `dijkstra2` w.r.t. `dijkstra1` as the function (of the instance size n) $\frac{y'_2}{y'_1}$. Compare this empirical speedup to the theoretical one (the latter is equal to the ratio of the computational complexities of your implementations, expressed in big-O notation. Is the empirical one smaller or larger than the theoretical one? Motivate the answer.

Empirical speedup:

$$\frac{y_2}{y_1} = \frac{(1.69E-06)n^{1.756}}{(1.02E-07)n^{2.944}} \tag{2.3}$$

Theoretical speedup:

$$\frac{O(n^2)}{O(n^3)} = O(n^{-1}). \tag{2.4}$$

What this shows is the empirical change and the predicted change being very close, with the empirical slightly larger going from what essentially shows a big-O of 2.94 going to 1.76, which is a Δ of just more than -1!

The information given by this experiment is plenty proof and honestly much closer to the prediction than i expected at a mere 0.2 difference of an empirical $\delta = 1.188$ to the predicted $\delta = 1$. It shows me that the $O(n)$ calculations hold pretty well against real life, at least in this case. The nature of how I compiled my excel analysis meant that I was able to rerun multiple times and see how results changed which also shows me its pretty consistently close to this too. I did this because it's not very scientific if its not repeatable. A small disclaimer with the empirical numbers I have shown is that this was one of the better runs in terms of showing the numbers I wanted to see, I would often get deltas as low as 0.7, but what I think that shows is in fact the difference is affected a lot by the background processes running at the time on the computer, and also that despite it still being a similar distance from the expected delta it is still pretty close.

With the final version of my code (the one submitted with this report and in the coursework file) shows a general tending of the change in O to be slightly more than 1, I think this may be due to my dijkstra 1 method being less efficient than it could be.

Something I feel like would have been useful to do to see in slightly more depth how the algorithms affected the efficiency would have been to see how the actual density (number of arcs) on the graph would affect in tandem with our node calculations, this however would take a considerable amount more time and probably some 3D graphs and other things I don't understand so I'm happy to leave it here.