We looked at Dijkstra's algorithm again, and we discussed which of the three problems from 20170912 can be solved with Dijkstra’s algorithm. I asserted (without proof) that Problems 1 and 3 are solvable, but Problem 2 is effectively impossible to solve ... by which I mean that it is almost certain there is no efficient algorithm that solves Problem 2.

We spent a few minutes talking about how to modify the original algorithm to solve Problem 1. I won’t give the definition of Problem 1 here – it is in the pdf of the powerpoint slides from 20170912.

So what needs to change? Well, the value of a path is no longer the sum of the weights of the edges: it is now the minimum of the weights of the edges in the path. When we add a new edge to a path, we need to determine if it reduces the capacity of the path. This is done by comparing the weight of the new edge to the current value of the path, and using the smaller of the two as the value of the expanded path. If this is larger than the current value of the vertex, we use it to update the best path info for the vertex.

Since we are maximizing instead of minimizing (ie, trying to find the path from A to B with the highest value rather than the lowest cost) we need to change the choice from smallest to largest.

Also, the initial values of “Cost” need to change ... Cost(A) needs to start at \(\infty\) and all other costs need to start at 0 (ideally we should change the name “Cost” to “Bandwidth” but I’m trying to keep the changes as small as possible).
Here is the revised algorithm, with the changes marked in red.

```plaintext
Cost(A) = \infty  \quad \# LaTeX won’t let me show \infty in red
Cost(v) = 0 \quad \text{for all } v \neq A  \quad \# \text{initially we have not found any paths from A to any other vertex}

# \text{Pred(x)} will be used to keep track the predecessor of x
Pred(v) = "" \quad \text{for all } v

# create a set Reached to contain the vertices for which we have already
# found the shortest paths. This makes sure we only process each vertex once.
Reached = {} \quad \# \text{initially, we have found the road to nowhere}
                    \# (Hey! Hey!)

# create a set Candidates to contain the vertices that are candidates for
# selection
Candidates = \{A\}  \quad \# \text{Candidates can be implemented in a}
                   \quad \# \text{variety of ways (heap, hash table, etc.)}

while Candidates != {} \quad \# \text{keep going as long as there are candidates}
    let x be the vertex in Candidates with \text{maximum Cost value}
    add x to Reached, and remove x from Candidates
    for each vertex y such that y is a neighbour of x
    \quad \text{and y is not in Reached:}
        \quad if \text{min(Cost(x), w(x,y))} > \text{Cost(y)}: \quad \# \text{we have now found a better}
        \quad \quad \# \text{path from A to y}
        \quad \quad if \text{Cost(y)} == 0: \quad \# \text{if we haven’t seen y before}
        \quad \quad \quad add y to Candidates
        \quad \quad Cost(y) = \text{min(Cost(x), w(x,y))} \quad \# \text{update our information about}
        \quad \quad \quad \# \text{the best path from A to y}
        \quad \quad \quad Pred(y) = x \quad \# \text{make note of the fact that on this path,}
        \quad \quad \quad \# \text{y’s predecessor is x}

return Cost, Pred
```
Note that none of the changes affect the complexity of the algorithm. This means we can solve Problem 1 just as efficiently as we can solve the problem of finding least-weight paths (which is what the original version of Dijkstra’s Algorithm achieves). This idea of establishing that different problems have the same level of difficulty – that is, take the same amount of time to solve – will be crucial to our upcoming discussion of problem classes.

As an exercise, you should work out the modifications to Dijkstra’s Algorithm to solve Problem 3 – the problem involving flights. Being able to adapt known algorithms to solve new problems is an important skill.

We moved on to a brief discussion of scheduling problems. As with the first set of problems, the goal was to introduce some easily defined problems and talk about whether they are easy or hard to solve.

Suppose we have a set of tasks $T = \{t_1, \ldots, t_n\}$ and for each task we know its duration: let $d_i$ be the time required to complete $t_i$

**Problem 4:** In what order should we complete the tasks so as to minimize the time to complete them all?

It’s pretty obvious that the answer to Problem 4 is ... all orders of completion result in exactly the same total time. It’s just the sum of the $d_i$ values. So the complexity of choosing the order is $O(1)$ – we don’t have to do any work at all, we just have to access the set of tasks in whatever order it is already in.
**Problem 5:** In what order should we complete the tasks so as to minimize the average of the set of times when we complete the tasks.

As an example, suppose we have just two tasks with \( d_1 = 4 \) and \( d_2 = 8 \). If we carry out \( t_2 \) first and \( t_1 \) second, the completion times are 8 and 12, with an average of 10. However if we carry out \( t_1 \) first and \( t_2 \) second, the completion times are 4 and 12, with an average of 8. It seems intuitive that we should carry out the tasks in order of duration, with the shortest tasks first. This is correct ... can you prove it?

So the complexity of the natural algorithm for Problem 5 is just the complexity of sorting the \( n \) tasks, which is \( O(n \log n) \)

**Problem 6:** Suppose we have two workers who can divide the tasks between them. How should we assign tasks to the workers so as to minimize the time at which all tasks are complete?

For example, suppose we have four tasks with \( d_1 = 7, d_2 = 8, d_3 = 4, d_4 = 2 \).

We could assign \( \text{worker}_1 \) to \( \{t_1, t_4\} \) with total time 9, and assign \( \text{worker}_2 \) to \( \{t_2, t_3\} \) with total time 12 ... so the time at which all the tasks are complete is 12.

Or, we could assign \( \text{worker}_1 \) to \( \{t_1, t_3\} \) and \( \text{worker}_2 \) to \( \{t_2, t_4\} \), which results in all tasks being completed at time 11.

I leave it to you to determine if there is any better way to divide these tasks between the two workers.

Clearly the ideal division would be to give each worker a subset of the set of tasks that would total exactly half of the total time requirement. If that isn’t possible, we want to get as close as we can to that ideal.
How many potential solutions are there to Problem 6? \( \text{worker}_1 \) can be assigned any subset of the set of tasks, so there are \( 2^n \) ways to assign tasks to \( \text{worker}_1 \) (with all the unassigned tasks going to \( \text{worker}_2 \)), but this overcounts the total number of solutions by a factor of 2, because \( \text{worker}_1 \) and \( \text{worker}_2 \) are effectively identical.

For example, with 5 tasks, assigning \( \text{worker}_1 \) to \( \{t_1, t_3, t_4\} \) and \( \text{worker}_2 \) to \( \{t_2, t_5\} \) is going to give exactly the same completion time as assigning \( \text{worker}_1 \) to \( \{t_2, t_5\} \) and \( \text{worker}_2 \) to \( \{t_1, t_3, t_4\} \).

So the total number of possible solutions to consider is \( \frac{2^n}{2} \), which is just \( 2^{n-1} \)... and for large values of \( n \), this is a very large number. Making a list of all possible solutions and then choosing the best one would take a long time.

The surprising thing is that this problem, like Problem 2, is effectively unsolvable – we don’t have, and we believe we never will have, an efficient algorithm for this problem. We will study a method to reduce the number of possible solutions to consider, but we will still have to deal with an exponential number of them.

Hopefully, this is strange and puzzling. What is it about Problem 2 and Problem 6 that makes them so hard to solve?

That is exactly what we are going to discuss for the next week or so.