Homework 5

Simon Zheng
260744353

December 3rd, 2017

1 Closest pair of points

The algorithm should mostly stay the same except at the part where we must search for a possible closer pair of points crossing the middle line.

When create the "boxes", we

2 Cell-hopping frog

We can make some assumptions from the statement "*At every step, the frog can jump from its current cell* (i, j) *to any cell* $(i\prime, j\prime)$ *that satisfies* $i\prime + j\prime = i + j + 1$":

The frog can only ever move up and right. When the frog is at $(1, 1)$, it can only move to $(1, 2)$ or $(2, 1)$. These two cells form a sort of diagonal perpendicular to $(1, 1)$.

It is the same for the next hop: The frog can only move to $(1, 3), (2, 2), (3, 1)$ which form a sort of diagonal perpendicular to the movement.

This is always the case because this is how we can satisfy the equation for allowed cell movements.

But the more interesting thing is that, when at any diagonal (or "level"), which represents the number of hops the frog has done, the next hop always has $i + j$ possible cells. E.g. $(1, 1)$ has $1 + 1 = 2$ hop options to $(1, 2), (2, 1)$, which has $1 + 2 = 2 + 1 = 3$ hop options to $(1, 3), (2, 2), (3, 1)$, etc.

Now, if the table isn't already created, create it (as a double array matrix or graph, it does not matter) in $O(n^2)$ in such a way that we can store a value inside.

Starting from $(1, 1)$, store a value of 1 in it. This is the base case.

Next, the algorithm stores the same value of 1 in the next "level" (cells in the next diagonal) $(1, 2), (2, 1)$.

Same for the next one $((1, 3), (2, 2), (3, 1))$, except that you add up all the values from the cells of the previous diagonal.

The idea is that every cell stores this computed value from their "parents" or previous cells (from the previous diagonal). Thus, we will "push" the added up values of every possible path to the destination and then we just get the final value from the destination cell.

If a cell is marked as forbidden, just put 0, and as such, we stop it from adding to future paths. If an entire diagonal is full of forbidden cells, then everything will add up to 0, indicating there are 0 possible paths.

To make things slightly more efficient, we can store 0 when the matrix or graph is created.

This still requires visiting every cell, but only once, until we reach the level with the destination in it, and we only need to add up values from the "parents"/previous diagonal cells, which would be visiting them a second time only. Thus, this is $O(2n^2) = O(n^2)$ (if we must create the cells it is just one other n^2 operation so it stays the same).

3 Largest subsequence

4 Job processing with weights and deadlines

5 Currency exchange

So we know that an α value between 0 and 1 is a disadvantageous exchange, and anything greater than 1 is advantageous.

We can use this fact and the fact that the logarithm of a number between 0 and 1 is negative to find an "advantageous cycle".

First, we convert the weight of every edge to its negative log, which is $O(n + m)$ where there are n nodes and m edges.

After this, we can use the algorithm seen in class to find a negative cycle, which is what we want. This is the reason we want the negative of the logarithm, as without this, disadvantageous α weights would be negative and the opposite for advantageous ones, yet the algorithm searches for negative cycles.

Algorithm to find the shortest path with negative weights:

Input: A node u in a graph G
Output: The nodes of the shortest path

```
$D[s] = 0$ and $D[v] = \infty$ where $v \neq s$

for ($i = 1$ to $n - 1$)
    forall ($v$)
        x = $min_{wv \in E} c_{wv} + D[w]$
        if ($x < D[v]$)
            $D[v] = x$
```

The running time is the same, and added with converting the weights to the negative logarithm, this gives us $O(n + m) + O(nm) = O(nm)$.

6 Number of shortest paths

We can use the same algorithm then for question 2 (frog hopping on cells):

Run BFS on the graph from s to t, so stop at t. At each node or vertex, add up the values stored in all of its "parents".

We initially put a value of 1 in s. A parent is defined as every neighbouring vertices of a vertex u which has already been visited, and as such its children are defined as its neighbours that weren't visited.

The operation of adding can be done at the same time as when we get all the neighbours in a standard BFS algorithm and is $O(1)$ for each.

Thus, the running time stays $O(n+m)$ where n, m are respectively the number of vertices and edges.