Comp 251 Assignment 5

Ian Benlolo 260744397
McGill University
December 4, 2017

1. Using this distance, called the Manhattan Distance, not much has to change in our algorithm. The Manhattan distance still reflects absolute distance between two points; if the manhattan distance is larger, so is the euclidean one. The divide part is the same (except we use the Manhattan Distance instead of Eucliean one obviously), problem arises in the conquer step. When we set up our boxes to try to isolate every point, we cannot take boxes of size $\delta/2by\delta/2$ anymore because if we were to do so there could be two points in the same cell. To isolate points, we would need to take a fraction of value smaller than 0.5, say $\frac{1}{4}$.
   This ensures there is at most one point within every cell. Now making the points δ apart, we get nine cells, so we must check "the next 8 neighbours" instead of 11 as in the algorithm in section 5.4 in the book.

2. We begin by setting the table as a graph, with nodes as a $(1,1)$ cell and edges as a path from cell to cell. So cell $(1,1)$ has an edge to $(1,2)$, $(2,1)$, cell $(1,2)$ has an edge to $(1,3),(2,2)$ and $(3,1)$, etc. If we had no cells missing, the number of paths in an $n \times n$ board would be $1 \times 2 \times 3 \times \cdots \times n \times n-1 \times \cdots \times 1$. But after building our graph, we can see that this can be done by simply multiplying the number of children every node has (since they're all connected), and this extends to a board with a missing piece. So our algorithm will start by keeping a counter (initialized as 1) of the number of paths and will run a DFS algorithm which will simply increment the counter at every step by making it equal to itself times the number of children the current node has. This would run in $O(m+n)$ as it is a simple adjustment to a DFS algorithm.

3.

4. We make Opt[i,t] be the max profit of jobs $1,\ldots,i$ at time t. We start by sorting the jobs by deadline. We let $P(i)$ signify the job that is before job i which is compatible with it (they do not overlap). At time t, job i may either satisfy $t + t_i <= d_i$ or not.
   In the first case, we will have the relation $p_i + Opt[P(i), t - t_i]$ since we add the profit of this job to the profit of the previous suitable job. In the latter case, we take $Opt[i-1,t]$. The recurrence relation simply takes the max of these two cases since they are the only two cases that we can have.
   This algorithm would run in $O(n \log n)$ since the loop would only runs once over the jobs but the sorting take nlogn.

5. This question like the negative cycle problem but we are simply looking for a cycle that $\Pi\alpha_{uu} > 1$. We start by creating a vertex w and connecting it to all in the graph. This does not create a cycle since we are not creating any vertices towards w.
   We let D[v, i] be the the max profit of a path from s to v that has at most i edges. D[v, n-1] is the profit made from s to v since a path has n-1 edges.
   We set the base case as D[v,0]=1 if v is the vertex s and $D[v,0] = -\inf$ otherwise.
   The recursion relation will be $D[v,i] = max\{\alpha_{wv} * D[w,i-1], D[v,i-1]\}$for any vertex w such that wv is an edge.

6. We run a normal BFS algorithm and initialize s=vertex, s as visited, val[s] and count[s] as 1. Count is the number of shortest paths from vertex to the node, and val is the shortest distance. We consider the following couple of cases.
   A node v is visited for the first first time so it has only one path from source till v through u, so shortest path up to v is 1 + shortest path to u, and number of ways to reach v via shortest path is same as

count[u] because say u has 5 ways to reach from source, then only these 5 ways can be extended up to v as v is encountered first time via u (assuming there aren't parallel edges).

```
val[v] = val[u]+1,
count[v] = count[u],
visited[v] = 1
```

But now if v had already been visited we have the following three cases;
if the current node has val which is distance up to v through some other path which is equal to val[u]+1, in other words have equal shortest distances for reaching v using current path through u and the other path to v, then the shortest distance to v remains same, but the number of paths increases by the number of paths of reaching u.
If the current path that reached v is smaller than the previous value of val[v], then val[v] stores the current path and count[v] also gets updated. The third case would be if the current path has greater distance than the previous path, in this case, we don't change anything.
This algorithm runs in O(m+n) since it is a BFS.