Master 1 MoSIG

Algorithmic Problem Solving

APP3 Report
Hole Drilling
Team:
SACAD

Very good report.
Some points to be improved
(see comments)

Members:
Andrey **SOSNIN**
Majdeddine **ALHAFEZ**
Antoine **COLOMBIER**
Eman **AL-SHAOUR**
Son Tung **DO**

Grenoble, 12 November, 2017

1 Discussion on the current algorithm

1.1 Not an α-approximation

Let us assume that we have N points to drill and they are arranged as shown in figure1 where the distance between two horizontal points equals $2N$ and the distance between two vertical points equals to 1.

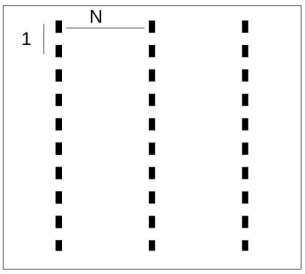

Figure 1: circuit board

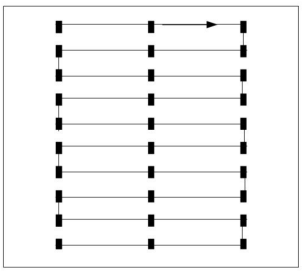

Figure 2: behavior of the current algorithm

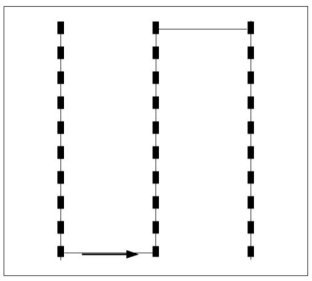

Figure 3: behavior of the greedy algorithm

Figure 2 shows the behavior of the current algorithm (this is the best behavior based on the given description where after drilling all the points in a row, we drill the point just bellow the last drilled point). The total cost of the path is $C = O(N^2)$

Figure 3 shows the behavior of what seems to be a better solution where we have $C' = O(N)$ The ratio is $O(N)$ and therefore such α doesn't exist and the current algorithm is not an α-approximation.

2 Greedy Approach

2.1 Propose an algorithm

This greedy approach consists of going through the set of points picking up the point with the minimum distance to the current point.

In order to compute the algorithm using the notation of point, we have a data structure for point

```
point {
        int x; /* x-coordinate */
        int y; /* y-coordinate */
}
```

Algorithm 1 Greedy Approach

Input: $A[N]$ array of N points to drill
Output: Array $A[N]$ with the order in which we drill the points

```
for k from 0 to N do
    index ← −1
    min_distance ← ∞
    for i from k+1 to N do
        distance ← find_distance(A[k], A[i])
        if distance < min_distance then
            index = i
            min_distance ← distance
        end if
    end for
    swap_elements(k + 1, index)
end for
```

The function $find\_distance(p_1, p_2)$ calculate the Euclidean distance between two points which is given by: $distance = \sqrt{(x_1 - x_2)^2 + (y_1 + y_2)^2}$

The complexity of this proposed algorithm is $O(N^2)$. The essential of this algorithm is similar to a sorting problem which can be implemented with complexity of $O(N \times log(N))$, but we still need compute the distance that costs $O(N^2)$

2.2 Showing it is not optimal

As we can see on figure4 bellow, the greedy solution is not optimal. Due to fact that the best choice in a local context might not be the best one for the final solution as we implement it in our greedy algorithm, we can then, admit that this solution is not solvable by a direct greedy solution.

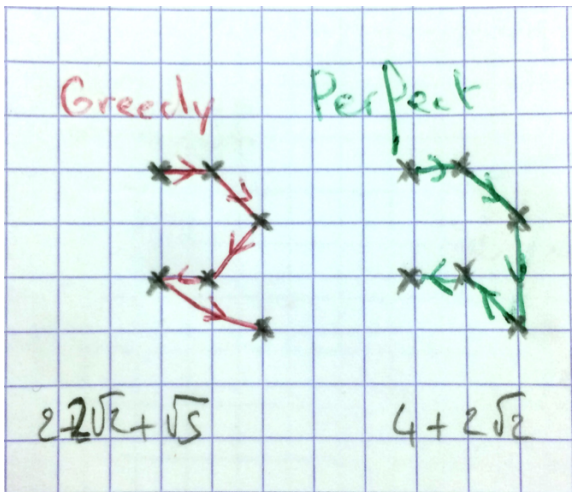

Figure 4: greedy vs optimal

3 Minimum Spanning Tree - MST

3.1 Propose an algorithm

In MST, we have nodes (the notation of point in our circuit board) and edges (represent the distance between 2 points in our circuit board).The idea of proposed algorithm based on the MST is to start with 1 node, we travel to the neighbors nodes. When we come to one neighbor node, we need explore its neighbors, except for the node we come from, until there is no neighbor in that node. At the node which have no neighbor but the previous node, we come back.

Algorithm 2 Algorithm based on MST

Input: MST for the point

Output: The *result_array* stores points in order that we need to move the drill

```
procedure EXPLORE(node, previous_node)   ▷ first call, we can have the previous_node is null
    result_array ← node                  ▷ add the current node to the result_array
    while exist a neighbor except the previous node do
        explore(oneneighbor, node)
    end while
    result_array ← node                  ▷ add the current node to the result_array again due to revisit
end procedure
```

3.2 Implementation

As output, Kruskal implementation is returning a Node containing all the neighbors sorted using the distance to itself. We have the implementation.

```
/* Data structure */
struct Node {
        int i;
        list<Node> neighbors
};

visited = create_array()
solution = create_array()

/* Entry point */
procedure drill_holes(holes)
        root_mst = kruskal_implementation(holes)

        for k from 0 to len(holes) - 1 do
                visited[k] = 0

        walkthrough_node(GetFirstNode(mst))

/* Recursive worker */
procedure walkthrough_node(node)
        index = node.i
        visited[index] = true

        solution += node
                n = len(node.neighbors)

        for i from 0 to n - 1 do
                if not visited[node.neighbors[i]]
                        walkthrough_node(node.neighbors[i])
```

Let n denote the number of nodes. For a complete graph, $|E| = O(n^2)$, So Kruskal has the complexity of $O(n^2 log(n))$. On the other hand, during the exploration, we check every node exactly once, so the complexity of this procedure is linear. Overall the algorithm is dominated by the Kruskal's cost.

3.3 2-approximation proof

We know that every spanning tree is more expensive than the weight of the MST, thus making a lower bound regarding the optimal solution : $MST \leq C^*$ (1)
Our algorithm consider the MST of the holes, in the worst case, we go through every edge in MST twice. $C = 2 \times MST$ (2)
From (1) and (2) we have $C \leq 2 \times C^*$.
We conclude that the algorithm is a 2-approximation.

4 Correctness of Kruskal's algorithm

We note by w the weight function.

We want to show that Kruskal's algorithm returns the minimal spanning tree.

First, the result is clearly a spanning tree since it doesn't have any cycles, and it is connected (as otherwise we would have encounter an edge that connects two disconnected parts without creating a cycle since the initial graph is connected). Now we are going to prove that it is minimal. Let K be the spanning tree produced by Kruskal's algorithm and T be an arbitrary spanning tree. We are going to show that by gradually transforming K into T by adding and removing edges, we can only increase its weight (therefore we will have $w(T) \geq w(K)$).

First, note that K and T have the same number of edges ($|V| - 1$) as they are both connected and acyclic. Suppose $T \neq K$. Let e_{min} be the edge with the minimum weight in $T \backslash K$.

Let $C = \{e_1, e_2, ..., e_k, e_{min}\}$ be the elementary cycle created by adding e_{min} to K. Let $e_j \in C \backslash T$ (e_j exists since the C is not contained in T, which is a tree). We have $w(e_j) \leq w(e_{min})$, since otherwise Kruskal would have chose e_{min} over e_j. Consider $K' = (K \backslash \{e_j\}) \cup \{e_{min}\}$. K' is a spanning tree, but its weight is greater or equal to the weight of K and $|K' \cap T| > |K \cap T|$.

Repeating the described procedure a finite number of times on K' in the place of K etc will produce T. Thus $w(T) \geq w(K)$.