Further Exploration with Kruskal's Algorithm

Basics of Kruskal Algorithm

Kruskal algorithm is another greedy algorithm used for finding the minimum spanning tree in a graph. It has variations regarding different implementations. But going down to the ultimately basic, it has the following one preset and two golden rules.

Preset:

> The edges in the given graph should be categorized into two groups: one for the visited, the other for the unvisited.

Rules:

1. Out of the unvisited group, pick (v-1) smallest edges and place them into the visited group; in the process of iteratively appointing the members of the visited group, the second rule needs to be abided by.
2. Each to-be appointed member of the visited group must not form up a cycle with any existing members from that group.

Pseudo Implementations of Kruskal Algorithm

With the basics in regard, variations of Kruskal algorithm could be derived.

In our research, we found variations for implementing Kruskal Algorithm. Below we will explain and analyze two variations on our radar. Variation 1 is the often seen one; variation 2 is one that we DIYed. Their specific pseudo implimentations could be illustrated as below:

Variation 1:

This variation could be seen a more standard one despite data structure representations in use for the graph and edges of the question.

1. We first enable struct "edge_list" type, a list of edges(with their index of vertices u, v; and their wight w). We will use this data structure type to facilitate our grouping of Visited and Unvisited edges. Particularly, "edge_list Visited" groups all edges that are found part of the to-be-finished MST; "edge_list Unvisited" groups all edges that are ready for choosing our next to-be member that is part of the the MST to its end.
2. Next, we need to sort all elements in "Unvisited" in ascending order. Ideally, we could see this sorting problem as a subproblem of the entire problem of finding the MST, and various sorting algorithms are available. Let's say that we use selection sort here.
3. After the edges in "Unvisited" are sorted in order, we implement another method/function to iterate edges in "Unvisited" to find (number of vertices in input array - 1) smallest edges that meet constraint of no cycle would form up out of some or all of them.

   Typically, we could use for loop to iterate the sorted "Unvisited" group; inside the for loop, we could implement an algorithm to find whether the iterated edge for review would form up a cycle with some or all of edges in the "Visited" group. Typically, there is a union-find algorithm that is a fit here.

   If the iterated edge were found to meet the constraint, the edge would be added to the "Visited" group.

   The for loop should stop once the number of elements in the "Visited" == (number of vertices in input array - 1).

Variation 2:

1. We first enable struct "edge_list" type, a list of edges(with their index of vertices u, v; and their wight w). We will use this data structure type to facilitate our grouping of Visited and Unvisited edges. Particularly, "edge_list

Visited" groups all edges that are found part of the to-be-finished MST; "edge_list Unvisited" groups all edges that are ready for choosing our next to-be member that is part of the the MST to its end.

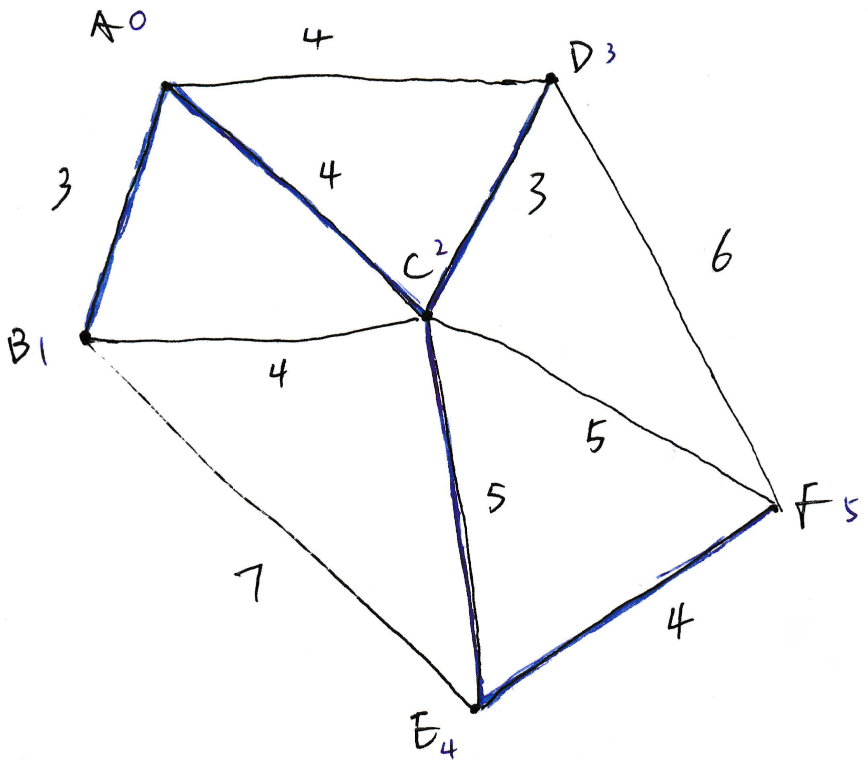
2. Next, we need to iterate all elements in "Unvisited" to find the next member to be added to "Visited". In our pseudo codes, we use a for loop to to find the smallest edge in "Visited"; then immediately in a nested for loop we check whether the counter would add up to 2 to determine whether this edge should be welcome to the "Visited" group. This for loop will break once a if clause found that the number of elements in "Visited" == (number of vertices in input array - 1).

   Once the for loop breaks, we return the MST, all edges eventually found in the "Visited" group.

You are welcome to note a piece of detailed pseudo code in Variation 2's regard appended in the end of this section.

Analysis of The Two Variations

So we are using Kruskal's algorithm. We will use the illustrated drawing graph below to instantiate our step-by-step explanation of the working process of the two variations



▼ Graph Representation illustration

Feel free to ponder whether this graph really set up in Euclidean 2D plane. The graph is for illustration only

If we apply variation 1 we get:

1. data structure representations preset;
2. Sort out the input adjacency matrix:

   Sorted:

   [0, 1] = 3, [2, 3] = 3, [0, 2] = 4, [0, 3] = 4, [1, 2] = 4, [4, 5] = 4, [2, 4] = 5, [2, 5] = 5, [3, 5] = 6, [1, 4] = 7.
3. Select edges for MST:

   [0, 1] = 3; no cycle formed with edges grouped to be part of MST;

   Let [0, 1] = 3 be part of MST

num edges in edge_list [Visited] == 1

[2, 3] = 3, no cycle formed with edges grouped to be part of MST;

Let [2, 3] = 3 be part of MST

num edges in edge_list [Visited] == 2

[0, 2] = 4, no cycle formed with edges grouped to be part of MST

Let [0, 2] = 4 be part of MST

num edges in edge_list [Visited] == 3

[0, 3] = 4, cycle formed with edges grouped to be part of MST;

Let [0, 3] = 4 not be part of MST

num edges in edge_list [Visited] == 3

[1, 2] = 4, cycle formed with edges grouped to be part of MST

Let [1, 2] = 4 not be part of MST

num edges in edge_list [Visited] == 3

[4, 5] = 4, no cycle formed with edges grouped to be part of MST

Let [4, 5] = 4 be part of MST

num edges in edge_list [Visited] == 4

[2, 4] = 5, no cycle formed with edges grouped to be part of MST

Let [2, 4] = 5 be part of MST

num edges in edge_list [Visited] == 5

For loop breaks when MST is formed because (number of vertices - 1) edges already be appointed to MST

If we apply Variation 2 we get:

1. data structure representations preset;
2. Iterate edges in Unvisited until a smallest edge from the set is determined to be part of MST, then erase the edge from the "Unvisited" and add it to the "Visited"; check the num edges in edge_list [Visited] ;

   iterate edges in "Unvisited" until a smallest edge from the set is determined to be part of MST, then erase the edge from the "Unvisited" and add it to the "Visited"; check num edges in edge_list [Visited]

   ...

   iterate edges in "Unvisited" until a smallest edge from the set is determined to be part of MST, then erase the edge from the "Unvisited" and add it to the "Visited"; check num edges in edge_list [Visited]; break outer for-loop because num edges in edge_list [Visited] == (number of vertices - 1) edges already be appointed to MST.

   Specifically,

   [0, 1] = 3 the smallest from the "Unvisited"; no cycle formed with edges grouped to be part of MST;

   Let [0, 1] = 3 be part of MST

   [0, 1]: added to "Visited", and removed from "Unvisited"

   num edges in edge_list [Visited] == 1

   [2, 3] = 3 the smallest from the "Unvisited"; no cycle formed with edges grouped to be part of MST;

   Let [2, 3] = 3 be part of MST

   [2, 3]: added to "Visited", and removed from "Unvisited"

   num edges in edge_list [Visited] == 2

[0, 2] = 4 ; the smallest from the "Unvisited"; no cycle formed with edges grouped to be part of MST;

Let [0, 2] = 4 be part of MST

[0, 2]: added to "Visited", and removed from "Unvisited"

num edges in edge_list [Visited] == 3

[0, 3] = 4 ; the smallest from the "Unvisited"; cycle formed with edges grouped to be part of MST;

Let [0, 3] = 4 be part of MST

[0, 3]: added to "Visited", and removed from "Unvisited"

num edges in edge_list [Visited] == 3

[1, 2] = 4 ; the smallest from the "Unvisited"; cycle formed with edges grouped to be part of MST;

Let [1, 2] = 4 be part of MST

[1, 2]: added to "Visited", and removed from "Unvisited"

num edges in edge_list [Visited] == 3

[4, 5] = 4 ; the smallest from the "Unvisited"; no cycle formed with edges grouped to be part of MST;

Let [4, 5] = 4 be part of MST

[4, 5]: added to "Visited", and removed from "Unvisited"

num edges in edge_list [Visited] == 4

[2, 4] = 5 ; the smallest from the "Unvisited"; no cycle formed with edges grouped to be part of MST;

Let [2, 4] = 5 be part of MST

[2, 4]: added to "Visited", and removed from "Unvisited"

num edges in edge_list [Visited] == 5

For loop breaks when MST is formed because (number of vertices - 1) edges already be appointed to MST

Depending on what specific algorithms programmers will use to solve certain subproblems of the entire problem of figuring out the MST of a given graph, variations of Kruskal or Prim algorithm could show different running efficiency and space use efficiency; the certain subproblems here refer to those that require a sub mechanism/sub algorithm that represents a part of the entire algorithm. In the two variations this section discusses, we found in average situations (not worst situations) both two variations present a O(v log v) where v = number of vertices of the input graph. However, we could not rule out the cases where programmers use very different sub algorithms and data structure representations so that their variations of either Kruskal's algorithm or Prim's might present worse time efficiency.

Pseudo codes for Variation 2:

```
#include <stdio.h>

typedef struct edge{
    int u, v, w;
}edge;

typedef struct edge_List{
    ...
}edge_List;

// Declare Visited: the determined part of the MST
// Initially set Visited empty
edge_List Visited;
Visited[][] = {}

// Declare Unvisited
// Set Unvisited such that it contains all edges given in the input 2d array, the input gr
aph represnetation;
edge_List Unvisited;
```

```
Unvisited = inputArray;

// Construct an Empty Set for Unchosen Edges; this part is not perfectly necessary for gen
erating MST.
Unchosen[][] = {  }
```

```
void MST(arr[row][col]){

    // Initially Set smallestE as first edge in input array
    smallestE = arr[0][0]

    // Sort the next smallest edge in Unvisited
    // this part is similar to selection sort
    for(i = 0; i < row; i++)
    {
        for(j = 0; j < col; j++){
            if smallestE > Unvisited[i][j]{
                smallestE = Unvisited[i][j]


            // Judge here if smallest Edg forms up a cycle with some edges
               in the Visited
            // In stead of using union-find algorithm, the algorithm here checks
               whether the index
            // of vertices equal to vertices of edges in Visited Set
               (determined part of the MST)

            counter = 0
            for i in edge_List Visited:
                if i[u] = edge smallestE[u]
                    counter += 1;
                if i[v] = edge smallestE[v]
                    counter += 1;
                if i[u] = edge smallestE[v]
                    counter += 1;
                if i[v] = edge smallestE[u]
                    counter += 1;

            // if a cycle forms:
            if counter == 2:
                Unchosen.append smallestE
                Unvisited.del smallestE

            // if they forms no cycle:
            else:
                Visited.append smallestE
                Unvisited.del smallestE

    // When (number of edges) in Unvisited Group equals to
```

```
       (number of vertices - 1): break the for loop.
    // It is possible that len(edge_list) is not a direct use for rendering
       the number of edges in the edge_list;
    // hence a for loop and a counter is an alternative way to look up the
       number of edges in the edge_list;
    if len(Unvisited) == row - 1:
        break;
    }
    return Visited;

}
```

```
int main(){
    // instance input:
    arr[6][6] = {{0, 3, 4, 4, 0, 0}, {3, 0, 4, 0, 7, 0}, ... }

    MST[arr[6][6]]

    return 0;
}
```