EE3980 Algorithms

Homework 8. Minimum-Cost Spanning Tree

106061151 劉安得

1. Introduction

- Greedy Algorithm
  - Feasible Solutions

    Arrange the inputs to satisfy some constraints
  - Optimal Solution

    Find feasible solution that minimize or maximize an objective function

The greedy method is a algorithm that takes one input at a time. If a particular input results in infeasible solution, then it is rejected; otherwise it is included. The input is selected according to some measure. The selection measure can be the objective functions or other functions that approximate the optimality. However, this method usually generates a suboptimal solution.

- Spanning Tree

  Let G = (V, E) be an undirected connected graph. A sub-graph T = (V, E') is a spanning tree of G if and only if T is a tree. Note that spanning tree is not unique, and spanning trees have n - 1 edges (n = |V |.)

- Minimum-Cost Spanning Tree

  There is a cost function associated with each edge. The cost of a tree is the sum of the costs of the tree edges. A feasible solution of the minimum-cost spanning tree of a undirected graph G is any

spanning tree T of G. The optimal solution is a spanning tree with the minimum cost.

There are two algorithms to solve this problem, Prim and Kruskal

- Graph

  A graph, G, consists of two sets V and E.

  - The set V is a finite, nonempty set of vertices.
  - The set E is a set of pairs of vertices; these pairs are called edges.
  - They are also denoted by V(G) and E(G).
  - And the graph is also denoted by G(V, E).

- Disjoint Sets

  - forest can be used to represent disjoint sets.
  - Operations important to set manipulations
    - Union: Merge two disjoint sets into one.
    - Find(i): Given an element i find the set that contains i.

- Min Heap

  A min heap is a complete binary tree with the property that the value at each node is at least as small as the values at its children (if they exist).

In this homework, we use nine graphs as input data, record the CPU times and correlate the performance to analyses.

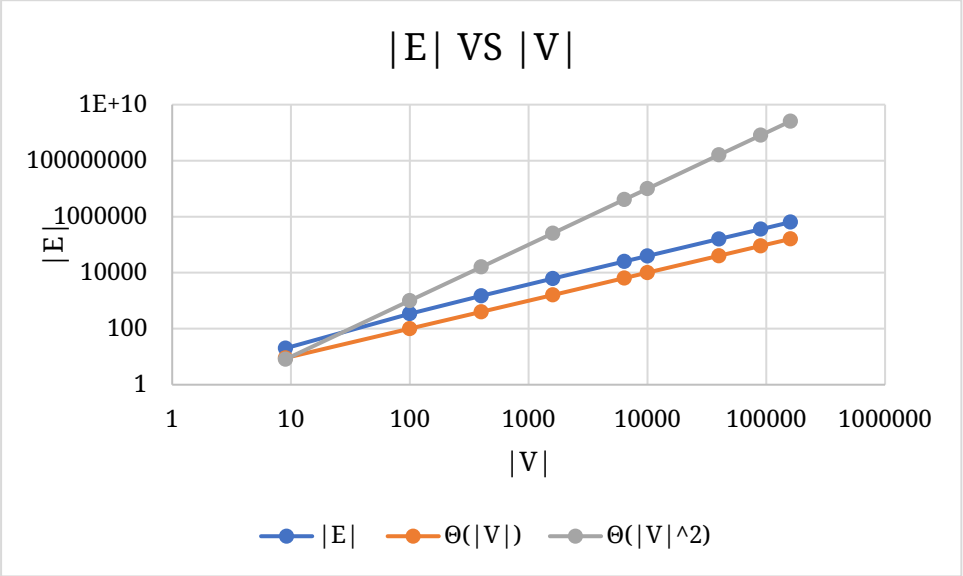
2. Approach

A. Keys of Implementation

- |E| = Θ(|V|)

  |V| is the number of vertices, |E| is the number of edges

  As the plot below shows, for the input data, |E| = Θ(|V|)



- Prim VS Kruskal

  Since |E| = Θ(|V|), the edge is sparse. Moreover, the time complexity of Prim is $O(|V|^2)$, and the time complexity of Kruskal is O(|E|*lg|E|) = O(|V|*lg|V|). Because the edge is sparse and the time complexity of Kruskal is smaller. I choose Kruskal algorithm to solve this problem.

- Find & Union

  By hw03, SetFind() and WeightedUnion() is the best algorithm for disjoint sets. Therefore, I choose them for my implementation.

- parent[]

  A single array, parent[], can represent the disjoint sets.

B. Kruskal

```
// Given a graph G(V, E) with cost function w find minimum cost spanning
tree.
// Input: V, E, n, w
// Output: minimum cost tree T and mincost.
Algorithm Kruskal(V, E, n, w, T)
{
    Construct a min heap from the edge costs using Heapity;
    for i := 1 to n do parent[i] := -1; // Enable cycle checking
    i := 0;
    mincost := 0;
    while ((i < n - 1) and ( heap not empty )) do {
        delete a minimum cost edge (u, v) from the heap;
        Adjust the heap;
        j := SetFind(u); // using parent array
        k := SetFind(v);
        if (j != k) then {
            i := i + 1;
            T[i, 1] := u;
            T[i, 2] := v;
            mincost := mincost + w[u, v];
            WeightedUnion(j, k); // modify parent array
        }
    }
    return mincost;
}

// Find the set that element i is in.
// Input: element i
// Output: root element of the set.
Algorithm SetFind(i)
{
    while (parent[i] >= 0) do i := parent[i];
    return i;
}

// Form union of two sets with roots, i and j, using the weighting rule.
// Input: roots of two sets i, j
// Output: none.
Algorithm WeightedUnion(i, j)
{
```

```
    temp := parent[i] + parent[j]; // Note that temp < 0.
    if (parent[i] > parent[j]) then { // i has fewer elements.
        parent[i] := j;
        parent[j] := temp;
    }
    else { // j has fewer elements.
        parent[j] := i;
        parent[i] := temp;
    }
}

// To enforce min heap property for n-element heap A with root i.
// Input: size n min heap array A, root i
// Output: updated A.
Algorithm Heapify(A, i, n)
{
    j := 2×i; // A[j] is the lchild.
    item := A[i];
    done := false;
    while j <= n and not done do { // A[j + 1] is the rchild.
        if j < n and A[j] > A[j + 1] then
            j := j + 1; // A[j] is the smaller child.
        if item < A[j] then // If smaller than children, done.
            done := true;
        else { // Otherwise, continue.
            A[j/2] := A[j];
            j := 2×j;
        }
    }
    A[j/2] := item;
}
```

I. Correctness

By [Horowitz] p. 244

Let G be any undirected connected graph. Let t be the spanning tree for G generated by Kruskal's algorithm. Let t' be a minimum-costs panning tree for G. We show that both t and t' have the same cost.

Let E(t) and E(t') respectively be the edges in t and t'. If n is the number of vertices in G, then both t and t' have n-1edges. If E(t) = E(t'), then t is clearly of minimum cost. If E{t) != E(t'), then let q be a minimum-cost edge such that q belongs to E(t) and does not belong to E(t'). Clearly, such a g must exist. The inclusion of q into t' creates a unique cycle. Let q, e1, e2, ..., ek be this unique cycle. At least one of the ei's, 1 <= i <= k, is not in E(t) as otherwise t would also contain the cycle q, e1, e2, ..., ek. Let ej be an edge on this cycle such that ej does not belong to E(t). If ej is of lower cost than q, then Kruskal's algorithm will consider ej before q and include ej into t. To see this, note that all edges in E(t) of cost less than the cost of q are also in E(t') and do not form a cycle with ej. So cost(ej) >= cost(q).

Now, reconsider the graph with edge set E(t') U {q}. Removal of any edge on the cycle , e1, e2, ..., ek will leave behind a tree t". In particular, if we delete the edge ej, then the resulting tree t" will have a cost no more than the cost of t'.

II. Time Complexity

Assume that element comparison dominates the CPU time

h is the height of the tree in which the element i is.

Statement	s/e	Freq.	Total steps
`Algorithm SetFind(i)`	0	-	0
`{`	0	-	0
`  while (parent[i] >= 0)`	1	h	h
`  do i := parent[i];`	0	h-1	0
`  return i;`	0	1	0
`}`	0	-	0
Total			h

$$t = h = \Theta(h)$$

Assume that element comparison dominates the CPU time

Statement	s/e	Freq.	Total steps
`Algorithm WeightedUnion(i, j)`	0	-	0
`{`	0	-	0
`temp := parent[i] + parent[j];`	0	1	0
`if (parent[i] > parent[j]) then {`	1	1	1
`parent[i] := j;`	0	1 / 0	0
`parent[j] := temp;`	0	1 / 0	0
`}`	0	-	0
`else {`	0	-	0
`parent[j] := i;`	0	0 / 1	0
`parent[i] := temp;`	0	0 / 1	0
`}`	0	-	0
`}`	0	-	0
Total			1

$t = 1 = \Theta(1)$

Assume that element comparison dominates the CPU time.

Statement	s/e
`Algorithm Heapify(A, i, n)`	0
`{`	0
`j := 2×i;`	0
`item := A[i];`	0
`done := false;`	0
`while j <= n and not done do {`	0
`if j < n and A[j] > A[j + 1] then`	1
`j := j + 1;`	0
`if item < A[j] then`	1
`done := true;`	0
`else {`	0
`A[j/2] := A[j];`	0
`j := 2×j;`	0
`}`	0
`}`	0

`A[j/2] := item;`	0
`}`	0
Total	

$$T_{Heapify}(i,n) \leq \lceil lg(n+1) \rceil - \lfloor lg(i) \rfloor \leq lg(n) + 1 = \text{O(lg(n))}$$

Assume that element comparison dominates the CPU time.

n is the number of vertices, e is the number of edges

Statement	s/e
`Algorithm Kruskal(V, E, n, w, T)`	0
`{`	0
`Construct a min heap from the edge costs using Heapity;`	$\Sigma T_{heap}(i, e)$
`for i := 1 to n do parent[i] := -1;`	0
`i := 0;`	0
`mincost := 0;`	0
`while ((i < n - 1) and ( heap not empty )) do {`	2
`delete a minimum cost edge (u, v) from the heap;`	0
`Adjust the heap;`	$T_{heap}(1, i-1)$
`j := SetFind(u);`	h
`k := SetFind(v);`	h
`if (j != k) then {`	1
`i := i + 1;`	0
`T[i, 1] := u;`	0
`T[i, 2] := v;`	0
`mincost := mincost + w[u, v];`	0
`WeightedUnion(j, k);`	1
`}`	0
`}`	0
`return mincost;`	0
`}`	0
Total	

Because of the weighted union function, by Lemma 2.3.6 on page 21 of the class handout,

lec23.pdf, the height is no greater than $\lfloor lg\ \text{n} \rfloor + 1$

$$T(n) \leq \sum_{\mathrm{i}=1}^{\mathrm{e}/2}(lg(\mathrm{e})+1)+\sum_{\mathrm{i}=1}^{\mathrm{e}-1}(lg(i)+2h+2)+(n-1)\times 1$$

$$\leq \sum_{j=1}^{\mathrm{e}/2}(lg(\mathrm{e})+1)+\sum_{j=1}^{\mathrm{e}-1}(lg(\mathrm{e})+2\lfloor lg\ \mathrm{n}\rfloor+3)+n-1$$

$$\cong \frac{3}{2}e\times lg(e)+e\times 2lg(n)+n$$

$$\cong \frac{7}{2}n\times lg(n) \text{ (since e = Θ(n))}$$

$$= \mathrm{O(n*lg(n))}$$

III. Space Complexity

- $S_{Find}(n)$

  n for parent[], one for each variable: *i*

  S = n+1 = Θ(n)

- $S_{Union}(n)$

  n for parent[], one for each variable: *i*, *j* and *temp*

  S = n+3 = Θ(n)

- $S_{Heapify}(n)$

  n for array *A,* one for each variable: *i*, *j*, *n*, *item* and *done*

  $S_{Heapify}(n)$ = n+5 = Θ(n)

- S(n, e)

  3e for E[] and w[], n for parent[], 2n-2 for T[], one for *i*, *j*, *k*, *u*, *v*, *mincost*

  S(n, e) = 3e + 3n-2 + 6 + $S_{Heapify}(n)$ = 3n + 3e + 4 + 5 = 3n + 3e + 9 = Θ(n+e) = Θ(n)

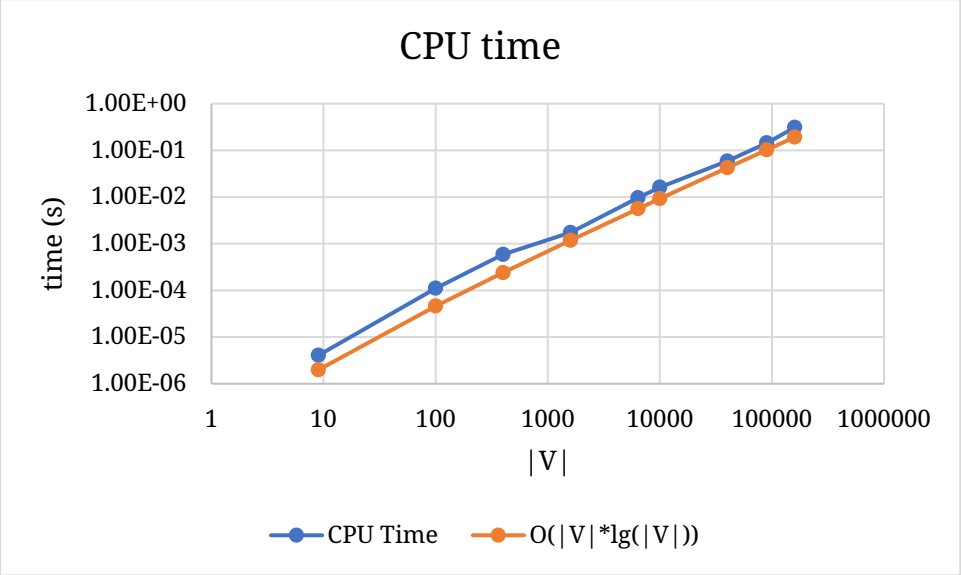
C. Main Function

```
// Driver function to solve Minimum-Cost Spanning Tree.
// Input: graph files
// Output: CPU time used, Minimum-cost spanning tree
Algorithm main(void)
{
    Read V & E and the edges of the graph;
    t0 := GetTime();
    Kruskal(G);
    t1 := GetTime();
    t := (t1 - t0);
    write(t, Minimum-cost spanning tree);
}
```

3. Result & Observations

A. CPU Time (Units: s)

|V|	|E|	CPU Time	Minimum cost
9	20	4.05312e-06	0.56
100	342	0.000110865	150.22
400	1482	0.000588894	2781.42
1600	6162	0.00173688	47766.82
6400	25122	0.00964594	791389.62
10000	39402	0.0161691	1945547.02
40000	158802	0.058846	31562194.02
90000	358202	0.143974	160519941.02
160000	637602	0.309583	508488788.02



B. Conclusions/ Observations

The result of analysis matches the plot, the time complexity is O(|V|*lg(|V|)),