CS-204: Design and Analysis of Algorithms

220001048, 220001049, 220001051

March 12, 2024

1 Prim's Algorithm

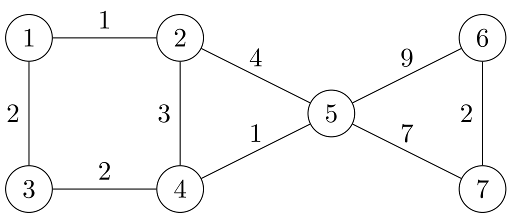
Given a weighted undirected graph, Prim's algorithm returns a minimal spanning tree. It is a greedy algorithm. The analysis of this algorithm is presented here.

1.1 Algorithm

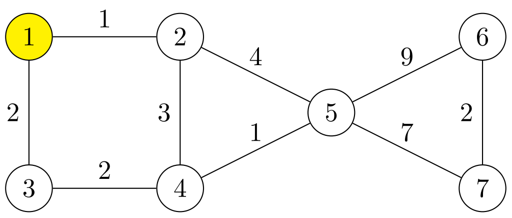
Algorithm 1 Prim's Algorithm

```
Input: Graph $G$ with vertices $V$, edges $E$, and weight function $w : E \to \mathbb{R}$
Output: Shortest path from a given vertex to all other vertices.
Procedure: PrimMST$(G, w)$
Let $n = |V|$
for $i = 1$ to $n$ do
    $visited[i] = 0$
    $d[i] = \infty$
    $N[i] = \text{NIL}$
end for
$TV = \{1\}$   $TE = \phi$
$visited[1] = 1$
for all $(1, u) \in G.E$ do
    $d[u] = w[1, u]$
    $N[u] = 1$
end for
for $u = 2$ to $n$ do
    Get $u \in V$ such that $d[u]$ is minimum and $visited[u] = 0$
    $TV = TV \cup \{u\}$
    $TE = TE \cup (u, N(u))$
    for all $(u, v) \in G.E$ such that $visited[v] = 0$ do
        if $d[v] > w[u, v]$ then
            $d[v] = w[u, v]$
            $N[v] = u$
        end if
    end for
end for
```

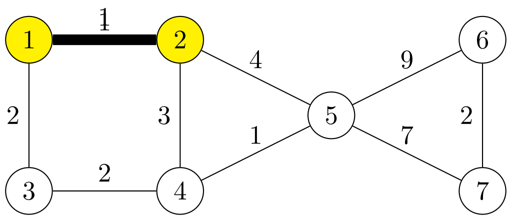
1.2 Example



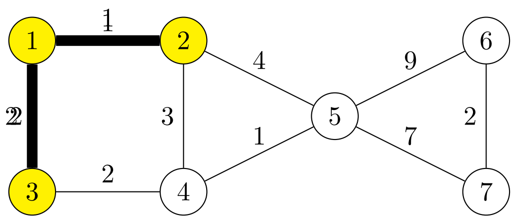
visited	0	0	0	0	0	0	0
distance	∞	∞	∞	∞	∞	∞	∞
neighbour	NIL	NIL	NIL	NIL	NIL	NIL	NIL



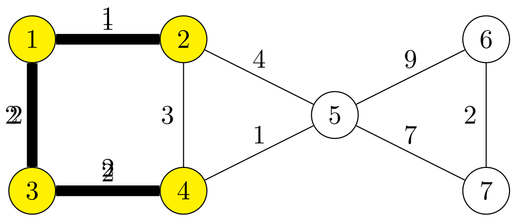
visited	1	0	0	0	0	0	0
distance	0	1	2	∞	∞	∞	∞
neighbour	NIL	1	1	NIL	NIL	NIL	NIL



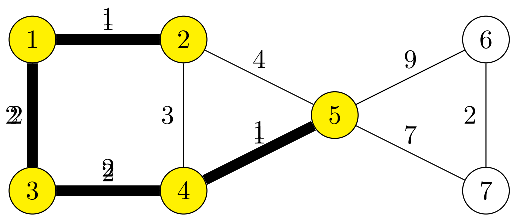
visited	1	1	0	0	0	0	0
distance	0	1	2	3	4	∞	∞
neighbour	NIL	1	1	2	2	NIL	NIL



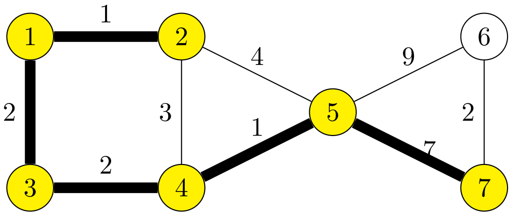
visited	1	1	1	0	0	0	0
distance	0	1	2	2	4	∞	∞
neighbour	NIL	1	1	3	2	NIL	NIL



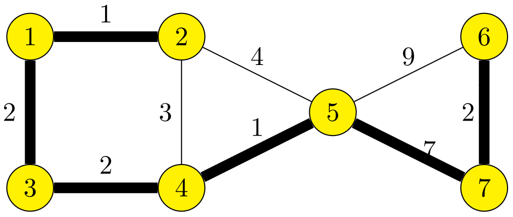
visited	1	1	1	1	0	0	0
distance	0	1	2	2	1	∞	∞
neighbour	NIL	1	1	3	4	NIL	NIL



visited	1	1	1	1	1	0	0
distance	0	1	2	2	1	9	7
neighbour	NIL	1	1	3	4	5	5



visited	1	1	1	1	1	0	1
distance	0	1	2	2	1	2	7
neighbour	NIL	1	1	3	4	7	5



visited	1	1	1	1	1	1	1
distance	0	1	2	2	1	2	7
neighbour	NIL	1	1	3	4	7	5

1.3 Time and Space Complexity Analysis

1.3.1 Time analysis

The key operations in this algorithm are as follows

1. We maintain three arrays:
   - *visited*: stores the vertices currently in the spanning tree.
   - d: stores the minimum edge weight from the current spanning tree to each vertex.
   - N: stores the other vertex of the minimum weight edge.

   Initializing these arrays takes $O(V)$ time.

2. Mark vertex 1 as visited and update the d and N arrays. This requires traversing all vertices adjacent to vertex 1. (decrease-key operation)

3. Extract the least weight vertex V-1 times (extract-min operation)

4. Iterate through all vertices adjacent to the chosen vertex and update the minimum edge weight connecting current spanning vertex to the vertex (decrease-key operation).

The decrease key operation in a binary heap takes $O(\log V)$ time. After decreasing the value in a priority queue, we swap the element with it's parent element until it satisfies the heap property. At max, the number of swaps will be height of the heap which is $O(\log V)$

Data Structure	**Decrease-Key**	**Extract-Min**
Array	$O(1)$	$O(V)$
Binary Heap	$O(\log V)$	$O(\log V)$

The second and fourth steps can be combined as checking all edges of the graph and applying the decrease key operation whenever applicable. In the worst case, it will be applied every time. So the time required for these two steps will be time taken check all the edges in the representation multiplied by time taken for decrease-key operation.

Adjacency list:
The first step takes $O(V)$ time. Checking all the edges of the graph takes O(E) time in adjacency list representation. So second and fourth steps together take $E \cdot O(\text{decrease-key})$ time. The third step takes $V \cdot O(\text{extract-min})$ time.
So the time complexity is:

$$\begin{aligned} &O(V) + E \cdot O(\text{decrease-key}) + V \cdot O(\text{extract-min}) \\ &= E \cdot O(\text{decrease-key}) + V \cdot O(\text{extract-min}) \end{aligned}$$

We can use a Binary heap to obtain the time complexity $O(V \log V + E \log V) = O((V+E)\log V)$.
We can use a Array to obtain the time complexity $O(V^2 + E) = O(V^2)$.
The optimal time complexity when using adjacency list representation is $O((V+E)\log V)$ which can be obtained by using a binary heap.

Adjacency matrix:
The first step takes $O(V)$ time. Checking all the edges of the graph takes $O(V^2)$ time in adjacency matrix representation. So second and fourth steps together take $V^2 \cdot O(\text{decrease-key})$ time. The third step takes $V \cdot O(\text{extract-min})$ time. So the time complexity is:

$$\begin{aligned} &O(V) + V^2 \cdot O(\text{decrease-key}) + V \cdot O(\text{extract-min}) \\ &= V^2 \cdot O(\text{decrease-key}) + V \cdot O(\text{extract-min}) \end{aligned}$$

We can use a Binary heap to obtain the time complexity $O(V \log V + V^2 \log V) = O(V^2 \log V)$.
We can use a Array to obtain the time complexity $O(V^2 + V^2) = O(V^2)$.
The optimal time complexity when using adjacency matrix representation is $O(V^2)$ which can be obtained by using an array.

1.3.2 Space analysis

The extra space used is as follows

1. O(V) space to maintain the *visited* and N arrays.

2. O(V) space to maintain d for all the mentioned data structures
3. O(V) space to store the edges and vertices of the minimal spanning tree.

So the overall space complexity is $O(V)$

1.4 Notes

When proving the correctness of the algorithm using *minimum separator lemma* , the edge weights were assumed to be distinct. If the edge weights are not distinct, we can have multiple minimum spanning trees.

2 Kruskal Algorithm

Algorithm 2 Kruskal's Algorithm

```
Input: Graph $G$ with vertices $V$, edges $E$, and weight function $w : E \to \mathbb{Z}$
Output: Shortest path from a given vertex to all other vertices.
Procedure: KruskalMST($G, w$)
Sort edges $E$ based on edge weight
Let $n = |V|$
for $i = 1$ to $n$ do
    $c[i] = i$
end for
$TE = \phi$
while $|TE| < (n - 1)$ do
    Let $(u, v) = E.next$ // gets the next unprocessed edge
    if $c[u] \neq c[v]$ then
        $TE = TE \cup \{(u, v)\}$
        for $j = 1$ to $|V|$ do
            if $C[j] = C[u]$ then
                $C[j] = C[v]$
            end if
        end for
    end if
end while
```

2.1 Time and Space analysis

2.1.1 Time complexity analysis

1. Sort the edges based on their weights. This operation takes $O(E \log E)$ time.

$$E \log E \leq E \log V^2 = 2E \log V$$

   So time taken to sort edges = O(E log V)

2. Maintain an array c (component array) to keep track of the component each vertex is in. Initializing this array takes $O(V)$ time (make operation).
3. Iterate through the edges and check if the vertices belong to the same component (find operation). This operation is performed for all edges in the worst case.
4. When adding a vertex to the minimum spanning tree , combine the components (union operation). This operation is performed V times.

In this algorithm, we directly work on the list of edges, so there is no need to create an adjacency list or matrix representation. When find and union operations are implemented naively as in the algorithm, the time complexity of the find operation is $O(1)$ and the time complexity of the union operation is $O(V)$ in the worst case. So the overall time complexity is:

$$\begin{aligned} &O(E\log V) + O(V) + E\cdot O(\text{find}) + V\cdot O(\text{union}) \\ &= O(E\log V) + O(V) + O(E) + O(V^2) \\ &= O(E\log V + V^2) \end{aligned}$$

2.1.2 Space analysis

The extra space used is as follows

1. O(V) space to maintain the c array.
2. O(V) space to store the edges and vertices of the minimal spanning tree.

So the overall space complexity is $O(V)$

2.2 Making it better - Union Find data structure

The union-find data structure is characterized by three operations:

- **Make:** Creates a new disjoint component.
- **Union:** Takes the union of two disjoint components.
- **Find:** Returns the component in which a given element is present.

These operations are used in Kruskal's algorithm. If two vertices are in the same component, we cannot add the edge between them as it leads to a cycle.

2.2.1 Example

- Initial state:

$$\begin{aligned} &\text{Component 1:}\{1,2,3\} \\ &\text{Component 2:}\{4,5,6,7\} \\ &\text{Component 3:}\{8,9\} \end{aligned}$$

- find(2) returns 1.
- Make 10:

$$\begin{aligned}&\text{Component 1:}\{1,2,3\}\\&\text{Component 2:}\{4,5,6,7\}\\&\text{Component 3:}\{8,9\}\\&\text{Component 4:}\{10\}\end{aligned}$$

- union(1, 3)

$$\begin{aligned}&\text{Component 1:}\{1,2,3,8,9\}\\&\text{Component 2:}\{4,5,6,7\}\\&\text{Component 4:}\{10\}\end{aligned}$$

- find(9) returns 1

2.2.2 Time complexity analysis

- The **make** function takes $O(1)$ time.
- The **find** function takes $O(1)$ time if we maintain the component of each element.
- The **union** function takes amortized $O(\log n)$ time. This is achieved with the help of small-to-large merging. When taking a union, the smaller set is always added to the larger set. We also need to maintain a size array.

Using this data structure, time complexity of Kruskal's algorithm is:

$$\begin{aligned}&O(E\log E)+O(V)+E\cdot O(\text{find})+V\cdot O(\text{union})\\&=O(E\log E)+O(V)+O(E)+O(V\log V)\\&=O(E\log E+V\log V)\\&=O(E\log V^2+V\log V)\\&=O((2E+V)\log V)\\&=O((E+V)\log V)\end{aligned}$$

2.3 Notes

Amortized analysis is a technique used to analyze the average-case time complexity of an algorithm. It is employed when a sequence of operations is performed, and some operations may be more expensive than others.

For example, merging sets of size 1 and $n-1$ is much cheaper than merging sets of size $n/2$ and $n/2$. We cannot assign the same weightage to them as given in worst-case analysis.