Kathmandu University

Department of Computer Science and Engineering

Dhulikhel, Kavre

COMP 314

Algorithms and Complexity

Lab Report 2

Submitted By:

Aayush Man Shakya(48)

Submitted To:

Dr. Prakash Poudyal

Department of Computer Science and Engineering

Date of Submission:

Jan 30, 2024

1. Activity Selection Problem

The activity selection problem is an optimization problem concerning the selection of non-conflicting activities to perform within a given time frame, given a set of activities each marked by a start time and finish time.

The greedy choice is to always pick the next activity whose finish time is the least among the remaining activities and the start time is more than or equal to the finish time of the previously selected activity. We can sort the activities according to their finishing time so that we always consider the next activity as the minimum finishing time activity.

Time Complexity: O(n)

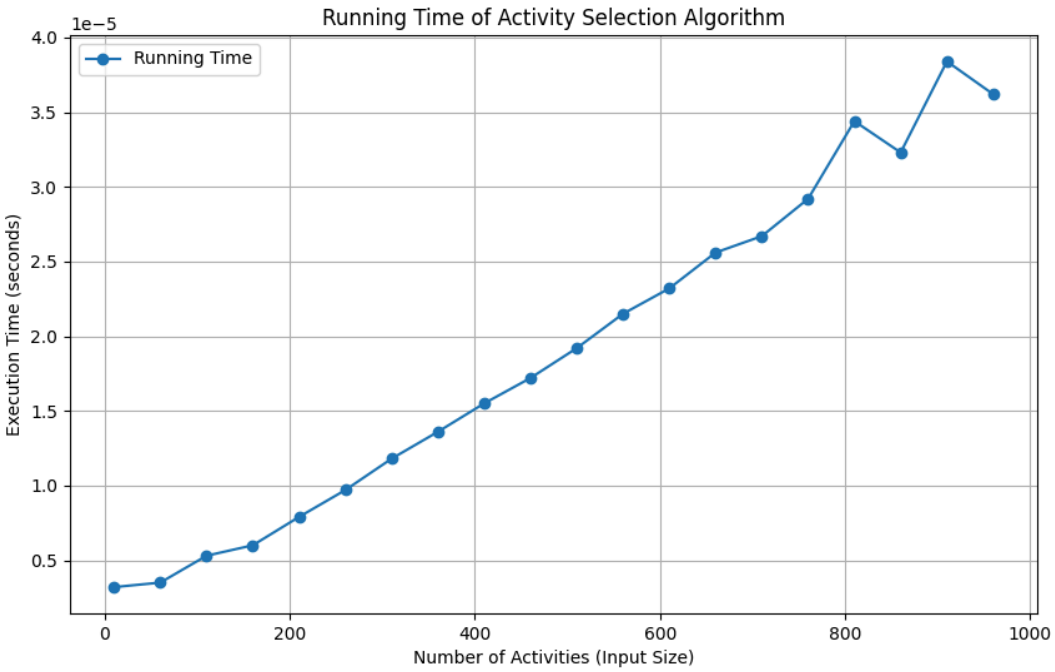
```python
def printMaxActivities(s, f):
    n = len(f)
    i = 0  # Index of the first activity

    selected_activities = [i]

    # Consider the rest of the activities
    for j in range(1, n):
        if s[j] >= f[i]:
            selected_activities.append(j)
            i = j

    return selected_activities
```

Graph:



2. Minimum Spanning Trees

a. Kruskal's Algorithm

In Kruskal's algorithm, we sort all edges of the given graph in increasing order. Then it keeps on adding new edges and nodes in the MST if the newly added edge does not form a cycle. It picks the minimum weighted edge at first and the maximum weighted edge at last. Thus, we can say that it makes a locally optimal choice in each step in order to find the optimal solution. Hence, this is a Greedy Algorithm.

Time Complexity: O(E * logV)

Code:

```python
class UnionFind:
    def __init__(self, size):
        self.parent = list(range(size))
        self.rank = [0] * size

    def find(self, x):
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])
        return self.parent[x]

    def union(self, x, y):
        xroot = self.find(x)
        yroot = self.find(y)
        if xroot == yroot:
            return False
        if self.rank[xroot] < self.rank[yroot]:
            self.parent[xroot] = yroot
        else:
            self.parent[yroot] = xroot
            if self.rank[xroot] == self.rank[yroot]:
                self.rank[xroot] += 1
        return True

def kruskal(vertices, edges):
    edges_sorted = sorted(edges, key=lambda x: x[2])
    uf = UnionFind(vertices)
    mst = []
    for u, v, w in edges_sorted:
        if uf.find(u) != uf.find(v):
            uf.union(u, v)
            mst.append((u, v, w))
    return mst
```

b. Prim's Algorithm

Like Kruskal's algorithm, Prim's algorithm is also a Greedy algorithm. This algorithm always starts with a single node and moves through several adjacent nodes, in order to explore all of the connected edges along the way.

The algorithm starts with an empty spanning tree. The idea is to maintain two sets of vertices. The first set contains the vertices already included in the MST, and the other set contains the vertices not yet included. At every step, it considers all the edges that connect the two sets and picks the minimum weight edge from these edges. After picking the edge, it moves the other endpoint of the edge to the set containing MST.

Time Complexity:

Worst Case: O((V + E) log V), when the graph is densely connected.

Best Case: O(E log V), when the graph is already a minimum spanning tree.

Code:

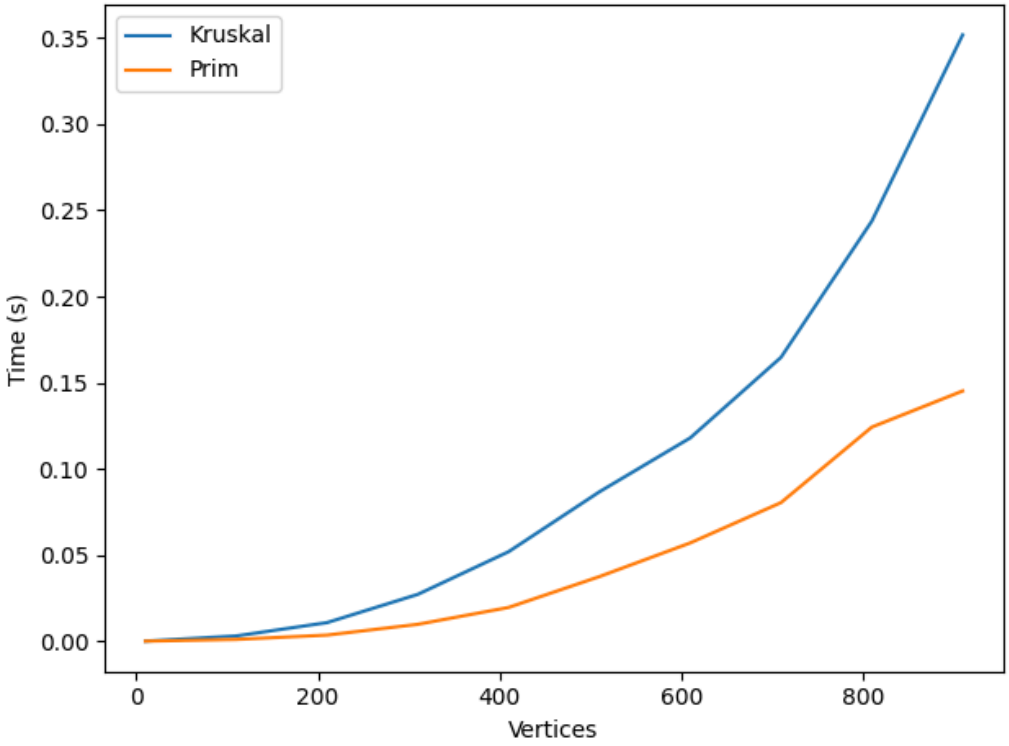
```
import heapq

def prim(adj_list, num_vertices):
    key = [float('inf')] * num_vertices
    parent = [None] * num_vertices
    key[0] = 0
    heap = [(0, 0)]
    visited = [False] * num_vertices

    while heap:
        current_key, u = heapq.heappop(heap)
        if visited[u]:
            continue
        visited[u] = True
        for v, w in adj_list[u]:
            if not visited[v] and w < key[v]:
                key[v] = w
```

```
                parent[v] = u
                heapq.heappush(heap, (w, v))
    return parent
```

Comparison Graph:



We can see from the graphs that Prim's algorithm performs better on randomly generated graphs. But, Prim's algorithm is typically preferred for dense graphs, leveraging its efficient priority queue-based approach, while Kruskal's algorithm excels in handling sparse graphs with its edge-sorting and union-find techniques.

3. N-Queens

The N-Queens problem consists in finding a position for N chess queens in a board of NxN squares where none of the queens are attacking each other.

a. Brute Force Approach

The brute-force approach works by generating all permutations of queen placements (ensuring no column conflicts) and checking each configuration for diagonal attacks.

```
import itertools

def is_valid(perm):
    for i in range(len(perm)):
        for j in range(i+1, len(perm)):
            if abs(i - j) == abs(perm[i] - perm[j]):
                return False
    return True

def n_queens_bruteforce(n):
    count = 0
    for perm in itertools.permutations(range(n)):
        if is_valid(perm):
            count +=
    return count
```

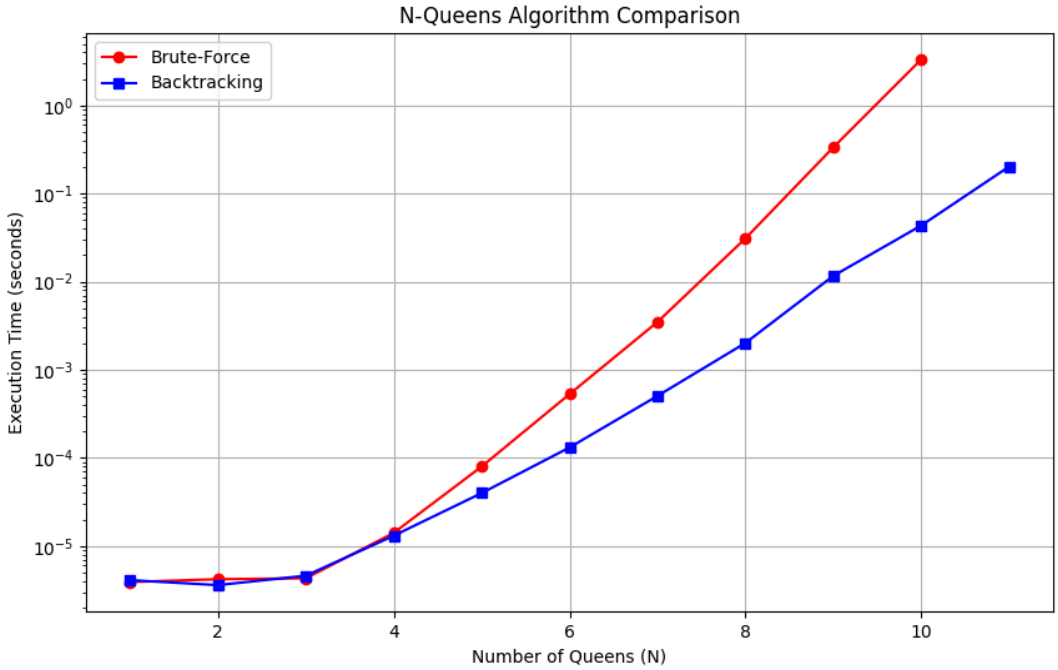
b. Backtracking Approach

In the backtracking approach, if a conflict is found, it skips that position and tries the next one. When all n queens are safely placed, it counts it as a valid solution. It's much faster than brute-force because it avoids checking impossible paths early.

```
def n_queens_backtracking(n):
    def backtrack(row, cols, diag1, diag2):
        if row == n:
            return 1
        count = 0
        for col in range(n):
            d1 = row - col
```

```
            d2 = row + col
            if col in cols or d1 in diag1 or d2 in diag2:
                continue
            cols.add(col)
            diag1.add(d1)
            diag2.add(d2)
            count += backtrack(row+1, cols, diag1, diag2)
            cols.remove(col)
            diag1.remove(d1)
            diag2.remove(d2)
        return count
    return backtrack(0, set(), set(), set())
```

Comparison Graph:



From the above graph, we can see that both bruteforce and backtracking perform similarly on smaller board sizes. But, when the board size starts to increase, we

can see that the bruteforce approach performs slower and slower. For huge board sizes, bruteforce is impractical and backtracking is preferred.

Conclusion

Hence, greedy algorithms such as Activity Selection, Prim's, Kruskal's were implemented and graphed. And N-Queens problem was also implemented using both bruteforce and backtracking approach. Algorithms were compared with each other to find which one performs better.