Kathmandu University

Department of Computer Science and Engineering

Dhulikhel, Kavre

Lab 2

COMP 314

(CE-III/II)

Submitted by

Puspa Hari Ghimire (19)

Submitted to

Dr. Prakash Poudyal

Department of Computer Science and Engineering

Submission Date:

January 27, 2025

Lab 2: Implementation of Greedy, Brute force and Backtracking Algorithms

Greedy methods:

This method builds up a solution piece by piece, always choosing the next piece that looks best at the moment. The main idea is to make locally optimal choice in the hope that this choice will lead to a globally optimal solution Greedy algorithms do not always yield optimal solutions, but for many problems they do.

Activity-Selection Problem:

It is the problem of scheduling several competing activities that require exclusive use of a common resource, with a goal of selecting a maximum-size set of mutually compatible activities.

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.

Recursive greedy algorithm:

Input: start times s, finish times f, the index k that defines the subproblem S_k it is to solve, and the size n of the original problem

```
RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n)
1  m = k + 1
2  while m ≤ n and s[m] < f[k]        // find the first activity in S_k to finish
3       m = m + 1
4  if m ≤ n
5       return {a_m} ∪ RECURSIVE-ACTIVITY-SELECTOR(s, f, m, n)
6  else return ∅
```

Iterative greedy algorithm:

```
GREEDY-ACTIVITY-SELECTOR(s, f)
1  n = s.length
2  A = {a_1}
3  k = 1
4  for m = 2 to n
5       if s[m] ≥ f[k]
6            A = A ∪ {a_m}
7            k = m
8  return A
```

Output for Recursive Greedy Activity Selection:

```
PS C:\Users\Lenovo\Documents\COMP 314 Algo\Algorithms Lab\Lab-2> python -u "c:\Users

Selected activities using recursive greedy approach are:
Activity 1: Start = 0, Finish = 4
Activity 2: Start = 4, Finish = 11
Activity 6: Start = 13, Finish = 17
Activity 8: Start = 18, Finish = 23
Activity 13: Start = 24, Finish = 28
Activity 17: Start = 30, Finish = 33
Activity 25: Start = 36, Finish = 38
Activity 32: Start = 38, Finish = 41
Activity 48: Start = 45, Finish = 46
Activity 62: Start = 48, Finish = 50
Activity 70: Start = 50, Finish = 52
PS C:\Users\Lenovo\Documents\COMP 314 Algo\Algorithms Lab\Lab-2>
```

Analysis:

In the recursive greedy approach for activity selection we have created n = 100 activities. All activities have random start time from 0 to 50 and finish times are greater than start time, up to 60. Here 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. Here, activity 25's successor is activity 32 because the finish time of activity 25 is 38 which is also the start time of activity 32. Hence, this demonstrates the greedy choice.

Output for Iterative Greedy Activity Selection:

```
PS C:\Users\Lenovo\Documents\COMP 314 Algo\Algorithms Lab\Lab-2> python -u "c:\User
Selected activities using iterative greedy approach are:
Activity 1: Start = 4, Finish = 5
Activity 5: Start = 8, Finish = 17
Activity 9: Start = 21, Finish = 23
Activity 14: Start = 24, Finish = 27
Activity 18: Start = 29, Finish = 30
Activity 22: Start = 31, Finish = 33
Activity 35: Start = 39, Finish = 40
Activity 43: Start = 40, Finish = 45
Activity 47: Start = 46, Finish = 47
Activity 50: Start = 47, Finish = 49
Activity 76: Start = 49, Finish = 55
PS C:\Users\Lenovo\Documents\COMP 314 Algo\Algorithms Lab\Lab-2>
```

Analysis:

Here, the selection process is same as above and only the implementation is iterative.

Minimum Spanning Tree:

A spanning tree of a connected graph G is a tree that consists solely of edges in G and that includes all of the vertices in G. Our solution to generate a spanning tree must satisfy the following constraints:

- We must use only edges within the graph.
- We must use exactly n-1 edges.
- We may not use edges that would produce a cycle.

A minimum spanning tree of a weighted graph is a spanning tree of least weight, i.e. a spanning tree in which the total weight of the edges is guaranteed to be the minimum of all possible trees in the graph.

Kruskal's algorithm:

Finds a safe edge by finding, of all the edges that connect any two trees in the forest, an edge of least weight.

```
MST-KRUSKAL(G, w)
1  A = ∅
2  for each vertex v ∈ G.V
3       MAKE-SET(v)
4  sort the edges of G.E into nondecreasing order by weight w
5  for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
6       if FIND-SET(u) ≠ FIND-SET(v)
7            A = A ∪ {(u, v)}
8            UNION(u, v)
9  return A
```

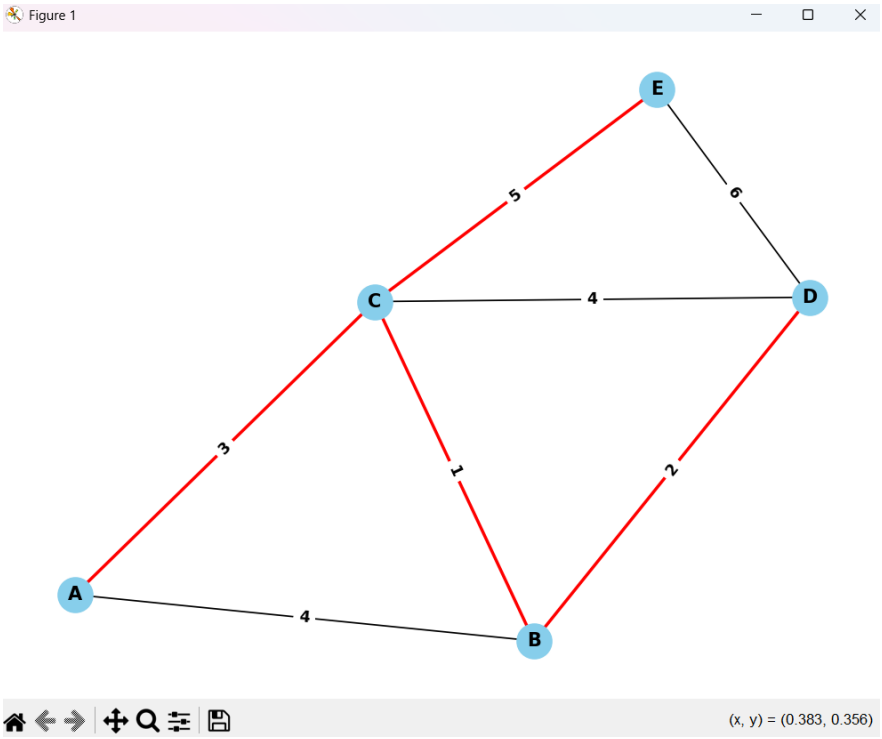
Prim's Algorithm:

Grows a single tree and adds a light edge (edge with the lowest weight) in each iteration

Steps:

1. Start by picking any vertex to be the root of the tree.
2. While the tree does not contain all vertices in the graph, find shortest edge leaving the tree and add it to the tree.

Kruskal's Output:

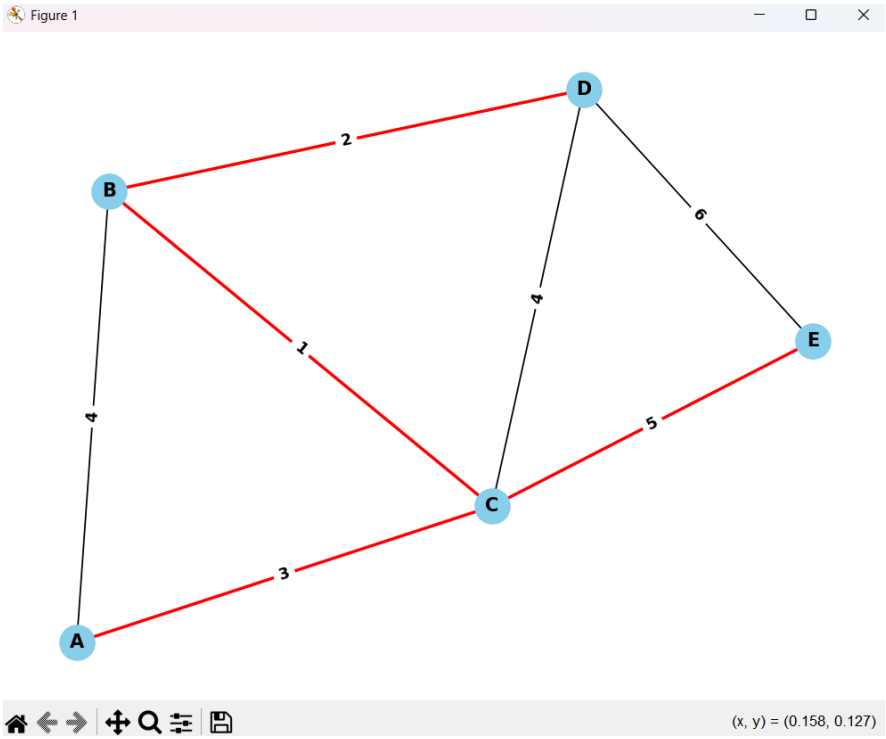



```
PS C:\Users\Lenovo\Documents\COMP 314 Algo\Algorithms Lab\Lab-2> py
thms Lab\Lab-2\mst_kruskal.py"
(Kruskal's Algorithm) Minimum Spanning Tree Edges and Weights:
Edge: (B, C), Weight: 1
Edge: (B, D), Weight: 2
Edge: (A, C), Weight: 3
Edge: (C, E), Weight: 5
PS C:\Users\Lenovo\Documents\COMP 314 Algo\Algorithms Lab\Lab-2>
```

Analysis:

The highlighted red edges form the minimum spanning tree. The edge (B, C) is the edge having the least weight i.e. 1. It connects vertices B and C. Similarly, edge (B, D) with weight 2 connects B and D. Then edge (A, C) with weight 3 connects vertices A and C. Lastly, edge (C, E) connects, vertices C and E. This concludes a formation of minimum spanning tree containing 5 vertices and 4 edges.

Prim's Output:




```
PS C:\Users\Lenovo\Documents\COMP 314 Algo\Algorithms Lab\Lab-2> pyth
thms Lab\Lab-2\mst_prim.py"
Minimum Spanning Tree Edges and Weights:
Edge: (A, C), Weight: 3
Edge: (C, B), Weight: 1
Edge: (B, D), Weight: 2
Edge: (C, E), Weight: 5
PS C:\Users\Lenovo\Documents\COMP 314 Algo\Algorithms Lab\Lab-2>
```

Analysis:

Here, A is the starting node. The edge (A, C) has the least weight i.e. 3 starting from A. Then, edge (C, B) leaving from C has weight 1, which is the minimum among possible paths. After that, edge (B, D) having weight 2 connect vertices B and C. Finally, edge (C, D) connects the last vertex E with weight 5. The highlighted red edges form the minimum spanning tree.

Huffman Coding:

Coding is assigning binary codewords to (blocks of) source symbols. Huffman coding is a lossless data compression algorithm. In this, we assign variable-length codes to input characters, based on the frequencies of corresponding characters. Instead of using ASCII codes, we store the more frequently occurring characters using fewer bits and less frequently occurring characters using more bits.

There are mainly two major parts in Huffman Coding

- Build a Huffman Tree from input characters.
- Traverse the Huffman Tree and assign codes to characters.

Building a Huffman tree:

1. Organize the entire character set into a row, ordered according to frequency from highest to lowest (or vice versa). Each character is now a node at the leaf level of a tree.
2. Find two nodes with the smallest combined frequency weights and join them to form a third node, resulting in a simple two-level tree. The weight of the new node is the combined weights of the original two nodes.
3. Repeat step 2 until all of the nodes, on every level, are combined into a single tree.

Now we assign codes to the tree by placing a 0 on every left branch and a 1 on every right branch. A traversal of the tree from root to leaf gives the Huffman code for that particular leaf character. These codes are then used to encode the string.

Uncompression: Read the file bit by bit

1. Start at the root of the tree
2. If a 0 is read, head left
3. If a 1 is read, head right
4. When a leaf is reached, decode that character and start over again at the root of the tree

Output:

```
PS C:\Users\Lenovo\Documents\COMP 314 Algo\Algorithms Lab\Lab-2> python -u "c:\Users\Lenovo\Documents\

Step 1: Frequency Table:
defaultdict(<class 'int'>, {'d': 1, 'a': 2, 't': 3, 's': 2, 'r': 2, 'u': 2, 'c': 1, 'e': 1})

Step 2: Huffman Tree Built Successfully.

Step 3: Huffman Codes Generated:
Character: e --> Code: 000
Character: u --> Code: 001
Character: t --> Code: 01
Character: r --> Code: 100
Character: s --> Code: 101
Character: a --> Code: 110
Character: c --> Code: 1110
Character: d --> Code: 1111

Step 4: Encoded Text:
11111100111010101100001111001001100000101

Step 5: Decoded Text:
datastructures
PS C:\Users\Lenovo\Documents\COMP 314 Algo\Algorithms Lab\Lab-2>
```

Analysis:

Given text is “datastructures”. In step 1, a frequency table is generated. For instance, t has frequency 3 so, it is assigned the least number of codes which is 01. Similarly, the characters are assigned the code according to their frequency. Then the given text is encoded using the code. Decompression is applied to decode the text.

Backtracking:

Backtracking is a problem-solving algorithmic technique that involves finding a solution incrementally by trying different options and undoing them if they lead to a dead end. When a dead end is reached, the algorithm backtracks to the previous decision point and explores a different path until a solution is found or all possibilities have been exhausted.

N-Queens Problem:

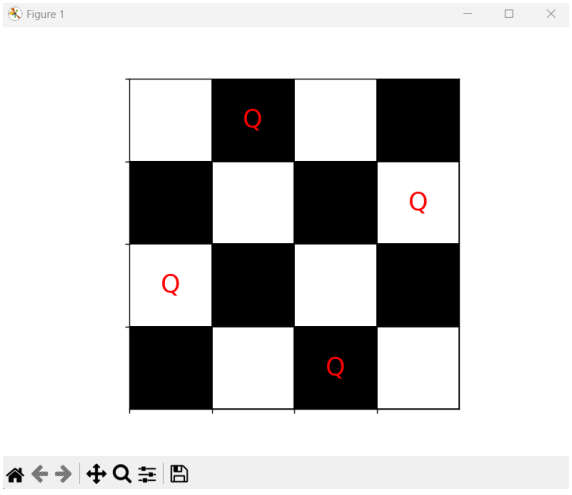
- N queens are to be placed on an N x N chessboard without attacking each other.
- All solutions to the N-queens problem can be represented as n-tuples ($x_1, \ldots x_n$), where x_i is the column on which each queen i is placed.
- Explicit constraints:
  - $S_i = \{1, 2, \ldots, n\} \circ 1 \leq x_i \leq n$
- Implicit constraints:
  - No two x_i's can be the same.
  - No two queens can be on the same diagonal

N-queens problem: Backtracking algorithm

1. Place a queen on the first available square in row 1.
2. Move onto the next row, placing a queen on the first available square there (that doesn't conflict with the previously placed queens).
3. Continue in this fashion until either:
   a. Solved the problem, or
   b. Got stuck. When we get stuck, remove the queens that got us there, until we get to a row where there is another valid square to try.

```
PS C:\Users\Lenovo\Documents\COMP 314
Enter the number of queens: 4

Current Board State:
Q . . .
. . . .
. . . .
. . . .



Current Board State:
Q . . .
. . Q .
. . . .
. . . .



Current Board State:
Q . . .
. . . Q
. . . .
. . . .



Current Board State:
Q . . .
. . . Q
. Q . .
. . . .
```

```
Current Board State:
. Q . .
. . . .
. . . .
. . . .



Current Board State:
. Q . .
. . . Q
. . . .
. . . .



Current Board State:
. Q . .
. . . Q
Q . . .
. . . .



Current Board State:
. Q . .
. . . Q
Q . . .
. . Q .



Solution Found using Backtracking.
PS C:\Users\Lenovo\Documents\COMP 314
```



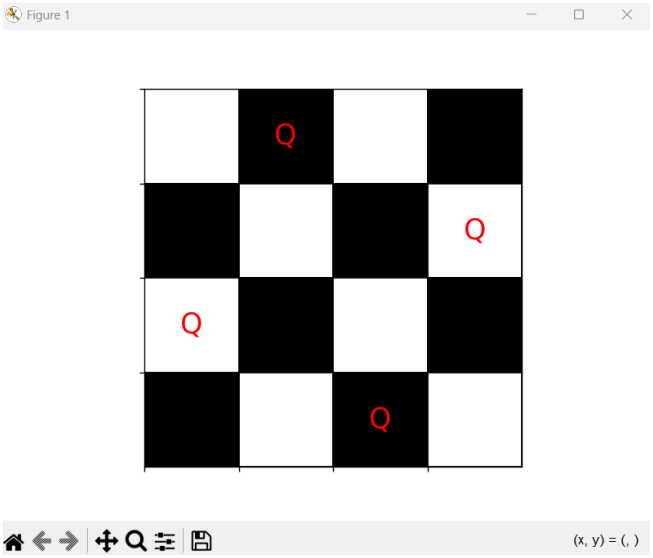
Analysis:

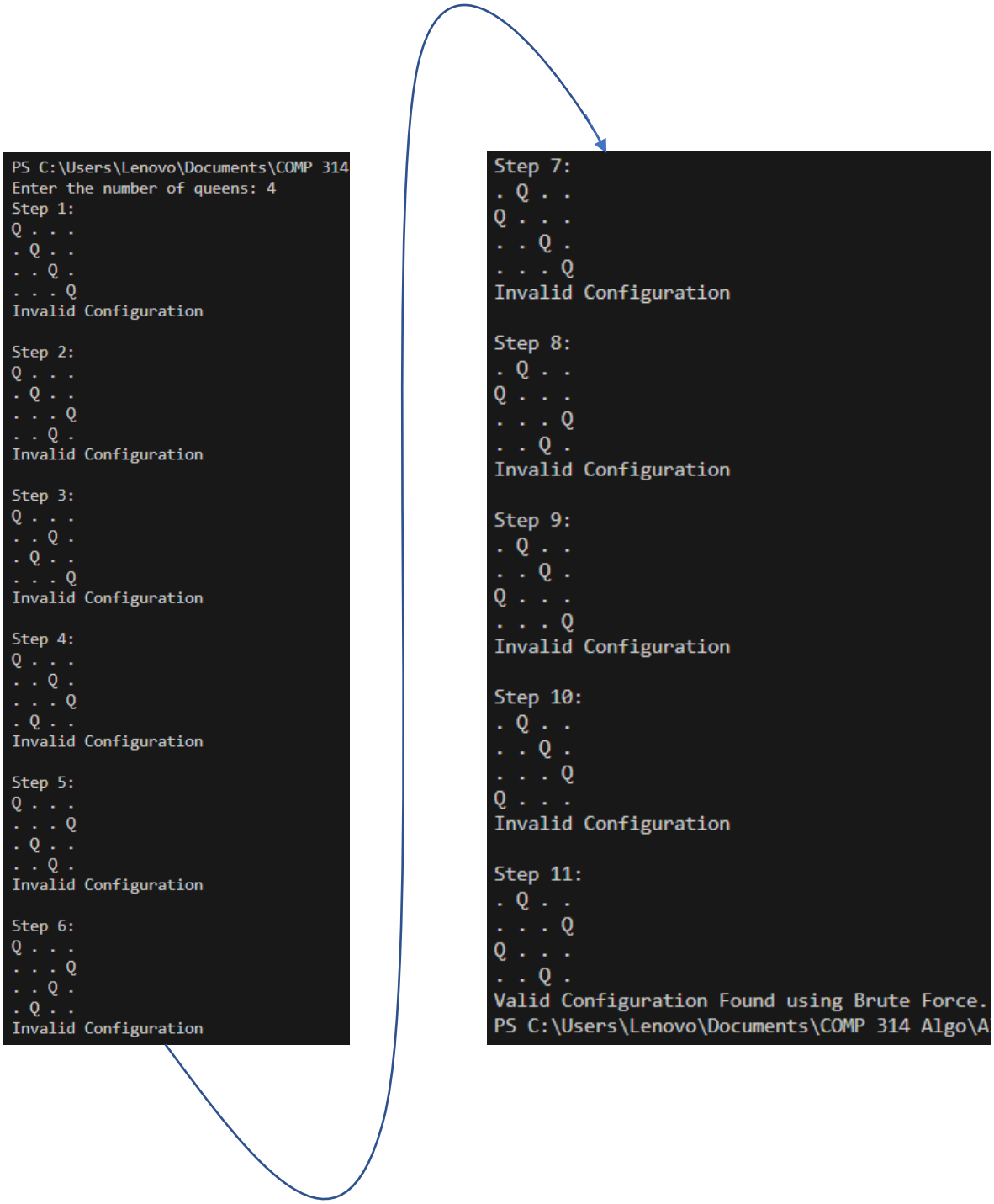
Here, N = 4 is taken. The first queen is placed at (1). Then second queen is at a position where first queen cannot attack it. So, the combination is (1, 3). Now, third queen has either column 2 or 4 but, it can be attacked by the second queen. We cannot place third queen so, it backtracks to the second queen that got there. Now, the placement is (1, 4). Third queen is safe with positions (1, 4, 2). Fourth queen has no safe placement so, it backtracks to the first queen that got this combination. The first queen is placed at (2). Safe placement for second queen is (2, 4), followed by third queen (2, 4, 1). The final queen is place with combination (2, 4, 1, 3) which is valid and hence, the problem for 4-queens is solved.

Brute-force:

Brute-force is a straightforward approach to solving a problem, usually directly based on the problem statement and definition. It systematically enumerates all possible candidates for the solution and checks whether each candidate satisfies the problem statement.

Output:

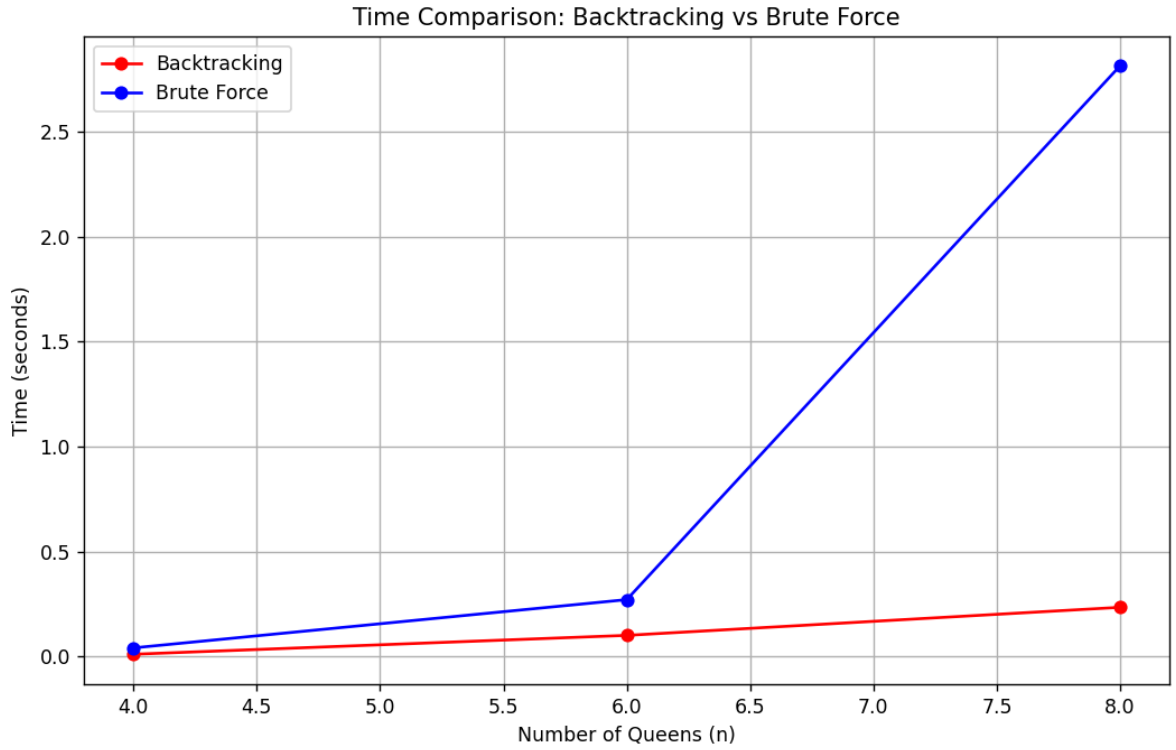




Analysis:

Brute-force checks every possible combination that exists. Here N = 4 is taken for comparison with backtracking. We can see that even if the placement of queen is wrong, it still checks for that possibility. We can see that it took 11 steps to find the first possible solution whereas backtracking took only 8 steps to find the first valid sequence. The difference is small in this case but as the value of n increases, the difference in becomes very high.

Backtracking vs Brute-force:




```
Backtracking Time for n=4: 0.011727 seconds
Brute Force Time for n=4: 0.041895 seconds

Backtracking Time for n=6: 0.101540 seconds
Brute Force Time for n=6: 0.271660 seconds

Backtracking Time for n=8: 0.235034 seconds
Brute Force Time for n=8: 2.812257 seconds
PS C:\Users\Lenovo\Documents\COMP 314 Algo\Algorithms Lab\Lab-2>
```

Analysis:

For n = 4, time for backtracking is 0.01s whereas, for brute-force is 0.04s. The difference is not so great compared to the other values of n. For n = 6, we can see that backtracking completes it in 0.10s, but brute-force takes 0.27s. The difference is much greater for n = 8. It takes 0.23s and 2.81s for backtracking and brute-force respectively to get first valid solution. The difference is much greater as we move towards the larger value of n. Using brute-force, the time complexity of the n-queens problem is $O(n^n)$ but using backtracking it is reduced to O(n!).