COSC2658 - Data Structures & Algorithms - Group 2

Group Project

Date: Jan 9, 2022

Team members:

1. Tran Dang Bao Nhi (s3751881)
2. Nguyen Vu Thuy Duong (s3865443)
3. Nguyen Quoc Cuong (s3748840)
4. Nguyen Thi Thuy Tien (s3757934)

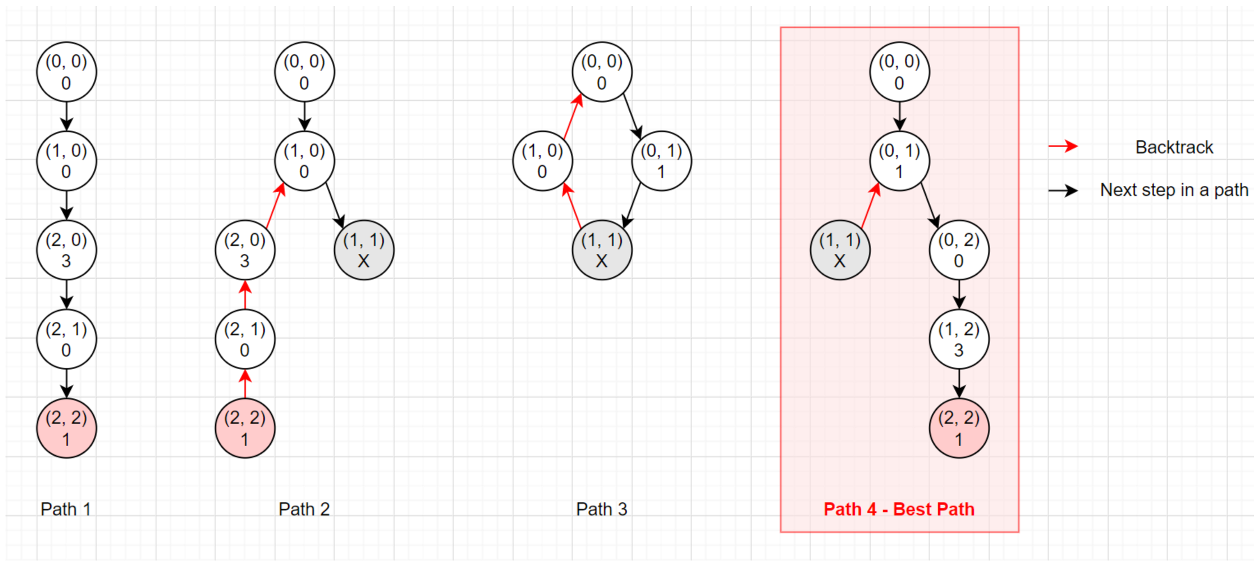
Problem Definition

The Greedy Gnomes Problem can be classified as a shortest path problem. To be specific, a matrix of known size with each cell containing one of the following: "." for land, "X" for rock and a positive integer for the number of gold in that position is given. The task is to find the shortest path with the most gold starting from the starting position (top left corner) where movements can only be rightward or downward and we can not go through cells containing rock.

For the assignment, the team must find two solutions using exhaustive search and dynamic programming approaches and measure their performance and compare them.

Description of the Algorithms

1. Exhaustive Search Algorithm



Given a matrix of size 3x3. The problem resembles a tree with starting position (0, 0) as the root of the tree. When we pass each node in this matrix, we would check its type of value, whether it is a number bigger than zero (gold), a zero (dot), or an X (rock). For the first path, we travel down to the node (1, 0), its value is 0, so it is able to continuously travel down to the node (2,0). This position has a value of 3 (gold), this value would be added to the sum of the path. As the path has reached the last row, it is unable to move down. Therefore, it starts moving right to the node (2, 1) and (2, 2) (final node of the matrix). The checking processes are similar to the two previous nodes. After reaching the final node, the first path has ended. To start the second path, we backtrack to the node of the previous row (1, 0), then travel to the right-hand side to the node (1, 1). The node (1, 1) has the value ‘X’, so this path has been blocked. We have to backtrack to the node (0, 0) as there are no other available paths from the node (1, 0). As we traverse through the tree, we find all the possible paths. The selected path would have the best sum and the least steps, which is the path travels through the node (2, 0) with the sum is 5 and the total steps are 4.

To solve the similar problem for a matrix size M*N, we use depth-first search algorithm with following functions:

1. Subproblem 1: depthFirstSearch(row, col, map, visited)

This subproblem is to check the validation of the next step and add a new step to the current path as well as update the path in the path list.

2. Subproblem 2: isOptimalPath

This subproblem is to compare between the 2 paths, the just-completed path with the current best path to find the most optimal path that obtains the highest number of gold and least steps.

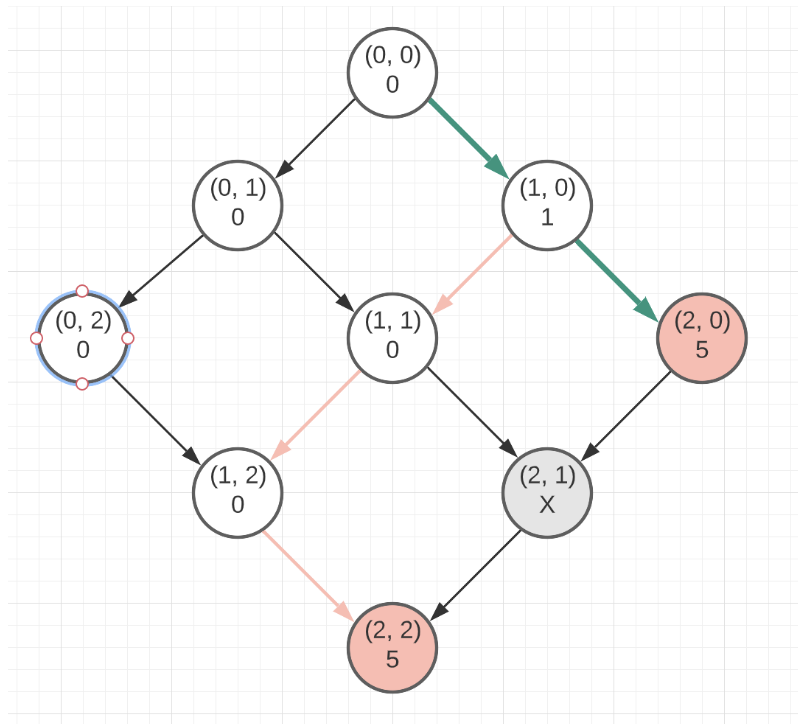
3. Inorder traversal: Left-Current-Right

For exhaustive search, the problem is like a tree that we go from the root to each branch then move to the next one until reaching the last branch of the tree. To solve this problem, we check all the nodes in the left (or down direction) of each row first, then move to the right (or right direction).

4. Time analysis: O(2^m*n)

Since we use depth-first search, we ensure that the maximum time complexity for visiting all the nodes is O(2^m*n), which would be discussed in detail in the next section.

2. Dynamic Programming Algorithm



Given a matrix of size 3x3. Since we can only go at max 2 directions at any position, the problem resembles a binary tree with the starting position (0,0) as the root of the tree. Both nodes (0, 1) and (1, 0) go to node (1, 1), hence, we can remember the result of node (1, 1) so that we don't have to compute the node twice. As we traverse through the tree, we find all possible endings including node (2,2), which is the final node, and node (2,0), which is blocked by a rock node (2,1). Starting from the possible endings, we compute backward the accumulated steps and gold collected to select the best path. For example, node (1, 0) can go to both node (1, 1) or node (2, 0), all paths have the same accumulated gold. However, the path to node (2, 0) is selected as it has the shortest path among the two.

To solve the similar problem for a matrix of size MxN, we apply the SRTBOT recursive algorithm paradigm to describe our algorithm. To be more specific, the best path is defined as the shortest path with maximum gold collected.

1. Subproblem: find_max(row, col)

The sub problem here is to find the best path from any position.

2. Relate: find_max(row, col) = value(row, col) + max{find_max(row + 1, col), find_max(row, col + 1)}

Since we only have two directions to go, the best path from the current position node is the best one between the best path of going down and the best path of going right.

3. Topological order: decreasing row and column

The topological order is to make sure the problem is acyclic. For this problem, we go from the leaves to the root of the binary tree presented in the next section. So the row and column is guaranteed to decrease to zero.

4. Base: find_max(end node) = value(end node)

The best path of the end node, which is the node with no possible next moves, is itself.

5. Original problem: find_max(0, 0)

The original problem is to find the shortest path with maximum gold from the starting position.

6. Time analysis: O(n)

Since we apply memoization, we make sure all nodes are visited at max once so the time complexity is O(n). Time analysis is discussed in more detail in the next section.

Pseudocode for the algorithms and Complexity Analyses

1. Exhaustive Search Algorithm

a. Pseudocode

```
class Node

    int row, col, value

    Node(row, col)                  // Constructor

    int getRow()                    // get the row order - y-coordinator

    int getCol()                    // get the column order - x-coordinator
    void setValue(value)                    // determine the value of that position A(row, col)


// Create a path, which consists of the number of gold and steps
class Path
    // arraylist contains all nodes in one path, would be updated when moving to the next step
    until the path reaches its boundary
    ArrayList<Node> path
    goldValue = 0, step = 0          // Initialize the total golds and steps obtained in a path
    Path()                           // Constructor
    void add(Node node)              // Create a function to add new node to the path
    void print()                     // Create a function to check and track the path
    ArrayList<Node> getPath()
    void setPath(path)               // determine the path when a path is updated or created
    int stepCount()                  // get the current total steps of a path
    int getGoldValue()
    void setGoldValue((goldValue)
    int getStep() return step
    void setStep(step)  this.step = step


class ExhaustiveSearch
    Path onePath                     // onePath is the current tracking path
    Path bestPath                    // this is the current most optimal path
    String path = "";                //this is the direction for the current tracking path


    method depthFirstSearch(row, col, map, visited,count)
```

```
count[0]++;

curNode = new Node(row, col);


// check if the node has been visited or not, or the next step is blocked by the wall
(boundary of the map) or by rock

if ((row or column of curNode is out of range) or (visited[row][col] is true or
contained rock)

        return

if (the position of the current cell is the end of the map or next movement of
curNode is both horizontally blocked and vertically blocked)

       if (value at position map[row][col] is a number)

              set value of that position

       onePath.add(current cell)

       //make a new path with current tracking path but remove all unnecessary
       element after the last gold value is found in the path (such as  ".")

       Path newPath = removeRedundant(onePath);

       // set direction for the newPath as a substring of  (String) path from 0 to
       the size-1 of the new path.

       newPath.direction = path.substring(0,newPath.stepCount());

       if (isOptimalPath(newPath,bestPath)) {

              set bestPath to that new path

       }

       onePath = copyArrayList(onePath);`

       return;


// Try for all the 2 directions (down,right) in the given order to get the
// paths in lexicographical order

// Check if downward move is valid

if (isSafe(row + 1, col, map, rows, cols, visited)) {

        path += 'D';                    // adding direction to path

        // if the current tracking path does not contain the cell at that position,
        add it to the path.

        if (!onePath.getPath().contains(curESNode)) {

                onePath.add(curESNode);

                // if the value of the cell is different from 0 then set the current
                node's value to that value.

                if (Utility.parseValue(map[row][col]) != 0) {

                        curESNode.setValue(Utility.parseValue(map[row][col]));

                        // increase the total of that path value to the number of current
                        node values.

                        onePath.goldValue += curESNode.value;

                  }

        }

        // call a recursive function to check the next right cell of this cell.

        depthFirstSearch(row + 1, col, map, rows, cols, visited, count);

        // below lines of code only run when from the cell we could not go in the down
        direction anymore. Backtracking until we found the last point we can find a
        valid direction, this time going in the right direction.

        // backtracking -> remove the last direction

        path = path.substring(0, path.length() - 1);

        // take the last node of the current tracking path.

        ESNode ESNode = onePath.getPath().get(onePath.getPath().size() - 1);
        // backtracking -> remove that last node from the path

        onePath.path.remove(ESNode);

        // decrease current path gold value with the value of that cell.

        onePath.goldValue -= ESNode.value;

    }


    // Check if the right move is valid, logic is same as the down direction block code.

    if (isSafe(row, col + 1, map, rows, cols, visited)) {

        path += 'R';

        if (!onePath.getPath().contains(curESNode)) {

            onePath.add(curESNode);

            if (Utility.parseValue(map[row][col]) != 0) {

                curESNode.setValue(Utility.parseValue(map[row][col]));

                onePath.goldValue += curESNode.value;

            }

        }

        depthFirstSearch(row, col + 1, map, rows, cols, visited, count);

        // below lines of code only run when from the cell we could not go in the
        right direction anymore. Backtracking until we find the last point we can find
        a valid direction, this time going in the down direction or end of path.

        path = path.substring(0, path.length() - 1);

        ESNode ESNode = onePath.getPath().get(onePath.getPath().size() - 1);

        onePath.path.remove(ESNode);

        onePath.goldValue -= ESNode.value;

    }
```

```
        // Mark the cell as unvisited for

        // other possible paths

        visited[row][col] = false;

    }


method isOptimalPath( Path newPath, Path bestPath)

        boolean flag = newPath.goldValue > optimalPath.goldValue;

        if (compare if the step of newPath is less than bestPath if  newPath gold Value ==
        best Path Gold Value )

            flag = true

            return flag;
```

b. Complexity

Input size: M*N (M, N is the number of row and column of input matrix G respectively)

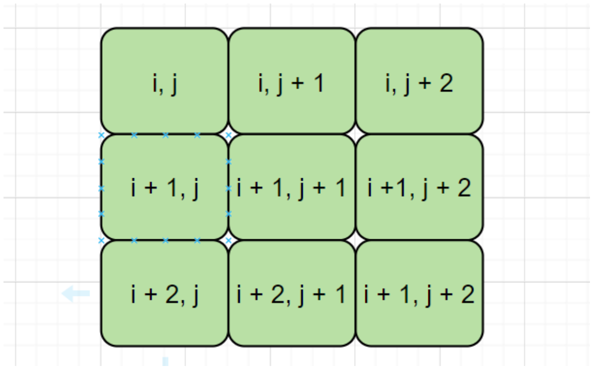
Termination condition: node is completely blocked both directions (either by rock, wall or both)

Number of basic operation (comparisons, additions, assignments) each recursion: constant A

Recurrence at one position (comparisons, checking conditions):

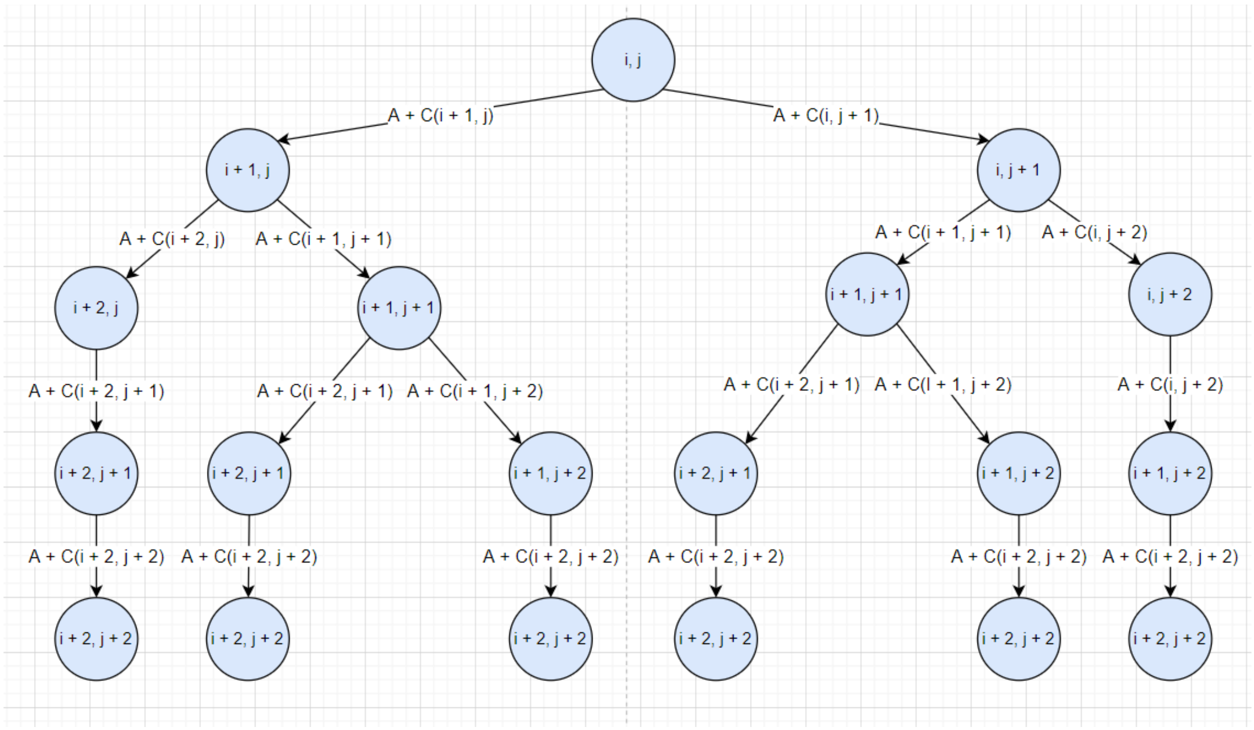
As the execution time of each step approximates the other steps. Therefore, we assume that each step would have the same value of time.

i. Position (i + 2, j + 2)

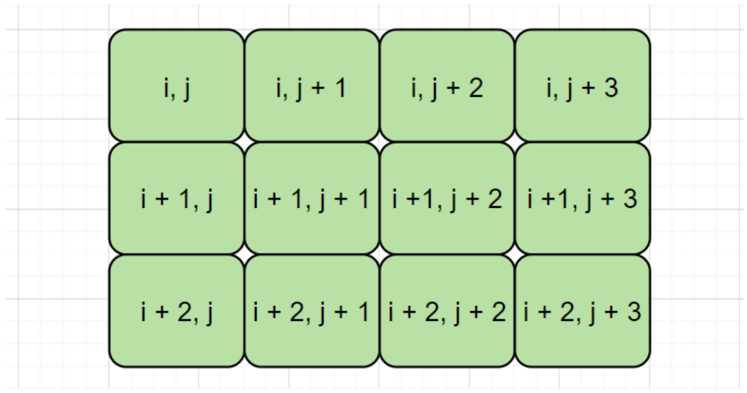


Based on the tree diagram below, there are 18 steps in total to get through all possible directions in the matrix 3x3 from position (i, j) to position (i + 2, j + 2).

18 = 2^(2+2) + 2, approximately 2^(2+2) -> O(2^(M+N)) time complexity.

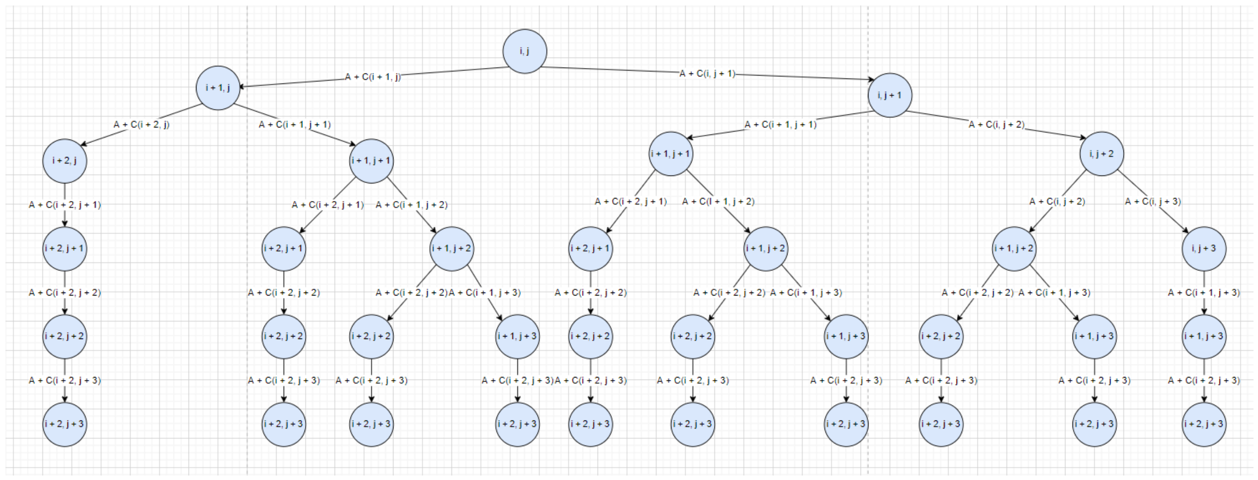


ii. Position (i + 2, j + 3)



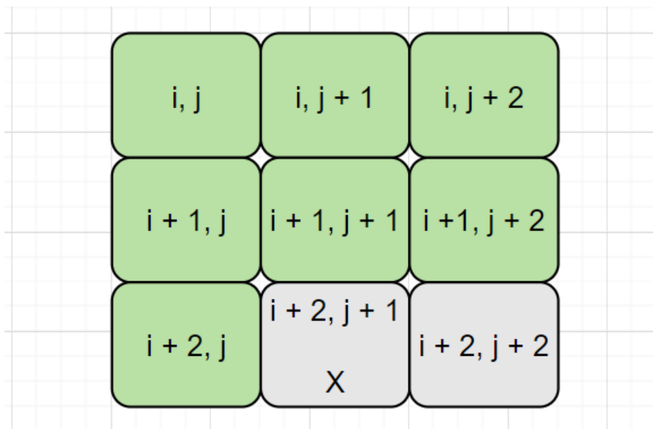
Similarly to the position (i + 2, j + 2),it takes 33 steps in total to pass all possible directions from (i, j) to the position (i + 2, j + 3), which equals to 2^(2+3) + 1, approximately 2^(2+3) - O(2^(M+N)) time complexity.

From the two given examples, we can see that, for the exhaustive search method, the sum of the number of rows and columns has affected the time complexity of the program in an exponential manner (2^(M+N)).

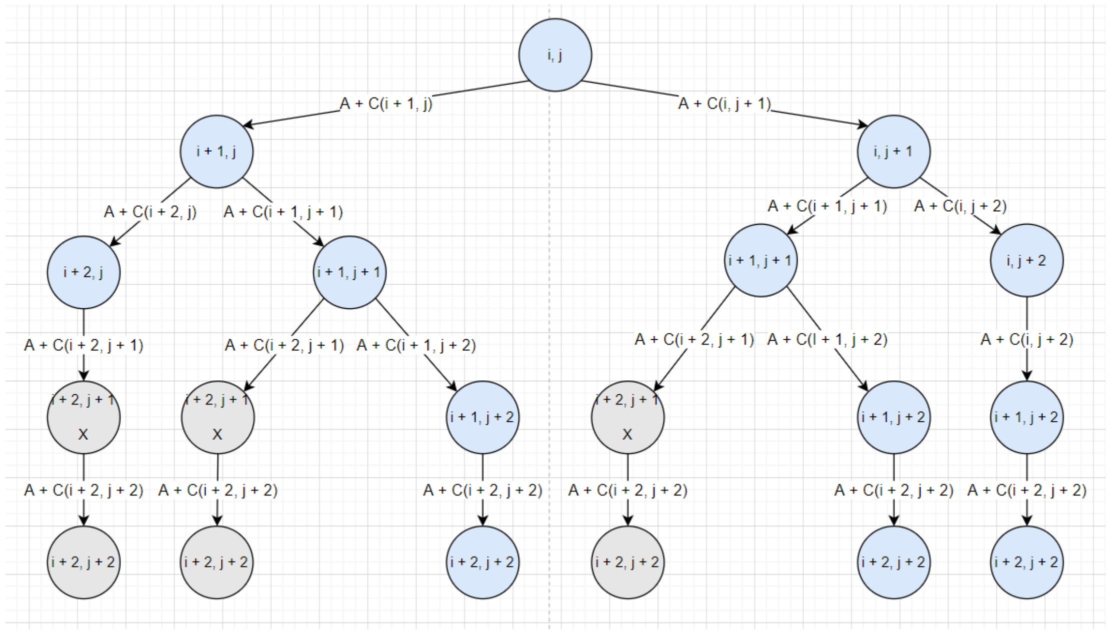


Hence, with the matrix G of size MxN, starting from position G(0, 0), visiting all nodes excluding the final one (M x N - 1 nodes) by going downward then rightward to get all the possible paths to reach the final position G(M, N) would have the following result.

C(0, 0) = (2^(M+N))*A -> O(2^(M+N) time complexity



In case that the map includes rocks, the big O notation has not been changed. In the below situation, the rock is contained in the position (i + 2, j + 1), so the step from (i + 2, j + 1) to (i + 2, j + 2) has been blocked. As a result, the number of steps reduces to 3 steps. The difference between step (i + 2, j +1) and step (i + 2, j + 2) is (0, 1). 3 = 2^(0 + 1) + 1, approximately 2^(M+N).



In conclusion, depth-first search allows the algorithm to visit each node at once or more times based on the size of the map and collect all the possible paths including the best one, resulting in O(2^n) time complexity.

2. Dynamic Programming Algorithm

a. Pseudocode

```
findMax(map, row, col, computedNodesList)

     Node node = new Node()

     computedNodesList[row][col] = node                  //storing the current node



     //termination condition

     if (node is blocked)

           node.accummulatedValue = map(row, col)

            //only count step if node has gold

            if node.accummulatedValue != 0

                 node.stepCounts = 1

           return node



     //check if node is not blocked

     if (node is not blocked in any direction)

           //check if rightNode already exists to retrieve it or to calculate it

           if computedNodesList[row][col+1] exists

                 rightNode = computedNodesList[row][col+1]

           else

                 rightNode = find_max(map, row, col+1,computedNodesList)



           //check if downNode already exists to retrieve it or to calculate it

           if computedNodesList[row + 1][col] exists

                 downNode = computedNodesList[row + 1][col]

           else
```

```
                downNode = find_max(map, row + 1, col,computedNodesList)


        //compare 2 nodes and get the better one

        if rightNode has more gold than downNode or if rightNode has equals gold but shorter
        than downNode

                node.child = rightNode

                node.accummulatedValue = rightNode.accummulatedValue + map[row][col]

                //only count step if node has accumulated gold

                if node.accummulatedValue != 0

                        node.stepCounts++

        else

                node.child = downNode

                node.accummulatedValue = downNode.accummulatedValue + map[row][col]

                //only count step if node has accumulated gold

                if node.accummulatedValue != 0

                        node.stepCounts++


//check if not is vertically blocked

if (node is vertically blocked)

        //check if downNode exists to retrieve it or to calculate it

        if computedNodesList[row + 1][col] exists

                downNode = computedNodesList[row + 1][col]

        else

                downNode = find_max(map, row + 1, col,computedNodesList)

        node.child = downNode
        node.accummulatedValue = downNode.accummulatedValue + map[row][col]
        //only count step if node has accumulated gold
        if node.accummulatedValue != 0
            node.stepCounts++

    //check if not is horizontally blocked
    if (node is horizontally blocked)
        //check if downNode exists to retrieve it or to calculate it
        if computedNodesList[row][col + 1] exists
            rightNode = computedNodesList[row][col + 1]
        else
            rightNode = find_max(map, row, col + 1, computedNodesList)
        node.child = rightNode
        node.accummulatedValue = rightNode.accummulatedValue + map[row][col]
        //only count step if node has accumulated gold
        if node.accummulatedValue != 0
            node.stepCounts++
    return node
```

b. Complexity analysis

Input size: M*N (M, N is the number of row and column of input matrix G respectively)

Termination condition: node is completely blocked both directions (either by rock, wall or both)

Number of basic operation (comparisons, additions, assignments) each recursion: constant A

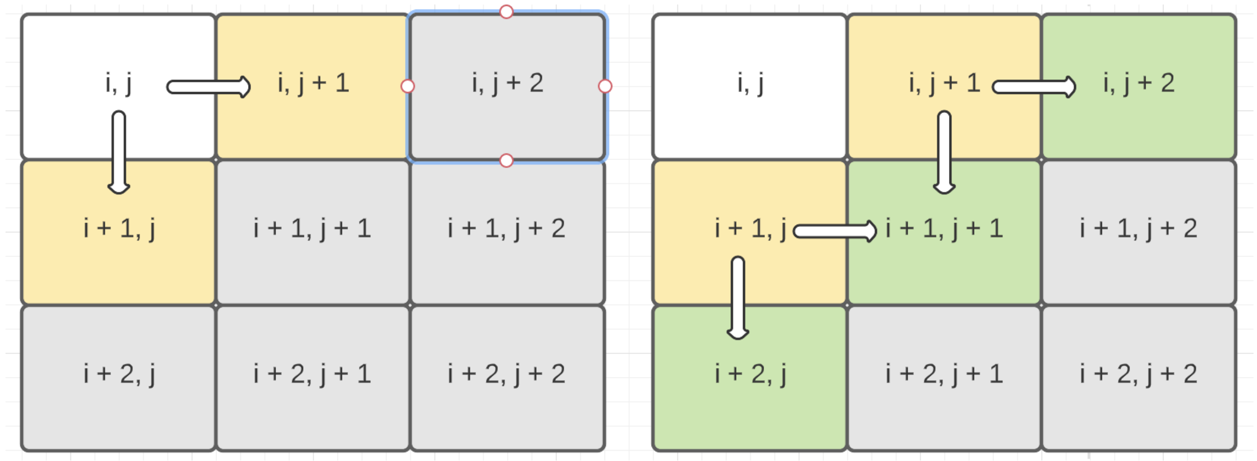
Recurrence at one position (calculate, compare and choose the best path between going downward and rightward): $C(i, j) = A + C(i + 1, j) + C(i, j + 1)$

$C(i + 1, j) = A + C(i + 1, j + 1) + C(i + 2, j)$

$C(i, j + 1) = A + C(i + 1, j + 1) + C(i, j + 1)$

Since we use memoization, we remove the burden of recalculating $C(i + 1, j + 1)$. Hence, we have:

$C(i, j) = 3A + C(i + 1, j + 1) + C(i + 1, j) + C(j, i + 1)$



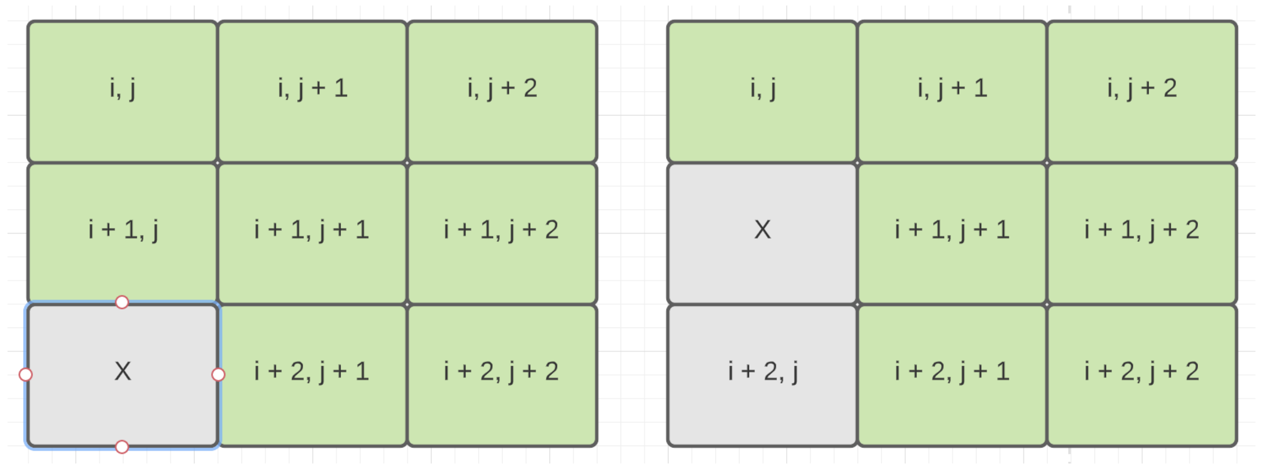
The above equation shows that visiting n nodes would result in nA execution time plus all next possible moves. To be more specific, visiting node G(i, j) only results in A execution time following 2 possible next moves, continuing to visit node G(i + 1, j) and G(i, j + 1) results in a total 3A execution time following 3 possible next moves as demonstrated in the above figure.

Hence, with the matrix G of size MxN, starting from position G(0, 0), visiting all nodes excluding the final one (M x N - 1 nodes) by going downward and rightward to reach the final position G(M, N) would have the following result.

$C(0, 0) = (M * N - 1)A + C(M, N)$

Since G(M, N) is the final node, the function C(M, N) only takes A execution time without a further next possible recursive move. Hece, we have:

C(0, 0) = (M * N - 1)A + A = (M * N)A -> O(n) time complexity



Given there are rocks in the matrix, all the rocks would not be visited. At the same time, the rocks and the walls would make some nodes unreachable as demonstrated in the figure above. Given the number of rocks equals X.

C(0, 0) <= (M * N - X)A -> O(n) time complexity

Equals sign happens when the rocks and the walls don't make any other nodes unreachable.

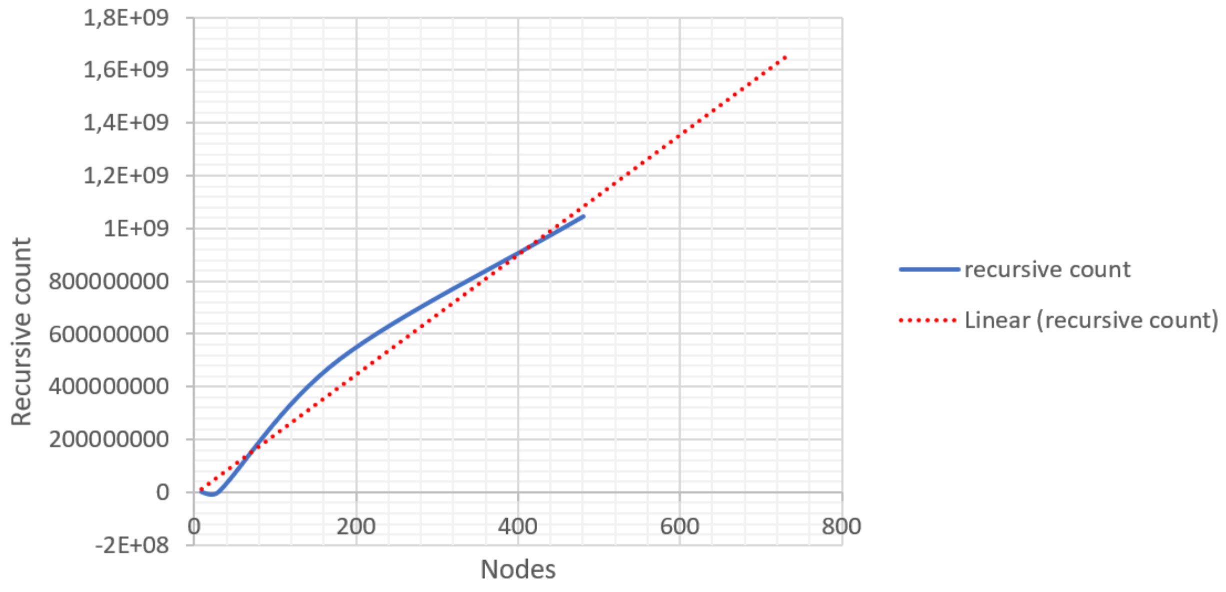
In conclusion, memoization allows the algorithm to visit each node at max once, resulting in O(n) time complexity.

Graphs and Empirical Analyses

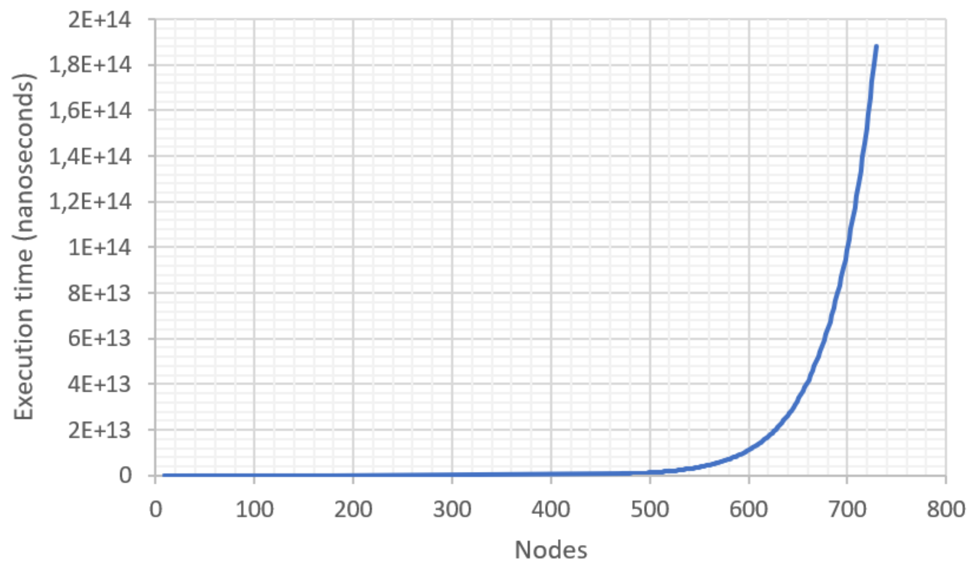
1. Exhaustive Search Algorithm

nodes	recursive count	execution time (nanoseconds)
9 (3x3)	7	28848089
30 (5x6)	68	35740074
180 (12x15)	504884248	961543720
480 (20x24)	1045114201	817282255623
729 (27x27)	1.65E+09 (estimated)	1.88E+14 (estimated)

Using the aforementioned algorithm, several experiments have been tested on different map sizes to prove the speed of the approach. All the experiments happened in the same conditions with the same system for empirical comparison. As seen from the table above, we gradually increase the total nodes (map size) and see the yields of recursive calls as well as the execution time for each size.



Based on the above figure, the number of recursive calls increases significantly correlated with the number of nodes, which approves that the number of nodes (map size) affects the visiting time through a point. As the map expands, one node has been visited more frequently.

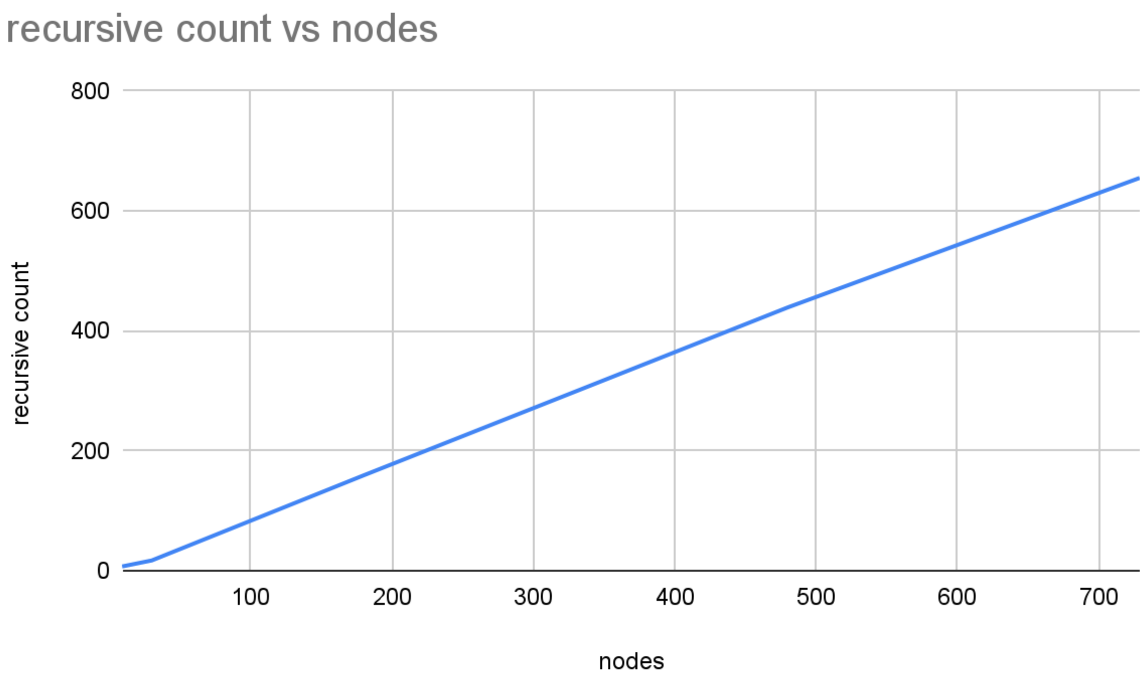


The above graph illustrates the execution time of the algorithm. The time increases exponentially correlated with the number of nodes.

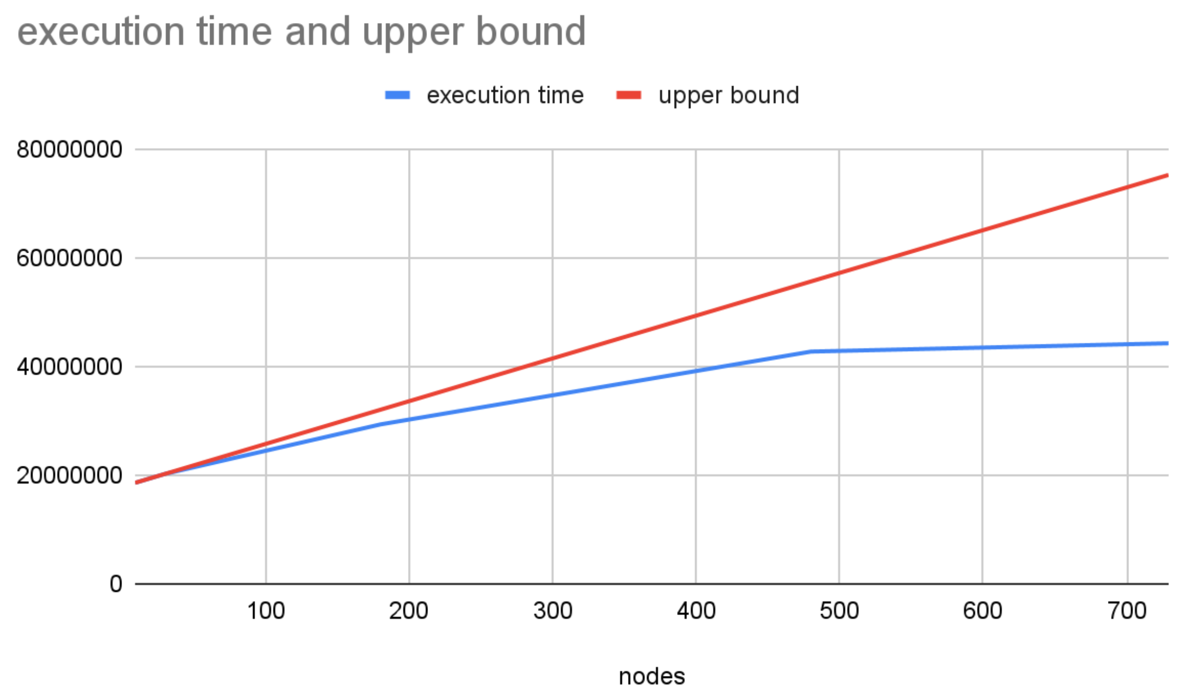
2. Dynamic Programming Algorithm

nodes	recursive count	execution time	upper bound
9 (3x3)	7	18594919	18594919
30 (5x6)	17	20246512	20246512
180 (12x15)	159	29338591	32043605
480 (20x24)	439	42730979	55637791
729 (27x27)	655	44254207	75220965

Using the aforementioned algorithm, several experiments have been done to test the speed of this approach. All the experiments are run against the same machine for empirical comparison. As seen from the table above, we gradually increase the number of nodes and see the yielded results of recursive call execution time, along with a pre-computed theoretical upper bound that is an indicator for linear time complexity.



As can be seen from the graph, the recursive call is positively correlated with the number of nodes in the matrix with almost one by one ratio. This supports our analysis that each node is visited at most once.



According to the above graph, the execution time is positively correlated with the number of nodes and is under the theoretical upper bound suggesting a linear time complexity.

3. Comparison

From the analyzed data, it is concluded that the dynamic programming method has a higher efficiency than the exhaustive search one.

For time complexity, the dynamic programming algorithm returns results in O(n) time corresponding to the number of nodes (the size of map), while the exhaustive search one takes O(2^(m+n)) time - much longer. With the map 27x27, the program running dynamic method runs in 44254207 nanoseconds (small amount of time). In contrast, the program of exhaustive search one need approximately 1.88E+14 nanoseconds to run (an extremely large amount of time). Hence, dynamic programming yields better results in terms of time complexity.

For space complexity, each node in the dynamic program has been visited once with all map sizes and the optimization paths from each node are stored in an extra array, while in the exhaustive search program, as the map expands, the more time one node has been visited and the more arrays needed to store all possible paths for later comparison. Hence, dynamic programming yields better results in terms of space complexity.

Conclusion

In conclusion, we have successfully implemented two solutions for the Greedy Gnomes Problem applying exhaustive search and dynamic programming. After thorough analysis and comparison, our dynamic programming solution has yielded a linear time complexity compared to exponential time complexity of exhaustive search which has been proven both theoretically and empirically.