TITLE OF THE MINI PROJECT
Solving a snake and ladder using BFS

A MINI PROJECT REPORT

18CSC204J -Design and Analysis of Algorithms Laboratory

Submitted by

NUNE NITESH [RA2011028010140]

MANTHURI HRITHIKESH [RA2011028010133]

Under the guidance of

Dr. B Yamini

Assistant Professor, Department of Networking and Communication

In Partial Fulfillment of the Requirements for the Degree of

BACHELOR OF TECHNOLOGY
in
COMPUTER SCIENCE ENGINEERING
with specialization in CLOUD COMPUTING

SRM INSTITUTE OF SCIENCE & TECHNOLOGY
Deemed to be University u/s 3 of UGC Act, 1956

DEPARTMENT OF NETWORKING AND COMMUNICATIONS

COLLEGE OF ENGINEERING AND TECHNOLOGY SRM

INSTITUTE OF SCIENCE AND TECHNOLOGY

KATTANKULATHUR- 603 203

JUNE 2022

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY
KATTANKULATHUR – 603 203

BONAFIDE CERTIFICATE

Certified that this mini project report titled "" is the bonafide work done by Nune Nitesh (RA2011028010140) and Manthuri Hrithikesh (RA2011028010133) who carried out the mini project work and Laboratory exercises under my supervision for **18CSC204J -Design and Analysis of Algorithms Laboratory**. Certified further, that to the best of my knowledge the work reported herein does not form part of any other work.

Dr. B Yamini
ASSISTANT PROFESSOR
18CSC204J -Design and Analysis of Algorithms
Course Faculty
Department of Networking and Communications

Signature of the Internal Examiner-I **Signature of the Internal Examiner-II**

ABSTRACT

The problem statement that we will be solving is- Given a snake and ladder board, find the minimum number of dice throws required to reach the destination or last cell from source or 1st cell. This is done by considering that the player can determine which number appears in the dice being biased. The player rolls the dice and if reaches a base of a ladder then he can move up the ladder to a different position/cell and if he reaches a snake then it brings him down away from the destination cell/position. This problem can be solved using a Breadth-First Search (BFS).

TABLE OF CONTENTS

LIST OF FIGURES

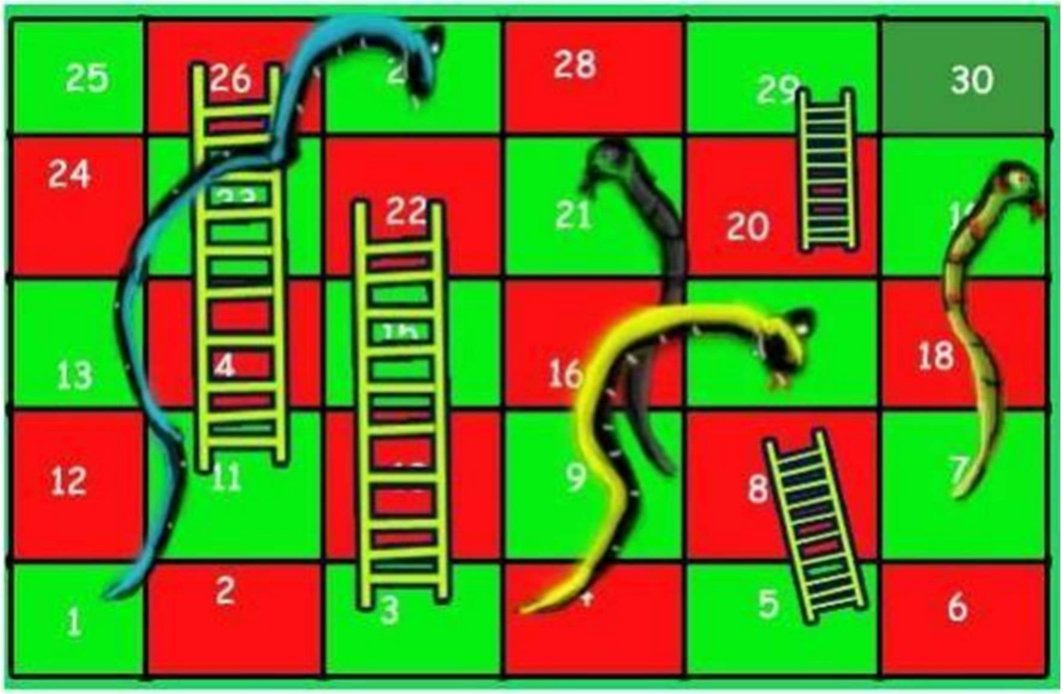
LIST OF SYMBOLS AND ABBREVIATION

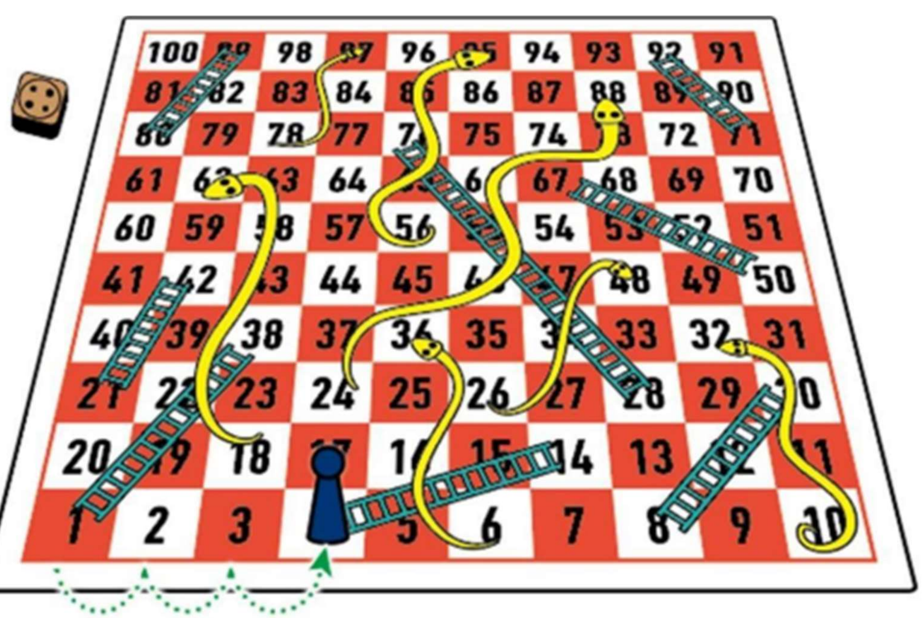
SYMBOLS/ ABBREVIATION	MEANING / EXPANSION
NLU	Natural Language understanding

CHAPTER-1

PROBLEM DEFINITION

Given a snake and ladder board, find the minimum number of dice throws required to reach the destination or last cell from source or 1st cell. Basically, the player has total control over outcome of dice throw and wants to find out minimum number of throws required to reach last cell. If the player reaches a cell which is base of a ladder, the player has to climb up that ladder and if reaches a cell is mouth of the snake, has to go down to the tail of snake without a dice throw.


25 26 28 29 30
24 22 21 20
13 4 16 18
12 11 9 8 7
1 2 3 5 6


100 98 96 94 93 91
81 82 83 84 86 87 88 90
79 78 77 75 74 72 71
61 64 67 68 69 70
60 59 57 56 54 51
41 44 45 48 49 50
38 37 35 33 32 31
23 24 25 26 27 29
20 18 14 13 11
1 2 3 5 6 7 8 9 10

CHAPTER-2

PROBLEM EXPLANATION

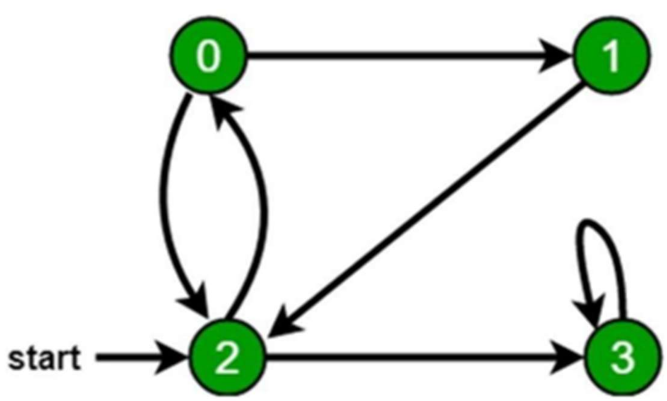
The idea is to consider the given snake and ladder board as a directed graph with number of vertices equal to the number of cells in the board. The problem reduces to finding the shortest path in a graph. Every vertex of the graph has an edge to next six vertices if next 6 vertices do not have a snake or ladder. If any of the next six vertices has a snake or ladder, then the edge from current vertex goes to the top of the ladder or tail of the snake. Since all edges are of equal weight, we can efficiently find shortest path using Breadth First Search of the graph

Following is the implementation of the above idea. The input is represented by two things, first is 'N' which is number of cells in the given board, second is an array 'move[0…N-1]' of size N. An entry move[i] is -1 if there is no snake and no ladder from i, otherwise move[i] contains index of destination cell for the snake or the ladder at i.

CHAPTER-3

DESIGN TECHNIQUES

Breadth-First Traversal (or Search) for a graph is similar to Breadth-First Traversal of a tree . The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a boolean visited array. For simplicity, it is assumed that all vertices are reachable from the starting vertex. For example, in the following graph, we start traversal from vertex 2. When we come to vertex 0, we look for all adjacent vertices of it. 2 is also an adjacent vertex of 0. If we don't mark visited vertices, then 2 will be processed again and it will become a non-terminating process. A Breadth-First Traversal of the following graph is 2, 0, 3, 1.



CHAPTER-4

ALGORITHM FOR THE PROBLEM

Begin

initially mark all cell as unvisited

define queue q

mark the staring vertex as visited

for starting vertex the vertex number := 0 and distance := 0

add starting vertex s into q

while q is not empty, do

qVert := front element of the queue

v := vertex number of qVert

if v = cell -1, then //when it is last vertex

break the loop

delete one item from queue

for j := v + 1, to v + 6 and j < cell, increase j by 1, do

if j is not visited, then

newVert.dist := (qVert.dist + 1)

mark v as visited

```
if there is snake or ladder, then

newVert.vert := move[j] //jump to that location

else

newVert.vert := j

insert newVert into queue

done

done

return qVert.dist

End
```

CHAPTER-5

EXPLANATION OF ALGORITHM

Consider each as a vertex in directed graph. From cell 1 you can go to cells 2, 3, 4, 5, 6, 7 so vertex 1 will have directed edge towards vertex 2, vertex 3….vertex 7. Similarly consider this for rest of the cells. For snake- connect directed edge from head of snake vertex to tail of snakevertex.(Seeexampleimageabove-snakefrom12to2.So directed edge from vertex 12 to vertex 2) For ladder- connect directed edge from bottom of ladder vertex to top of the ladder vertex.

- Now problem is reduced to Shorted path problem. So by Breadth-First Search (using queue) we can solve the problem.

Each vertex will store 2 information, cell number and number of moves required to reach to that cell. (cell, moves)

- Start from cell (vertex) 1, add it to the queue.
- For any index = i, Remove vertex 'i' from queue and add all the

vertices to which can be reached from vertex 'i' by throwing the dice once and update the moves for each vertex (moves = moves to reach cell 'i' + 1 if no snake or ladder is present else moves = cell to which snake or ladder will leads to)

- Remove a vertex from queue and follow the previous step.
- Maintain visited[] array to avoid going in loops.
- Once reach to the end(destination vertex), stop

CHAPTER-6

COMPLEXITY ANALYSIS

Time complexity of the above solution is O(N) as every cell is added and removed only once from queue. And a typical enqueue or dequeue operation takes O(1) time.

Another approach we can think of is recursion in which we will be going to each block, in this case, which is from 1 to 30, and keeping a count of a minimum number of throws of dice at block i and storing it in an array t.

So, basically, we will:

Create an array, let's say 't', and initialize it with -1.
Now we will call a recursive function from block 1, with variable let's say 'i', and we will be incrementing this.
In this we will define the base condition as whenever block number reaches 30 or beyond we will return 0 and we will also check if this block has been visited before, this we will do by checking the value of t[i], if this is -1 then it means its not visited and we move forward with

the function else its visited and we will return value of t[i].

After checking base cases we will initialize a variable 'min' with a max integer value.

Now we will initiate a loop from 1 to 6, i.e the values of a dice, now for each iteration we will increase the value of i by the value of dice(eg: i+1,i+2….i+6) and we will check if any increased value has a ladder on it if there is then we will update the value of i to the end of the ladder and then pass the value to the recursive function, if there is no ladder then also we will pass the incremented value of i based on dice value to a recursive function, but if there is a snake then we won't pass this value to recursive function as we want to reach the end as soon as possible, and the best of doing this would be not to be bitten by a snake. And we would be keep on updating the minimum value for variable 'min'.

Finally we will update t[i] with min and return t[i].

CHAPTER-7

CONCLUSION

We have implemented Breadth first Search algorithm to find the minimum steps to win the Snake and ladder game.

REFERENCES

https://www.geeksforgeeks.org/
https://www.javatpoint.com/daa-dutch-national-flag

https://www.techiedelight.com/min-throws-required-to-win-snake-and-ladder-game/

https://www.codingninjas.com/blog/2020/12/21/understanding-the-snake-and-ladder-problem/

https://stackoverflow.com/questions/65979700/snake-ladder-using-bfs

APPENDIX

CODE

CODE

```
// C++ program to find minimum number of dice throws required to
// reach last cell from first cell of a given snake and ladder
// board
#include<iostream>
#include <queue>
using namespace std;
// An entry in queue used in BFS
struct queueEntry
{
int v; // Vertex number
int dist; // Distance of this vertex from source
};
// This function returns minimum number of dice throws required to
// Reach last cell from 0'th cell in a snake and ladder game.
// move[] is an array of size N where N is no. of cells on board
// If there is no snake or ladder from cell i, then move[i] is -1
// Otherwise move[i] contains cell to which snake or ladder
```

```
at i
// takes to.
int getMinDiceThrows(int move[], int N)
{
// The graph has N vertices. Mark all the vertices as
// not visited
bool *visited = new bool[N];
for (int i = 0; i < N; i++)
visited[i] = false;
// Create a queue for BFS
queue<queueEntry> q;
// Mark the node 0 as visited and enqueue it.
visited[0] = true;
queueEntry s = {0, 0}; // distance of 0't vertex is also 0
q.push(s); // Enqueue 0'th vertex
// Do a BFS starting from vertex at index 0
queueEntry qe; // A queue entry (qe)
while (!q.empty())
{
qe = q.front();
int v = qe.v; // vertex no. of queue entry
// If front vertex is the destination vertex,
// we are done
if (v == N-1)
```

```
break;
// Otherwise dequeue the front vertex and enqueue
// its adjacent vertices (or cell numbers reachable
// through a dice throw)
q.pop();
for (int j=v+1; j<=(v+6) && j<N; ++j)
{
// If this cell is already visited, then ignore
if (!visited[j])
{
// Otherwise calculate its distance and mark it
// as visited
queueEntry a;
a.dist = (qe.dist + 1);
visited[j] = true;
// Check if there a snake or ladder at 'j'
// then tail of snake or top of ladder
// become the adjacent of 'i'
if (move[j] != -1)
a.v = move[j];
else
a.v = j;
q.push(a);
}
```

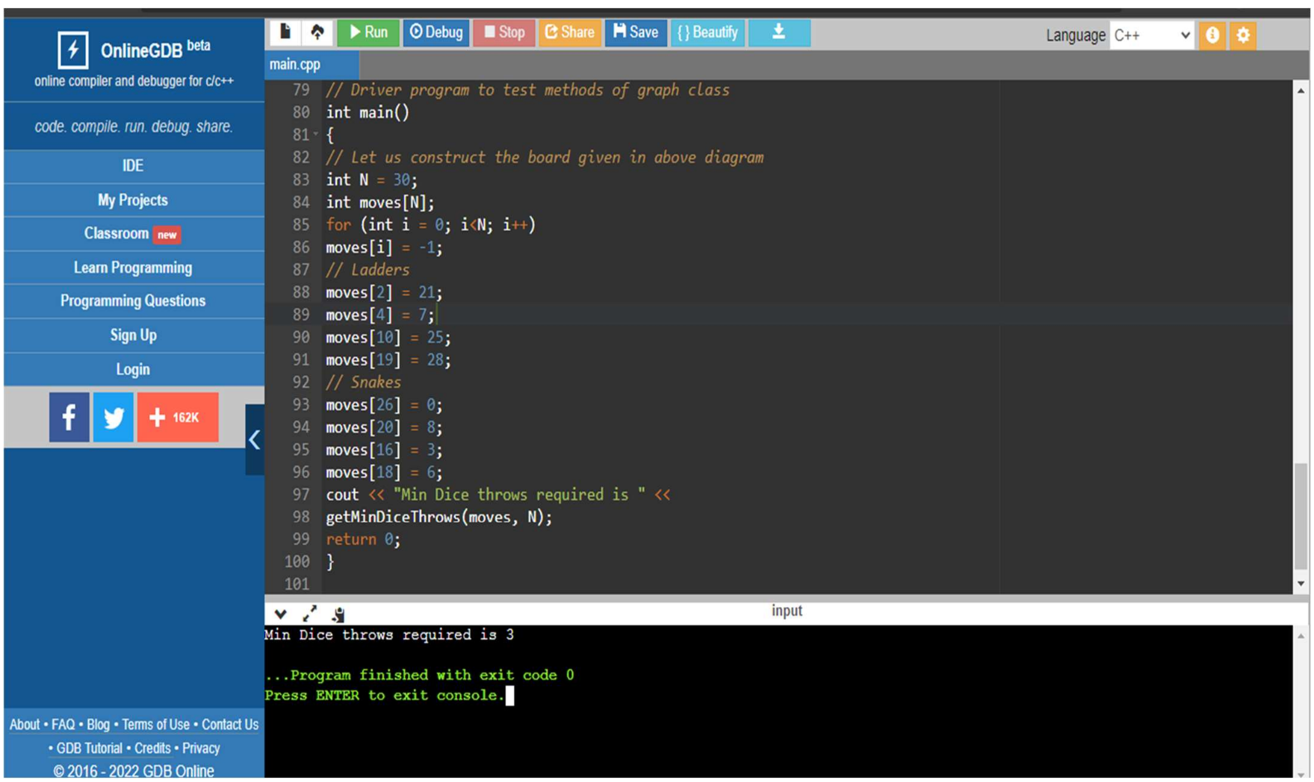
```
}
}
// We reach here when 'qe' has last vertex
// return the distance of vertex in 'qe'
return qe.dist;
}
// Driver program to test methods of graph class
int main()
{
// Let us construct the board given in above diagram
int N = 30;
int moves[N];
for (int i = 0; i<N; i++)
moves[i] = -1;
// Ladders
moves[2] = 21;
moves[4] = 7;
moves[10] = 25;
moves[19] = 28;
// Snakes
moves[26] = 0;
moves[20] = 8;
moves[16] = 3;
moves[18] = 6;
```

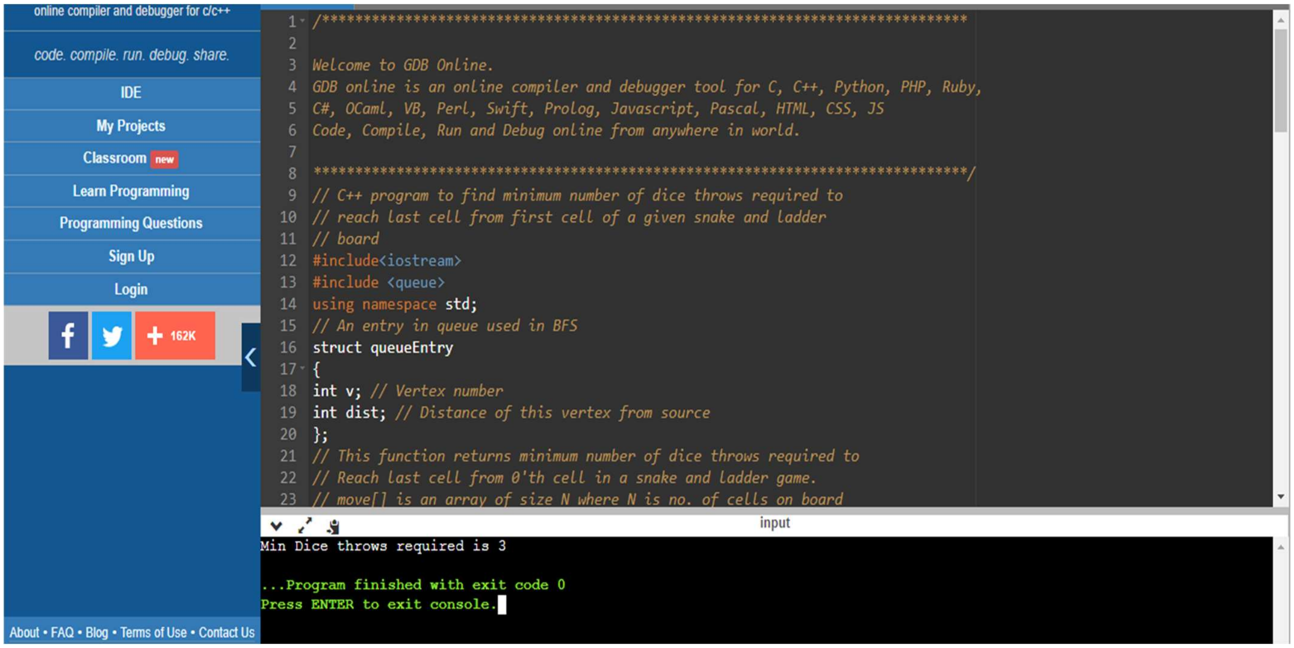
88.cout << "Min Dice throws required is " <<

89.getMinDiceThrows(moves, N);

90.return 0;

91.}




online compiler and debugger for c/c++
code. compile. run. debug. share.
IDE
My Projects
Classroom new
Learn Programming
Programming Questions
Sign Up
Login
162K
About • FAQ • Blog • Terms of Use • Contact Us
/***
Welcome to GDB Online.
GDB online is an online compiler and debugger tool for C, C++, Python, PHP, Ruby,
C#, OCaml, VB, Perl, Swift, Prolog, Javascript, Pascal, HTML, CSS, JS
Code, Compile, Run and Debug online from anywhere in world.
***/
// C++ program to find minimum number of dice throws required to
// reach last cell from first cell of a given snake and ladder
// board
#include<iostream>
#include <queue>
using namespace std;
// An entry in queue used in BFS
struct queueEntry
{
int v; // Vertex number
int dist; // Distance of this vertex from source
};
// This function returns minimum number of dice throws required to
// Reach last cell from 0'th cell in a snake and ladder game.
// move[] is an array of size N where N is no. of cells on board
input
Min Dice throws required is 3
...Program finished with exit code 0
Press ENTER to exit console.