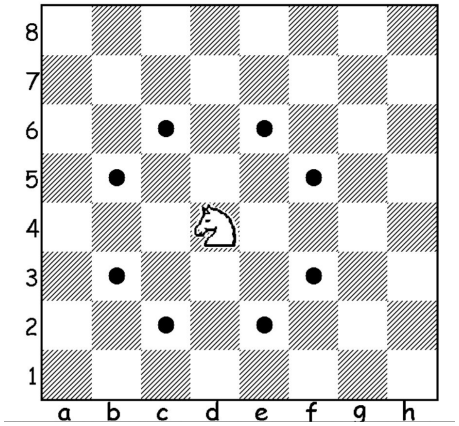
Data Structures Assignment

KNIGHT TRAVAILS

Deepesh Adwani - 210968198
Harsh Der - 210968044
Suzen Firasta - 210968058
Swarnav Mohanta - 210968038
Yash Srivastava - 210968144



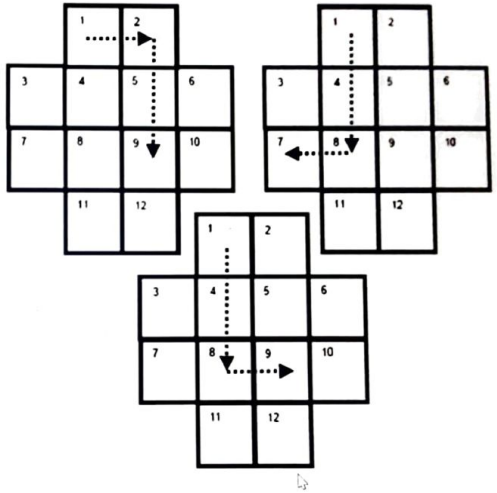
Topic and Objective of the Assignment:

- We were asked to make a project using a Data Structure of our choice.
- Our choice was a small game called **Knights Travels**.
- The objective of this assignment is to find the minimum number of moves from initial to the target position.
- A unique property of the Knight in the game of chess is that it can go to any possible tile (in this case a set of coordinates) given enough moves.
- This can be extended to any number of tiles provided its even.
- Hence we create a 8*8 grid and have the user give inputs to the next coordinate they wish to move. If the user exceeds the minimum number of moves they fail.

The Problem:

- A Gallant and brave knight must traverse the lands to rescue the princess and claim his rightful place in the kingdom.

- A knight on a board has to visit every square without passing through the same square more than once. Then returns to the starting square.



Data structures used:

- We use 2-D arrays to create our chess boards and queues to calculate the shortest path to from our source to the destination.
- The concept also uses Trees but we don’t actually create it as we aren’t showing the user the moves they have to make. The tree is used only to calculate the minimum number of moves that is needed.
- Conceptually we use trees to make use of the **Breadth First Search(BFS)** algorithm, for which queues is needed.

Reasons for using the Data structures we have used:

- Since we have a coordinate based system, we just keep track of the number of moves, and not the distance as the knight doesn't move linearly.
- The idea is to have the root node as the starting node and its children being each of the direction in which the knight can move.
- Since the knight is limited to moving in only certain directions this reduces the maximum possible children to 8. Here we **use the formula |dx| + |dy| = 3.**

Where,

dx = change of coordinates horizontally,

dy = change if coordinates vertically, keeping in mind that we are calculating the Manhattan's Distance and not the conventional distance.

- Now that we have fixed the number of possible moves (keep in mind this isn't a Binary Tree(BT) although some nodes may follow the properties of BT), we can further reduce the children by checking whether or not the coordinate in valid or not.
- Then we perform the Breadth First Search although not completely, and reduce the size of our tree drastically by using simple conditions.

Some additional functions and methods used:

- memeset(*ptr,val,size) : we use this function to initialise our arrays instead of writing explicit for loops this function copies val starting from *ptr till *ptr+size.
- rand()%val : this function returns a random value less than val. We use this function to generate the random start and end coordinates.
- srand(time(null)): this function is a bit specific for our case. Ideally this function is only limited to srand(int seed). Srand makes sure that the sequence of the random digits is unique to the seed value. Essentially each seed value will have a unique random sequence associated with it. The time(null) uses the pc/systems internal clock as the seed value so every time the start and end value will be unique.
- We also make use of pairs(), that groups the coordinates together for easier understanding.
- System(“cls”) just clears the terminal from anything that was present.

Workflow (contd):

- So basically the steps that work behind the scenes are:
  - Getting the Start and End Coordinates
  - Getting the most optimal path from start to finish
  - Setting max steps for the current instance of the game
  - Taking continuous user inputs for the coordinates
  - Continuous check for the users’ validity of inputted coordinates

Setting start and end coordinates:

- First we check we need a way to create the chessboard, for which we use 2-D arrays due to its simplicity.
- Before the user has access to the game 2 things go on behind the scenes:

1)Setting start and end coordinates

2)Calculating the number of moves needed to reach the end.

- The first is accomplished easily by using rand()[explained previously], the second needs some work
- We accomplish this by using BFS and subsequently using 2 more arrays to help .
- Once we have the distance to the target node(in this case the number of moves) we finally create the final chessboard and give the user the option to move wherever the need.

Checking validity of moves:

- To check the validity of the moves we need to use “isvalid” function
- One function is explicitly used for the BFS, one for just checking the coordinates.
- We require 2 functions as the whole point of the BFS is to find the minimum number of moves, so there will never be a case where we will need to go back
- So the first function is completely for the BFS where we also have a boolean array of the same size setting True if we have visited and false if not
- In both these functions the 2nd condition to be checked is the pretty straight forward which just checks whether the coordinates are in or out of the chessboard itself.

The Breadth First Search:

- The algorithm is pretty uncomplicated.
- In a tree we put in the root and on processing it we remove it and add its children in order.
- In this case we are not actually making a tree but we conceptually use a tree.
- Basically in this tree the root will be the tile we are currently at and the children will be all the possible places the user can go. Hence we can already remove most of the leaf nodes in the tree at each level just by checking whether the coordinates are inside the confinement of the board or not.
- The second condition of whether or not the node has been visited is useful for 2 reasons:
    - To have a small tree and not an infinite one
    - To find the steps towards the coordinate.

- So while we’re checking whether or not the coordinate is valid, we have another 2-D array which is useful to map the number of moves
- Basically whenever we establish that a node is valid we increment its value in the distance matrix to be 1 more than its parent.
- Essentially this means that the “distance” is the level of the nodes

The User Interface:

- As for the user interface it isn’t much since its a console based game.
- The user interface is a combination of tab-spaces and newline commands for it to look like it is.
- The user inputs are taken into another 2-D array which serves the purpose of telling the user where it can go.
- This array will be changing after every step. We decided to do this as it makes it easier for us to manage the 2 different arrays instead of 1.
- The first array shown to the user is technically the zeroth step. And after we take the first input we declare it to be as the 1st step.
- So the zeroth step array becomes the static array and every array after the user input is a dynamic one that changes based on the input taken.

- What the user sees once we start the program is:
  - The Start coordinate denoted by [S]
  - The End coordinate denoted by [F]
  - All possible moves from start denoted by [1]
  - All other tiles marked with [x]
- This is the static array only showing the start end and possible steps.
- For subsequent steps we initialise another array which depicts the step number as the next possible steps.
- This helps the user to understand which step number is it and on each step we re-initialise it to the current step number+1
-

Learnings and Outcomes:

- Implementing The Breadth First Search algorithm
- Solving problem statement regarding Shortest path
- Creating a user - friendly console application

Contribution and Sources:

- Deepesh Adwani - Validness of moves
- Harsh Der - Breadth First Search using queue
- Suzen Firasta - Collaboration of the snippets made
- Swarnav Mohanta - Loops and Arrays for traversing
- Yash Srivastava - Collaboration and PPT

Sources -
GeeksForGeeks
CppReference