Knight’s tour report

-Luis Cebrián Chuliá-
Stud.no. 500747256

Overview:

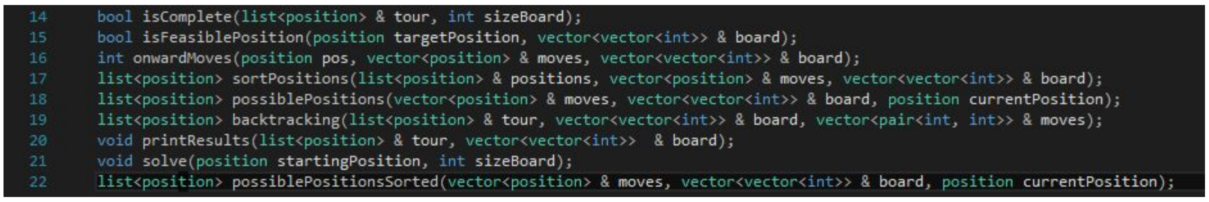
My solution to this problem makes use of backtracking in order to obtain the knight’s tour. It uses a vector of vectors of ints to represent the board and, each cell has a 0 or 1 either it has been visited (1) or not (0). The user can set the board size (but be careful, because it uses recursion, numbers greater than 35-40 can cause overflow), and the starter position as a letter followed by a number (e.g. a0, b3, ….). To represent the solution we used a similar approach as Bill Weinman used in his 9 queens problem. We show a board with a number in each cell. This number represents the path followed by the knight.

1-Requirements

-Programming language has to be C++
c++ is the language used to solve this problem.

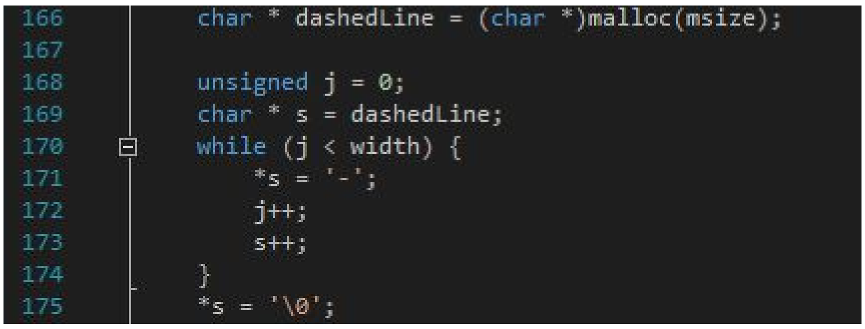
-The program makes use of pointers and references

Because we didn’t use an object-oriented approach, most of the functions receive as parameters large objects(in order to maintain the scope the functions need of these parameters) such as lists, vectors, and because it uses backtracking, the management of memory is very important. That’s why all the lists / vectors used are passed as a reference, in order to prevent c++ of making a copy of each parameter in each call of the functions.



```
bool isComplete(list<position> & tour, int sizeBoard);
bool isFeasiblePosition(position targetPosition, vector<vector<int>> & board);
int onwardMoves(position pos, vector<position> & moves, vector<vector<int>> & board);
list<position> sortPositions(list<position> & positions, vector<position> & moves, vector<vector<int>> & board);
list<position> possiblePositions(vector<position> & moves, vector<vector<int>> & board, position currentPosition);
list<position> backtracking(list<position> & tour, vector<vector<int>> & board, vector<pair<int, int>> & moves);
void printResults(list<position> & tour, vector<vector<int>>  & board);
void solve(position startingPosition, int sizeBoard);
list<position> possiblePositionsSorted(vector<position> & moves, vector<vector<int>> & board, position currentPosition);
```

In order to meet the requirement of using pointers, we implemented a few instruction to make use of them on the printResults(...) function. Here, we create a dashed line to print the results graphically.

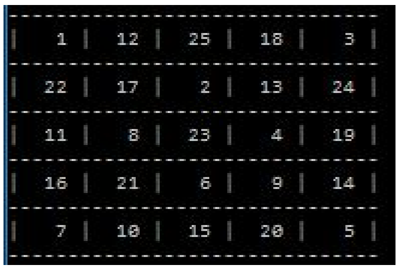


```
    char * dashedLine = (char *)malloc(msize);

    unsigned j = 0;
    char * s = dashedLine;
    while (j < width) {
        *s = '-';
        j++;
        s++;
    }
    *s = '\0';
```

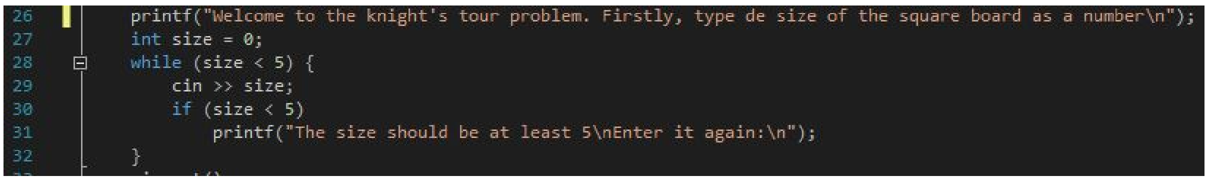
This could have been done easily with two lines of code and a for loop, but then we wouldn't make use of the pointers at all.

-There is a visual representation.
This is the representation used to show the results



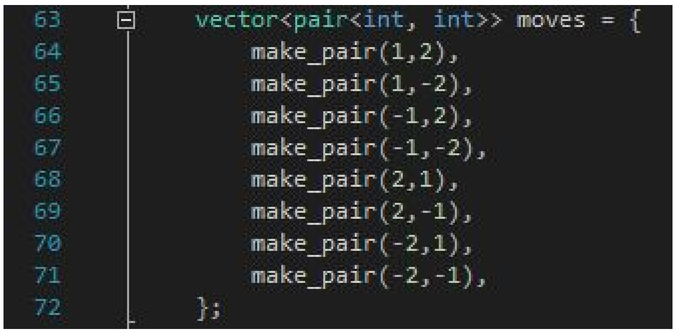
```
------------------------------
|  1 | 12 | 25 | 18 |  3 |
------------------------------
| 22 | 17 |  2 | 13 | 24 |
------------------------------
| 11 |  8 | 23 |  4 | 19 |
------------------------------
| 16 | 21 |  6 |  9 | 14 |
------------------------------
|  7 | 10 | 15 | 20 |  5 |
------------------------------
```

-The board is at least 5x5 in size.
Because the user can set the board's size, we force him to enter a number equal or greater than 5.



```
    printf("Welcome to the knight's tour problem. Firstly, type de size of the square board as a number\n");
    int size = 0;
    while (size < 5) {
        cin >> size;
        if (size < 5)
            printf("The size should be at least 5\nEnter it again:\n");
    }
```

-The knight has to move according to its movement rules.
As seen above, the knight is moving according its movement rules, defined in the program the possible moves as follows:

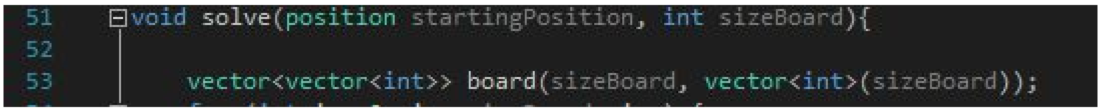


```
    vector<pair<int, int>> moves = {
        make_pair(1,2),
        make_pair(1,-2),
        make_pair(-1,2),
        make_pair(-1,-2),
        make_pair(2,1),
        make_pair(2,-1),
        make_pair(-2,1),
        make_pair(-2,-1),
    };
```

-The knight can only visit each square once.
As seen above, the knight only visits each square once. This is done by locking the visited board cell with a 1, and checking before moving there if it is free or not.

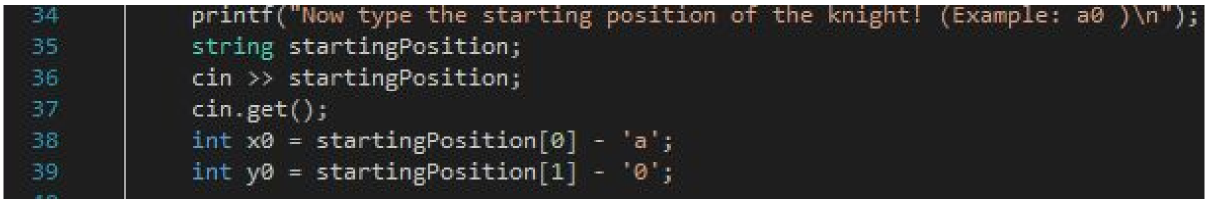
2-Exemplars

-Board size can be set by the user (NxN size).
The program ask the user to enter a number, which will be the size of the board later, by creating a vector of vectors using that number.



```
51  void solve(position startingPosition, int sizeBoard){
52  
53      vector<vector<int>> board(sizeBoard, vector<int>(sizeBoard));
```

-The user can decide where the knight starts.
After entering the size of the board the user is asked to enter the starting position with the pattern: letter followed by number. This string will be translated to an x,y coordinates.



```
    printf("Now type the starting position of the knight! (Example: a0 )\n");
    string startingPosition;
    cin >> startingPosition;
    cin.get();
    int x0 = startingPosition[0] - 'a';
    int y0 = startingPosition[1] - '0';
```

-Uses a more efficient solution than brute force (explain in report required)
We used backtracking to solve this problem. Basically what it does is, from the starting position, search the next position until it finds a solution or the path can't be finished. Then, undo the changes to the point it can continue with a new path until it finds the correct one. We will explain in more detail later in the report.

3-Explanation of the algorithm

(Note: We won't post pictures of the code in order to make the report readable, instead we will put the lines we are referring to)

1-Set up: First, we create the board as a vector of vectors of ints with the size entered by the user. Then we set all the board's values to 0. We will use this values to check if the cell has been visited or not. Then, we create the tour list that will hold the cells visited. After that, we define a vector of moves, to specify the allowed movements the knight can make. Then we call the function backtracking to solve the problem and print the results afterwards.
(function solve(), lines 51-77)

2- Backtracking: The backtracking function operates as follows: first we create a list named result, which will hold the result of the backtracking algorithm. This list will have either 0 elements or the completed tour in order to check if we have found a solution in our recursives calls or not.

Then we check if the tour is completed by calling the function isComplete(...) which will check if the size of the tour is equal to number of moves it has to be. If it is completed, we then return the tour list as a result.

If not completed, we will make a move. First we extract the last element of the tour (the current position of the knight) and calculate the next possible positions with the function possiblePositionsSorted(...) which returns the next possible positions sorted following the Warndorff's rule in order to find quickly a solution. Then we iterate over this positions adding the next position to the list and setting that cell of the board to 1 in order to mark it as visited.

Then we call the function backtracking to make our next move. If our recursive calls have been successful, it will return a list containing either the tour, or an empty list. If we have found a tour, we return the tour, if not, we remove the last move of the tour(because we have not been able to find a tour with that move), unmark that cell on the board and then check the next possible position in the next iteration of the loop.
(function bactracking(..) lines 79-100)

3-Printing the results

To print the results of the algorithm we make use of the board we used previously to mark the visited cells. We iterate over the tour list and then set that position on the board to the value of the counter. As a result of that we now have the board filled with the numbers 1 to N*N which will represent the knight's tour.

Once we have the board filled, we then create a dashed line using malloc() in order to create a dynamic dashed line based on the size of the board. We then fill the dashed line with '-' and then we iterate over the board to print the results nicely.
(function printResults(..) lines 156-184)

4-Auxiliary functions used

The code explained above is the core of the algorithm but, we implemented more functions in order to make code clean and readable. We explain what these functions do:

possiblePositions(...): This function calculates the next possible positions we can move the knight. It iterates over the vector moves and checks if the target position is valid. If it is, adds the position to the result.
(function possiblePositions(...) lines 133-142)

isFeasiblePosition(...): This function checks if the target position is valid given the state of the board. Also checks if the target position is out of bounds (coordinates less than 0 or greater than the size). This function is used by the possiblePositions(...) function.
(function isFeasiblePosition(...) lines 144-154)

sortPositions(...): This function sorts the list of positions passed according to the Warndorff's rule. To do so, iterates over each position and calculates the number of onward moves from that position and makes a pair with that number and the current position in the list. Then we sort the list. Because we are trying to sort a list of pairs, we have to define an anonymous function in which we define how to sort it. In our case, by the first number (the number of onward moves). This function is used by the sortPossiblePositions(...) function.
(function sortPositions(...) lines 110-127)

onwardMoves(...): This function calculates the number of possible moves from a given position just by calling the function possiblePositions and calculating its length. This function is used by the sortPositions(...) function
(function onwardMoves(...) lines 129-131)

4-Implementation details

1- Because we made use of complicated types (e.g. vector<pair<int,int>>) we defined an alias to make the code more readable. Given so, *pair<int,int>* will now be just *position*

2-Functions are placed following the step-down rule, presented in the book Clean code: A Handbook of Agile Software Craftsmanship. This means that every function is followed by those at the next level of abstraction, so that we can read the program, descending one level of abstraction at a time as we read down the list of functions. To do this, we had to put the decorators of the functions at the beginning of the file in order to avoid compilation errors.

3- We make use of the camelCase notation rather than the underscore notation because we think that makes the code more readable.

4- We use meaningful names in functions and variables in order to make code more readable and understandable to someone alien to the code (as explained in the book mentioned above).

5-Conclusion

Because we meet all the requirements, the code is well structured and all the exemplars have been implemented we think we deserve a 10 in this assignment.