University of Lausanne

Advanced Programming
Final Project

Shortest Path On a Chessboard

Author:
Andrea Danesi

Professor:
S. Scheidegger
Teaching Assistants:
Maria Pia Lombardo
Anna Smirnova

Advanced Programming

Andrea Danesi

Spring 2024

Abstract

The following project encapsulates the creation, through the use of Python, of codes aimed at simulating the path of the horse on the chessboard. The horse's objective is to start from a defined starting point and reach a point, also defined, of arrival. The chessboard is enriched by the presence of two obstacles: a rook and a bishop, which, like the horse, must comply with their own rules relating to the game of chess. The code makes use of object-oriented programming principles to model the game pawns and implement efficient algorithms for navigating the chessboard, ensuring that the horse reaches its destination point. The results show that various algorithms can be implemented to realise defined paths in the presence of obstacles.

1 Introduction

Chess is known to be one of the most popular games in the world whose characteristic that makes it one of the most strategic games is the presence of different movement rules for each category of pawns. The horse is known for its L-shaped movement, which also allows it to jump over other pawns. This project exploits this characteristic to implement algorithms in which the horse must traverse a path on the chessboard from a starting point to an end point, avoiding or, if necessary, eating the two obstacles: the rook and the bishop. The main challenge lies in the complexity of the rules of the game of chess, which must also be respected in the realisation of the algorithms, in fact this rule makes the planning of the horse's path an interesting challenge. This design not only models the horse's movement according to its rules, but also integrates the dynamic management of obstacles, allowing the horse to decide on the optimal strategy for reaching its destination. The chessboard is a complex environment where the presence of obstacles makes planning the horse's route an interesting challenge. This project integrates a dynamic management of obstacles allowing the horse to decide an optimal strategy to reach the destination.

1.1 Objective

The goal of this project is to train and implement some of the most efficient graph search algorithms that allows the horse to cross the chessboard while avoiding or catching obstacles with a few moves as possible. The algorithms that I implemented in this project are:

- Breadth-First Search (BFS)
- Depth-First Search (DFS)
- Algorithm A*
- Bidirectional Search (BS)

All of these algorithms focus on finding the shortest path using different methods. Not all are optimal and not all perform the shortest path analysis in the same way.The task is to implement the various algorithms and analyse them to see which one is the most efficient.

1.2 Scope

This type of project has a wide application in the real world. There are various situations in which an analysis, such as the one carried out in this work, might be useful to improve existing systems or create new ones. Some of these are:
Technology and transport companies: Enterprises like Google Maps or Waze may find this type of analysis interesting so to optimise travel routes to reduce road travel time or to optimise fuel consumption. In addition to this, the accuracy of road signs could also be increased, saving time during travel.
Industrial automation companies: They could use these algorithms for path planning of mobile robots within narrow spaces with many obstacles to ensure that the robots can navigate without collisions and efficiently.
Game developers: They could implement path finding algorithms for game characters to move through the game environment.

In other words, this work can be useful for many organisations in the world of technology and robotics, but not only. There are several aspects and analyses that could help optimising processes of movement and movement of characters, people and objects, thus enabling a more efficient work. For example, all companies involved in the distribution or production of products can improve transport routes, both from supplier to factory and from factory to customers. This, to make business cheaper and more efficient.

2 Research Question and the Relevant Literature

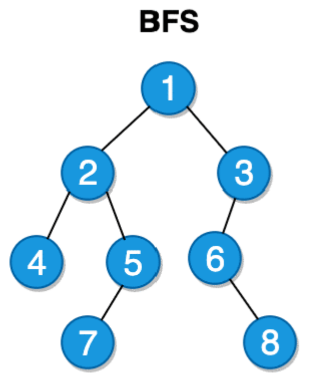
In order to be able to do a more focused and precise analysis, I created a research question that could best encapsulate the essence of this work and thus the research question guiding this study is:
Which path-finding algorithm is the most efficient in terms of the number of movements for moving a horse on a chessboard with obstacles?

This type of research is not new or innovative. There are several articles and researches that have implemented different path search algorithms to find the most efficient. Some only deal with the implementation of an algorithm and comment on its results, like is showed in Techie Delight article [6] for the BFS or in Baeldung [7] for the bidirectional search. These works explain and implement these algorithms and evaluate their performance. For example, on the bidirectional search, it was concluded that the algorithm should find the shortest path in a faster time than the one-way path. However, it is complicated to implement, you have to pay close attention to all the criteria needed to stop the algorithm when the two searches meet [7]. Others, on the other hand, compare the performance and efficiencies of each to determine which is the best, such as the article of Sharma [8]. In this paper, we compare different path search algorithms and analyze the performance of each, to determine which is the best. In this case, the article underlines the efficiency of the algorithm A* which is the best of all [8]. This result will also be confirmed by this study, as you will see later.

3 The Methodology and Algorithm Applied

As anticipated, the four algorithms I implemented are Breath-First Search (BFS), Depth-First Search (DFS), Algorithm A* and Bidirectional Search (BS). These protocols vary from each other in the way they retrieve and calculate the shortest path for the horse. In the following section, it will be explain how these algorithms vary and what their advantages or disadvantages might be.
Let us start with the first algorithm implemented, namely the BFS. It focuses on an exploration of the ramifications level by level, advancing cautiously and sequentially, leaving no option behind.[3]

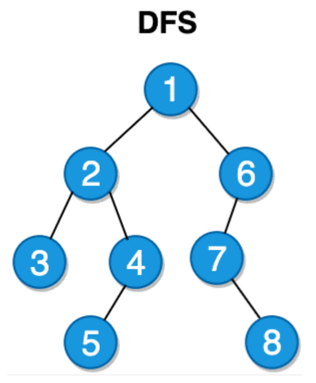
Figure 1: Graph of BFS



We can visualise the situation by looking at the image above, (Figure 1). It can be seen that the image represents nodes numbered according to the order of exploration of the algorithm. This means that the search for the shortest path starts from the starting point and expands to nearby nodes (directly connected to it) and is added to the queue for exploration. Once visited the first neighbors the nodes are removed from the tail and neighbors (nodes connected to the first neighbors) are inserted inside to be added as well. This kind of structure is called FIFO (First-In, First-Out) and this process is repeated until the end point is reached.[3]

The second algorithm implemented within this project is the DFS. Unlike the BFS, this algorithm focuses on digging deeper (as the name suggests) on each branch and then starting from the top. Thus, creating a structure called LIFO (Last-In, Last-Out).[3]

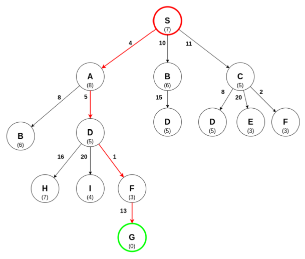
Figure 2: Graph of DFS



In the image above we can see its characteristics and have a clear outlook of how it proceeds to search for the shortest path. Exploring deep, the algorithm uses a stack and not a queue to record the nodes to explore. The operation is similar to the BFS except that instead of exploring by levels it does by ramifications. When it finds a dead end the algorithm goes back and tries another branch and this process repeats it until it finds the arrival.[3]

As far as the A* algorithm is concerned, we can say that it is a combination of the two previous algorithms. We can analyze the image 3 below where it can be seen that the sequence of the exploration of nodes is a mix of BFS and DFS.[4]

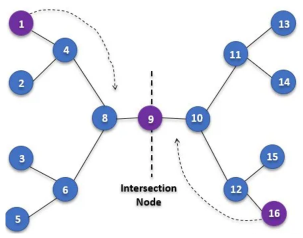
Figure 3: Graph of A*



It start from a point S with a cost g(S) = 0 and an expected cost of h(n) 7 and its total cost for S is f(S)=7 and adds it to the priority queue. Than remove the node with the lowest estimated total cost from the queue, calculate the total cost for all the neighbours and adds them to the queue. It repeat the process until the objective point is reached.
The expected cost is an estimate of the cost from the node to the end point and the algorithm use this information to choose which node and branch to explore first. So the heuristic function that is used to calculate the shortest path combine the cost of moving from a node to another adding the estimate cost. This allow the A* to explore before all the most promising nodes.
In summary, this algorithm explores nodes by balancing width (BFS) and depth (DFS), resulting in an efficient combination of the two algorithms using the heuristic function h(n) to guide the search for the shortest path.[4]

The last algorithm implemented is the BS, which differs the most from the others. As it can see from the image below, the BS has a completely different graph as well as a different structure.[5]

Figure 4: Graph of Bidirectional Search



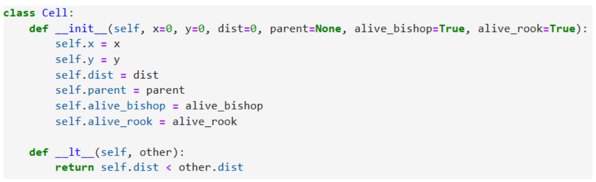
The particularity of this protocol is that it has precisely two search directions, one starting from the start point and the other from the end point. The meeting point is in the middle. This considerably reduces the number of nodes explored and it consequently finds a shorter and more efficient path. So the functioning of this algorithm is explained as follows: it start with a BFS search from the source node and another from the target node. It expand the nodes from the both directions and alternates between the two research. It continue until the two researches converge into a node and this will be our shortest path.[5]

This four algorithms (BFS, DFS, A* and BS) are used to find the shortest path and to explore the graphs in different ways. The first two are more basic algorithm search while the A* is more complex and use an heuristic cost function and the BS is optimizing the search by starting from two side. Each of this algorithm has its strengths and weaknesses and the choice to implement one of them depends of the specific scenario and its issues.

4 Implementation and Performance of the Algorithm

In this section, we discuss the implementation details, including the code used to perform the algorithms. Before starting with the implementation, it is best to get an overview of the situation and some of the functions that prepare the working environment. The first is, for example, the creation of a 'Cell' class, as we can see in the image below.

Figure 5: Class code

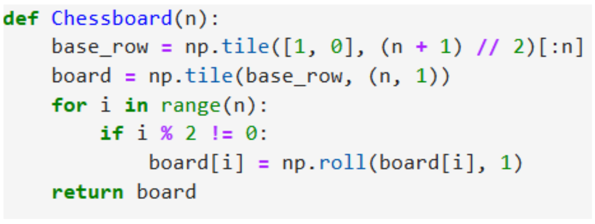


```python
class Cell:
    def __init__(self, x=0, y=0, dist=0, parent=None, alive_bishop=True, alive_rook=True):
        self.x = x
        self.y = y
        self.dist = dist
        self.parent = parent
        self.alive_bishop = alive_bishop
        self.alive_rook = alive_rook

    def __lt__(self, other):
        return self.dist < other.dist
```

It contains the information needed to define each cell within the chessboard such as: its position, dis-

tance from the start, the pointer to the parent node to reconstruct the path and the states of the two obstacles. It also contains a __lt__ method that is used to compare two objects within the class, i.e. two cells, based on distance. It is useful when using priority queues to order the nodes to be explored, like in the case of the A* algorithm.
The second function I used creates the chessboard like we see in the image below.

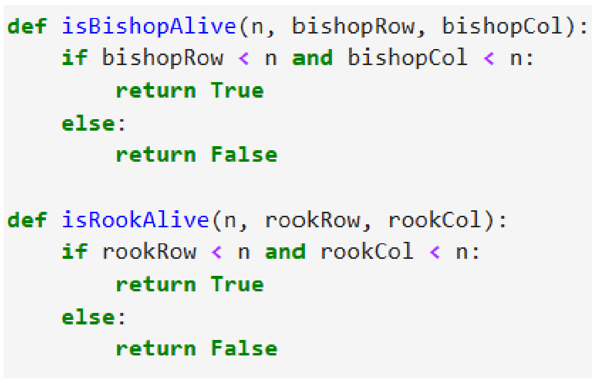
Figure 6: Class code



```python
def Chessboard(n):
    base_row = np.tile([1, 0], (n + 1) // 2)[:n]
    board = np.tile(base_row, (n, 1))
    for i in range(n):
        if i % 2 != 0:
            board[i] = np.roll(board[i], 1)
    return board
```

The first line code "base_row" is gonna creates a repetition of the array '[1, 0]' long enough to cover at least n elements. The number of repetitions is handled by "(n+1)//2" which represents the number of repetitions required to guarantee the length of the array to at least n. Finally "[:n]" breaks the array down to the exact length n thus creating an alternating line of 1 and 0 for the chessboard. The second line will repeat the base line created earlier n times thus obtaining an array $n\ x\ n$. Lastly, the loop created by the code "for i in range(n)" will allow all the elements of the rows with odd index "if i % 2 != 0:" to move to the right by one unit, "np.roll(board[i], 1)", thus creating our classic chessboard.
The last side function I am going to explain is the one concerning the living condition of the obstacles.

Figure 7: Alive condition of obstacles

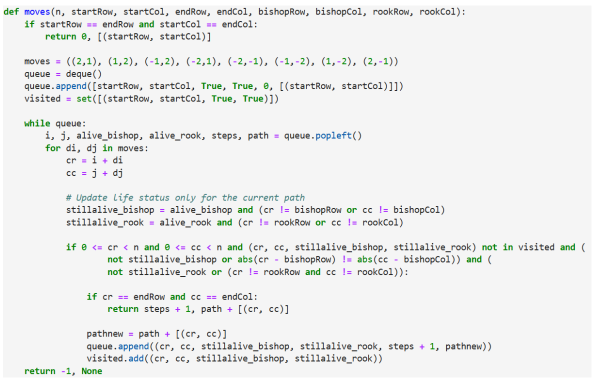


```python
def isBishopAlive(n, bishopRow, bishopCol):
    if bishopRow < n and bishopCol < n:
        return True
    else:
        return False

def isRookAlive(n, rookRow, rookCol):
    if rookRow < n and rookCol < n:
        return True
    else:
        return False
```

These are two rather simple functions that just make sure that the two obstacles are 'alive' i.e. within the chessboard. In fact, the first line only checks that the current row and column of obstacles is smaller than the size of the chessboard so that it returns 'True' if this condition is met or 'False' if it is not. It is not smaller equal to n because the chessboard always starts at 0 with the indices and so if the co-ordinates were equal to n, the obstacles would be outside it.
There are other functions to display and just one to control the co-ordinates of the pieces within the chessboard. For reasons of space, I will not paste the code images but you can find all these functions in the python document and you can see the output in the section of results. They only focus on the display of the pieces, the cells covered by obstacles, the path and the start and end cells.

I will now go on to analyse the codes concerning the algorithms. Let start with the first to be implemented, the Breath-First Search, that we can see in the (Figure 8). This function is called

Figure 8: BFS code



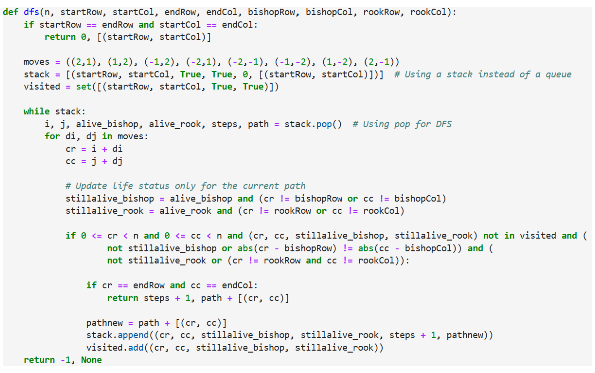
```python
def moves(n, startRow, startCol, endRow, endCol, bishopRow, bishopCol, rookRow, rookCol):
    if startRow == endRow and startCol == endCol:
        return 0, [(startRow, startCol)]

    moves = ((2,1), (1,2), (-1,2), (-2,1), (-2,-1), (-1,-2), (1,-2), (2,-1))
    queue = deque()
    queue.append([startRow, startCol, True, True, 0, [(startRow, startCol)]])
    visited = set([(startRow, startCol, True, True)])

    while queue:
        i, j, alive_bishop, alive_rook, steps, path = queue.popleft()
        for di, dj in moves:
            cr = i + di
            cc = j + dj

            # Update life status only for the current path
            stillalive_bishop = alive_bishop and (cr != bishopRow or cc != bishopCol)
            stillalive_rook = alive_rook and (cr != rookRow or cc != rookCol)

            if 0 <= cr < n and 0 <= cc < n and (cr, cc, stillalive_bishop, stillalive_rook) not in visited and (
                    not stillalive_bishop or abs(cr - bishopRow) != abs(cc - bishopCol)) and (
                    not stillalive_rook or (cr != rookRow and cc != rookCol)):

                if cr == endRow and cc == endCol:
                    return steps + 1, path + [(cr, cc)]

                pathnew = path + [(cr, cc)]
                queue.append((cr, cc, stillalive_bishop, stillalive_rook, steps + 1, pathnew))
                visited.add((cr, cc, stillalive_bishop, stillalive_rook))
    return -1, None
```

'moves' and it is responsible to find the shortest path, in terms of moves, for the horse to go from a start point to an end point. It starts with the definition of the base case, i.e. if the starting point and the finishing point coincide, the function returns 0 steps. We continue with the definition of some variables essential to the algorithm such as the possible movements of the horse on a chessboard with the variable moves, a queue containing information on the position of the horse, the state of the obstacles, the number of moves made and the route taken so far. The visited set is used to keep track of the cells already visited so that loops can be avoided. Following this, we find the main loop of code that takes as its reference the queue explained earlier which uses a FIFO (First-In, First-Out) structure, containing various information regarding the state of the horse and the obstacles. Each first element within the queue is extracted and removed thanks to the *.popleft()* function and then an important loop comes into play that calculates the new coordinates (cr = current row, cc = current column) for each possible move of the horse. We then update the state of the

obstacles by checking two conditions to verify that they are still alive, the variable alive_bishop/rook and the condition that the horse must not have the same coordinates as the two objects otherwise it would mean that it has eaten them.
Now it comes the most challenging part, the one in which I targeted the path of the horse towards its goal according to the position of the obstacles. This part of the code is managed by a long *if* function that contains all the necessary conditions to allow the horse to advance and explore the new cell. These conditions must all be fulfilled simultaneously and in fact they are linked together by *and* so to take into account all the aspects necessary to proceed. If only one is not satisfied it stops. The requirements are: the persistence of the current position within the board, that the current position has not already been visited, that the bishop is not yet alive or that the horse is not on one of the cells on the diagonals of the bishop and the same thing for the tower with its lines. If all these conditions are met the code performs three operations: it creates a new path with the new coordinates of the horse, adds the new status to the queue for further explorations and also adds it to the visited set to avoid cycles. If, instead, the new position of the horse should be the arrival the code will return the number of movements performed up to them and the coordinates performed to reach the goal. Finally, if the algorithm does not find a valid path it will return that there are no possible movements to do.

Moving on to the second algorithm used, the Depth-First Search. In the image below we can see the implementation code of the latter.

Figure 9: DFS code

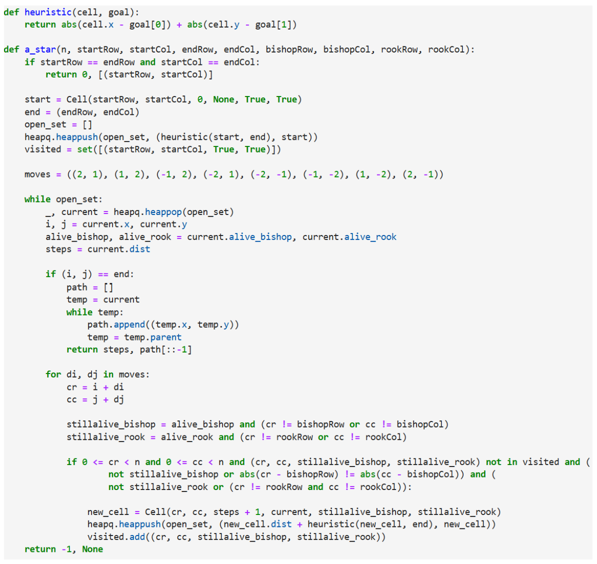


```
def dfs(n, startRow, startCol, endRow, endCol, bishopRow, bishopCol, rookRow, rookCol):
    if startRow == endRow and startCol == endCol:
        return 0, [(startRow, startCol)]

    moves = ((2,1), (1,2), (-1,2), (-2,1), (-2,-1), (-1,-2), (1,-2), (2,-1))
    stack = [(startRow, startCol, True, True, 0, [(startRow, startCol)])]  # Using a stack instead of a queue
    visited = set([(startRow, startCol, True, True)])

    while stack:
        i, j, alive_bishop, alive_rook, steps, path = stack.pop()  # Using pop for DFS
        for di, dj in moves:
            cr = i + di
            cc = j + dj

            # Update life status only for the current path
            stillalive_bishop = alive_bishop and (cr != bishopRow or cc != bishopCol)
            stillalive_rook = alive_rook and (cr != rookRow or cc != rookCol)

            if 0 <= cr < n and 0 <= cc < n and (cr, cc, stillalive_bishop, stillalive_rook) not in visited and (
                    not stillalive_bishop or abs(cr - bishopRow) != abs(cc - bishopCol)) and (
                    not stillalive_rook or (cr != rookRow and cc != rookCol)):

                if cr == endRow and cc == endCol:
                    return steps + 1, path + [(cr, cc)]

                pathnew = path + [(cr, cc)]
                stack.append((cr, cc, stillalive_bishop, stillalive_rook, steps + 1, pathnew))
                visited.add((cr, cc, stillalive_bishop, stillalive_rook))
    return -1, None
```

We can see that the structure of the code, mirrors that of the BFS but we know that the difference between these two algorithms is given only by the way in which the nodes (cells) are explored. In fact we can see that instead of using a *queue* it uses a *stack*. We notice it in the first part of the code and from the commands *stack.append* to add elements to the stack and from *stack.pop* to remove them following the order LIFO (Last-In, Last-Out). The information contained is always the same only changes the way and the order in which they are selected.

As for the algorithm A* we immediately notice from the image, that the code already has a structure and the steps slightly different from the previous two.

Figure 10: A code

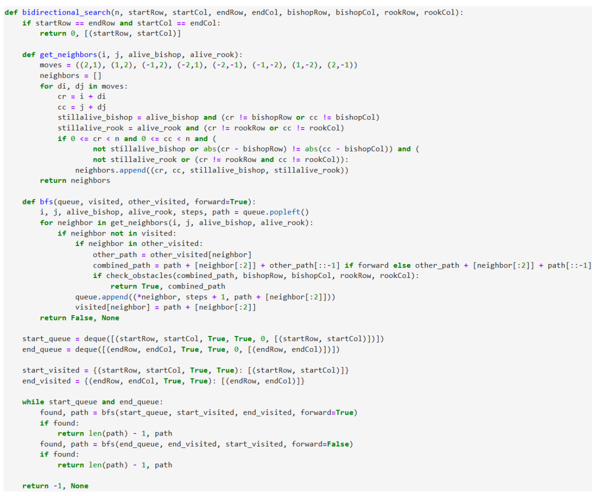


```
def heuristic(cell, goal):
    return abs(cell.x - goal[0]) + abs(cell.y - goal[1])

def a_star(n, startRow, startCol, endRow, endCol, bishopRow, bishopCol, rookRow, rookCol):
    if startRow == endRow and startCol == endCol:
        return 0, [(startRow, startCol)]

    start = Cell(startRow, startCol, 0, None, True, True)
    end = (endRow, endCol)
    open_set = []
    heapq.heappush(open_set, (heuristic(start, end), start))
    visited = set([(startRow, startCol, True, True)])

    moves = ((2, 1), (1, 2), (-1, 2), (-2, 1), (-2, -1), (-1, -2), (1, -2), (2, -1))

    while open_set:
        _, current = heapq.heappop(open_set)
        i, j = current.x, current.y
        alive_bishop, alive_rook = current.alive_bishop, current.alive_rook
        steps = current.dist

        if (i, j) == end:
            path = []
            temp = current
            while temp:
                path.append((temp.x, temp.y))
                temp = temp.parent
            return steps, path[::-1]

        for di, dj in moves:
            cr = i + di
            cc = j + dj

            stillalive_bishop = alive_bishop and (cr != bishopRow or cc != bishopCol)
            stillalive_rook = alive_rook and (cr != rookRow or cc != rookCol)

            if 0 <= cr < n and 0 <= cc < n and (cr, cc, stillalive_bishop, stillalive_rook) not in visited and (
                    not stillalive_bishop or abs(cr - bishopRow) != abs(cc - bishopCol)) and (
                    not stillalive_rook or (cr != rookRow and cc != rookCol)):

                new_cell = Cell(cr, cc, steps + 1, current, stillalive_bishop, stillalive_rook)
                heapq.heappush(open_set, (new_cell.dist + heuristic(new_cell, end), new_cell))
                visited.add((cr, cc, stillalive_bishop, stillalive_rook))
    return -1, None
```

The first difference we see in the formulation of the variables *start* and *end* with the coordinates of departure and arrival of the horse. These variables are then used to calculate priorities (lower cost) with the function *heuristic*. In this case we use a priority list to store the horse conditions ordered by estimated cost. To manage this priority list we use *heapq* adding information with . *heappush* such as: the priority (current cost + heuristic), the current cost (number of moves), the current coordinates of the horse, the living conditions of the obstacles and the path followed to that point. After setting these starting variables, we find the main loop of this algorithm. The difference from the other two algorithms is that we find the function *heapq.heappop*(open_set) to manage the priority list by removing and returning the element with the highest priority. The last differences are at the end of the loop where we define the variable *cost* to keep track of the number of moves made up to that point and the variable priority to estimate the remaining distance to the destination (sum of the current cost and heuristic). Finally, the function *heapq.heappush* adds the new state to the heap (open_set) and the loop continues until the latter variable is empty.

Let's now pass to the last implemented algorithm, the Bidirectional Search. Compared to the other three this code is definitely the longest, as we see in the figure below, because we have to manage the search from two different parts.

Figure 11: BS code



```python
def bidirectional_search(n, startRow, startCol, endRow, endCol, bishopRow, bishopCol, rookRow, rookCol):
    if startRow == endRow and startCol == endCol:
        return 0, [(startRow, startCol)]

    def get_neighbors(i, j, alive_bishop, alive_rook):
        moves = ((2,1), (1,2), (-1,2), (-2,1), (-2,-1), (-1,-2), (1,-2), (2,-1))
        neighbors = []
        for di, dj in moves:
            cr = i + di
            cc = j + dj
            stillalive_bishop = alive_bishop and (cr != bishopRow or cc != bishopCol)
            stillalive_rook = alive_rook and (cr != rookRow or cc != rookCol)
            if 0 <= cr < n and 0 <= cc < n and (
                    not stillalive_bishop or abs(cr - bishopRow) != abs(cc - bishopCol)) and (
                    not stillalive_rook or (cr != rookRow and cc != rookCol)):
                neighbors.append((cr, cc, stillalive_bishop, stillalive_rook))
        return neighbors

    def bfs(queue, visited, other_visited, forward=True):
        i, j, alive_bishop, alive_rook, steps, path = queue.popleft()
        for neighbor in get_neighbors(i, j, alive_bishop, alive_rook):
            if neighbor not in visited:
                if neighbor in other_visited:
                    other_path = other_visited[neighbor]
                    combined_path = path + [neighbor[:2]] + other_path[::-1] if forward else other_path + [neighbor[:2]] + path[::-1]
                    if check_obstacles(combined_path, bishopRow, bishopCol, rookRow, rookCol):
                        return True, combined_path
                queue.append((*neighbor, steps + 1, path + [neighbor[:2]]))
                visited[neighbor] = path + [neighbor[:2]]
        return False, None

    start_queue = deque([(startRow, startCol, True, True, 0, [(startRow, startCol)])])
    end_queue = deque([(endRow, endCol, True, True, 0, [(endRow, endCol)])])

    start_visited = {(startRow, startCol, True, True): [(startRow, startCol)]}
    end_visited = {(endRow, endCol, True, True): [(endRow, endCol)]}

    while start_queue and end_queue:
        found, path = bfs(start_queue, start_visited, end_visited, forward=True)
        if found:
            return len(path) - 1, path
        found, path = bfs(end_queue, end_visited, start_visited, forward=False)
        if found:
            return len(path) - 1, path

    return -1, None
```

This code partially takes over the BFS code but is a bit complicated by the double search direction. The beginning is the same as the others, also the sub-function named *get_neighbors* returns a list of tuples with the close valid cells that the horse can reach starting from the position *(i, j)*. The following function, called *bfs*, performs the steps of the BFS algorithm from one of the two directions (initial or final). Try to expand the current node by adding new nodes to the queue and checking if there is a meeting point between the search in the two directions. It creates a *for* loop in which all possible neighbors that have been defined with the first explained function are analyzed. For every possible neighbor, he makes sure that he hasn't already been checked in either direction. If this cell has already been checked in the other direction, the code will combine the two paths and verify that it is valid (not threatened by obstacles). As queues and visited lists are updated with new paths and cells visited from both directions. Finally, the algorithm alternates the search from one direction to another by specifying the variable *forward=True* for the direction from the start and *forward=False* for the direction from the end.
It remains to explain a function that is called within the code *bfs* or the function *check_obstacles* that is defined outside the main one. The latter is responsible for checking whether it is necessary or not to eat the two obstacles. To do so, start checking that the two objects are still alive or have not been eaten. Then it defines the variables that define whether you need to eat them or not *need_eat_bishop/Rook* checking that none of the cells in the horse's path are either on the diagonals covered by the bishop or on the lines covered by the tower. Thanks to all these functions I was able to apply the bidirectional algorithm efficiently.

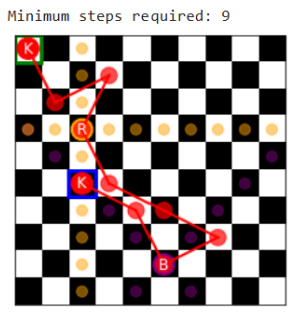
5 Maintenance and updating the Codebase

During a project it is always very important to make backups in order not to risk losing all the work done. For the coding part of this project I used the Jupyter application via the "Nuvolos" platform and saved my files as I advanced on it. This way, even if my computer broke, I could always access it through other computer devices. Sometimes I also saved the documents on my personal computer drive and on my GitHub repository so that I could have more copies in different places accessible from other devices. As for the paper instead, I used latex on the overleaf site that allowed me to always have a copy automatically saved of my document on a website also accessible from different devices. So despite being an individual project I used methods of maintaining and updating my basic code and my paper that were also accessible to other people so that I was sure not to lose all the work done.

6 Results

In this section we will analyse the results of all the algorithms we have implemented and try to understand the various differences. All output images you will see refer to the same start, finish and obstacle positions so they can be compared in a more efficient manner. This part of the output is managed by a configuration code for the coordinates of all the necessary elements on the board, but these can be changed at pleasure. In this case, I decided to set the destination on a cell covered by both obstacles so as to see how the different algorithms behave.
Let us begin by analysing the first output that we see in the image below.

Figure 12: BFS path

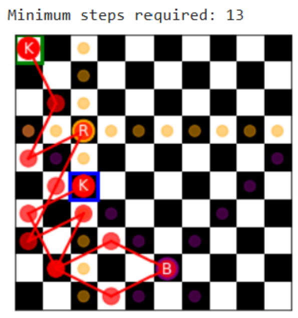


```
Minimum steps required: 9
```

We can first note that the algorithm behaves correctly because before going to the end point, it realises that it has to free the target by 'eating' the two obstacles that cover it. So starting from the green square, the horse moves in the direction of the rook to pass over it and "eat" it and then continues in the direction of the bishop to do the same. Finally, after clearing the end point, the horse can reach it calmly, all following the shortest path. We can also see that the number of steps required by the algorithm to make this path is 9, and by analysing the other outputs, we will see that this is the lowest and most efficient number.

With regard to the second implemented algorithm, we can see that the calculated path is different and longer than the first.

Figure 13: DFS path

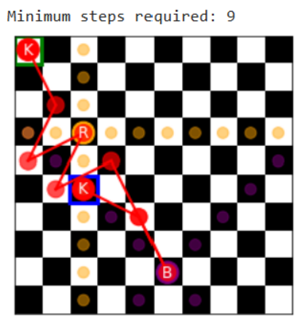


```
Minimum steps required: 13
```

One can see quite clearly the concept of depth that the DFS uses, as the path follows the first explored branch all the way down before turning back, following a more vertical line than that of the BFS. However, the sequence of destinations turns out to be the same, thus satisfying the preconditions that avoid the threatened cells and free the target first.

Switching to the third output: algorithm A*.

Figure 14: A* path

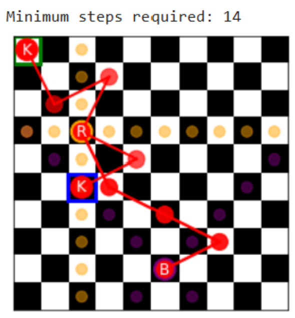


```
Minimum steps required: 9
```

Once again we can see that the path in red is not the same as the previous ones suggesting that A* uses another technique to calculate it. As we saw in the implementation section, the algorithm uses a cost calculation with a heuristic function by first exploring nodes that have a lower cost estimate to reach the goal. In this case the cost is represented by the number of moves and then first explore the cells that according to him would lead to a path with the least possible number of moves. In fact, we see that it is very efficient and it finds the path, always respecting the conditions of the obstacles, in only 9 moves.

Finally, we have the last output of the last implemented algorithm, the Bidirectional Search.

Figure 15: BS path



```
Minimum steps required: 14
```

Also this time we notice that the route is different from all those previously seen. You can clearly see the alternating search in the two directions because the path that starts from the end point goes towards the tower to "take" it and then both directions go towards the bishop. In this case the search is not very efficient probably because of the location of the obstacles to capture and the destination that forces the algorithm to follow a path not optimal with a number of moves equal to 14.

7 Conclusion

The project effectively demonstrates the application and performance of the various path search algorithms through quantitative analysis (number of steps and path followed) and visuals that allow us to answer our research question by comparing the effectiveness of these protocols. We can conclude that this project demonstrated the strengths and limitations of each algorithm inside a specific and controlled environment. A* and BFS appear to be as the most efficient for shortest path-finding, with A* offering a better balance between an optimal solution and computational efficiency. Obviously these are considerations about this case and this specific project, the choice of algorithms to be implemented depends so much on the goal of the work and the situation in which the analysis is being done. For future work you could think about applying other path search algorithms and compare them with those used in this study. You could also try hybrid approaches with the bidirectional algorithm and see if it becomes more efficient within similar scenarios.

Appendix

A List of Helper-Tools

I used Chat-GPT to compile the codes and for some detailed explanation of the algorithms. It was used as support for my project and not as a basis. When it helped me I always tried to understand in full and in every detail what it provided to me to see if it could be appropriate to my project.[1]

I also used an online dictionary to translate some words into English so that I could have a more correct and fluid text.[2]

References

[1] OpenAI. (2024). ChatGPT: AI Language Model. Retrieved from https://www.openai.com/chatgpt

[2] WordReference. (n.d.). Dizionario Online. Retrieved from https://www.wordreference.com/it/

[3] Kustpicx. (2021, June). Graph of BFS and DFS. Retrieved from https://kustpicx.blogspot.com/2021/06/compare-bfs-and-dfs-12-bfs-vs-dfs.html

[4] Codecademy. (2023, April). Graph of A* Algorithm. Retrieved from https://www.codecademy.com/resources/docs/ai/search-algorithms/a-star-search

[5] EDUCBA. (2023, March). Graph of Bidirectional Search Algorithm. Retrieved from https://www.educba.com/uninformed-search/

[6] Techie Delight. (2022). Chess Knight Problem – Find Shortest Path from Source to Destination. Retrieved from https://www.techiedelight.com/chess-knight-problem-find-shortest-path-source-destination/

[7] Baeldung. (2024, March). Bidirectional Search. Retrieved from https://www.baeldung.com/cs/bidirectional-search

[8] Sharma, S., & Kaur, P. (2018, June). Comparative Analysis of Search Algorithms. Retrieved from https://www.researchgate.net/publication/333262471-Comparative-Analysis-of-Search-Algorithms