CIS 579: Artificial Intelligence

Assignment #1

Comparing A* Search Algorithm to Branch and Bound Search

Nicholas Butzke

May 27, 2023

1 Introduction

This paper compares the effectiveness of the A* search algorithm to that of traditional branch and bound search in the context of a simplified version of the Four Knights Puzzle. Heuristic values will be calculated mathematically for estimations of steps to completion.

2 Problem Description

Write a program that allows compares the effectiveness of the A* search algorithm to that of traditional Branch and Bound search.

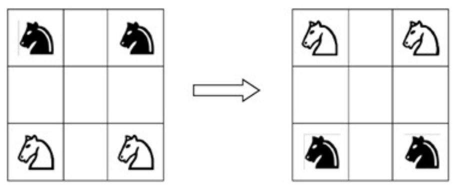


Figure 1: Start to Goal board state

The Four Knights puzzle is played on a 3 by 3 chess board. As depicted in *Figure 1*, the opposing knights (two white and two black) begin in opposite corners. The goal of this puzzle is to have the opposing knights switch sides of the three-by-three chess board. The knights are moved one at a time, using the standard movement rule (two squares ahead in any direction, then one square left or right).

3 Data Structure

The environment is represented by a chessboard with two white knights (W) and two black knights (B). The initial state of the chessboard is as follows:

$$\begin{matrix} B & . & B \\ . & . & . \\ W & . & W \end{matrix}$$

The goal is to find the shortest path from the initial state to a goal state. The goal state is represented by the following configuration of the chessboard:

$$\begin{matrix} W & . & W \\ . & . & . \\ B & . & B \end{matrix}$$

These environments, or board states, are stored in a class, `ChessBoard`, with attributes to track who's turn it is and lists of coordinate positions of each knight. These `ChessBoard` objects are nested into a `Node` class.

The `Node` class facilitates the relationship between `ChessBoard` objects. The attributes in the `Node` include a `ChessBoard` object for the board, scoring values to hold costs of the `Node` in relation to the tree, a `parent` `Node` object, and a list of `children` `Node` objects. Methods in this class are to generate valid child `Node`s and calculate scoring.

4 Algorithm Implementations

4.1 A* Implementation

The full implementation of the A* search algorithm can be found in the accompanying `.py` file. The breakdown of how the A* algorithm is implemented is as follows:

```
open_list: list[Node] = [Node()]
closed_list: list[Node] = []
```

These two lists are to act as a queue of `Node`s to evaluate and a list of nodes that have previously been evaluated. Following this is the main `while` loop to build and iterate over the tree.

```
while open_list:
    currentNode = min(open_list, key=lambda node: node.f_score)
    open_list.remove(currentNode)
    currentNode.make_children()
    for child in currentNode.children:
```

The section above begins and continues the `while` loop so long as the queue has a `Node` in it. The algorithm will then find the `Node` with the smallest `f` value, remove it from the queue, and generate it's child `Node`s. After which, the algorithm will begin iterating over those children performing 2 main tasks:

```
if child.board.boardState == goalState.board.boardState:
    optimalPath = [goalState]
    while currentNode is not None:
        optimalPath.append(currentNode)
        currentNode = currentNode.parent
    return optimalPath[::-1]
```

The first task is to check if the `child`'s `boardState` is equivalent to the goal `boardState`. If it is then the algorithm will walk back up the tree by logging the `Node` into a list and stepping to each `parent` until it reaches the initial `Node` with no `parent`. Returning the inverted list will return a list ordered from initial `boardState` to goal `boardState`.

```
child.calc_heuristic()
i = find(child, open_list)
if not (i != -1 and open_list[i].f_score < child.f_score):
    i = find(child, closed_list)
    if not (i != -1 and closed_list[i].f_score < child.f_score):
        open_list.append(child)
```

The second task will occur if the `child`'s `boardState` was not equivalent to the goal `boardState`. In this event the algorithm will then calculate the `child`'s heuristic scores and compare it to the `Node`s in the queue and the list of `Node`s that have been evaluated. If it is found to be a novel `Node` or a known `Node` with a lower cost than an evaluated equivalent `Node` then it will be added to the queue of `Node`s to evaluate.

```
closed_list.append(currentNode)
```

After the `for` loop of the children this final line within the `while` loop takes the `currentNode` and adds it to the list of `Node`s that have been evaluated. If the algorithm exits the `while` loop without returning a discovered path it will return a string stating that no path was found.

4.2 Branch and Bound Implementation

The full implementation of the Branch and Bound search algorithm can be found in the accompanying `.py` file. The breakdown of how the Branch and Bound algorithm is implemented is as follows:

```
open_list: list[Node] = [Node()]
closed_list: list[Node] = []
shortestPath = []
shortestPathLength = float('inf')
```

The first two lists are to act as a queue of `Nodes` to evaluate and a list of nodes that have previously been evaluated. The next two lists are to store the shortest path that has been discovered and it's length. These are separate to initialize the length as infinitely high and use it as a bound for the best path. Following this is the main `while` loop to build and iterate over the tree.

```
while open_list:
    currentNode = open_list.pop(0)
```

The section above begins and continues the `while` loop so long as the queue has a `Node` in it. The algorithm will then `pop` the `Node` at the front of the queue. The algorithm will perform 2 main tasks on the popped `Node`:

```
if currentNode == goalState:
path = []
while currentNode is not None:
    path.append(currentNode)
    currentNode = currentNode.parent
if len(path) <= shortestPathLength:
    shortestPathLength = len(path)
shortestPath = path[::-1]
```

The first task is to check if the `currentNode`'s `boardState` is equivalent to the goal `boardState`. If it is then the algorithm will walk back up the tree by logging the `Node` into a list and stepping to each `parent` until it reaches the initial `Node` with no `parent`. Storing the inverted list will store a list ordered from initial `boardState` to goal `boardState` in the `shortestPath` list.

```
else:
    if currentNode.g_score + 1 < shortestPathLength:
        currentNode.make_children()
        for child in currentNode.children:
            i = find(child, open_list)
            if i == -1 and child.g_score < shortestPathLength:
                i = find(child, closed_list)
                if i == -1:
                    open_list = [child] + open_list
```

The second task will occur if the `currentNode`'s `boardState` was not equivalent to the goal `boardState`. In this event the algorithm will then compare the `currentNode`'s spent cost (`g`) to the `shortestPathLength`. If the current cost of arriving at this `Node + 1` is greater than the `shortestPathLength` it can be determined to be a worse path and therefore ignored. If the current cost of arriving at this `Node + 1` is less than the `shortestPathLength` we will generate its child `Node`s. After which, the algorithm will begin iterating over those children to evaluate if they must be added to the queue. If the `child` is found to be a novel `Node` or a known `Node` with a lower cost than the shortest path then it will be added to the front queue of `Node`s to evaluate.

```
if open_list:
    mNode = min(open_list, key=lambda node: node.g_score)
    if mNode != open_list[0]:
        open_list.remove(mNode)
        open_list = [mNode] + open_list
closed_list.append(currentNode)
```

After the `for` of the children we check if the recent `pop` emptied the queue. If there are still `Node`s in the queue then find the one with the lowest spent cost (`g`) and move it to the front of the queue to be evaluated next. Finding the minimum node is faster than sorting the entire queue each time. The final line within the `while` loop takes the `currentNode` and adds it to the list of `Node`s that have been evaluated.

```
if shortestPath:
    return shortestPath
else:
    return "no path found"
```

When the algorithm exits the while loop if a path was found it will be stored in the `shortestPath` list and thus returned. If the `shortestPath` list is empty then no path was found and a string stating that no path was found will be returned.

5 Heuristic Calculation

The full implementation of the `calc_heuristic()` function can be found in the accompanying `.py` file. The breakdown of how the heuristics are calculated are as follows:

```
d1 = abs(whiteKnight[0]-0) + abs(whiteKnight[1]-0)
d2 = abs(whiteKnight[0]-0) + abs(whiteKnight[1]-2)
```

First, the distance from a knight to each destination is calculated.

```
if d1 == 0:
    d1 = 4
elif d1 == 4:
    d1 = 2
if d2 == 0:
    d2 = 4
elif d2 == 4:
    d2 = 2
```

The distances are adjusted for edge cases to allow the calculation to function properly.

```
whiteH += abs(d1 - 4) + abs(d2 - 4)
```

The heuristic calculation takes the inverse of the distances by subtracting 4 to best predict the cost of arrival at a destination. This calculation is performed for each knight. The example given was for the White knights however similar actions are taken for the Black knights.

```
self.h_score = (whiteH + blackH)*2
self.f_score = self.g_score + self.h_score
```

After the heuristic for each side is calculated, it is then summed with the other side, and finally multiplied. This is done to more greatly prioritize nodes that have been visited over exploring novel nodes. Then each score is update for the given `Node`.

6 Results

Both algorithms found an optimal path of 16 moves however the runtimes between the two algorithms were significantly different.

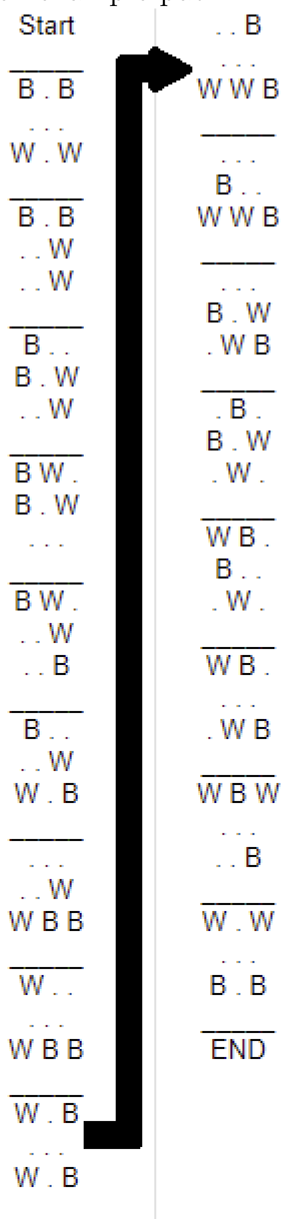
A*	**Branch and Bound**
3ms	12ms
3ms	11ms
4ms	11ms
3ms	11ms
4ms	11ms
3ms	12ms
3ms	12ms
3ms	12ms
4ms	12ms
4ms	11ms

The A* search algorithm ran for an average of 3.4ms over 10 runs.

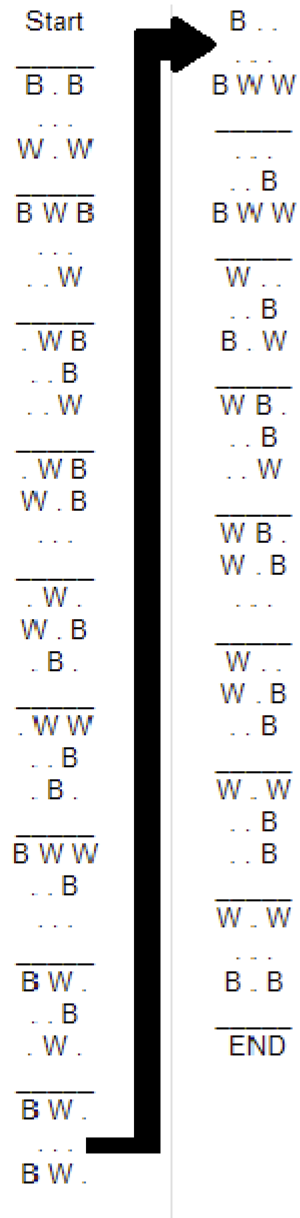
The Branch and Bound Algorithm ran for an average of 11.5ms over 10 runs.

A* search took 29.57% of the runtime of Branch and Bound search to find an equally optimal solution.

A* search example path:



Branch and Bound search example path:



7 Conclusion

While the A* search algorithm and the Branch and Bound search algorithm both found equally optimal paths from the initial state to the goal state, A* search completed the task significantly faster. Were a more complex problem provided Branch and Bound may have been able to show its strength in finding a more optimal solution at the cost of time. However, because the optimal solution was found through both algorithms and A* search preformed quicker, it can be concluded that A* search was better suited to solve this problem.