Introduction To Artificial Intelligence: Homework 1 Report

Andres Ponce(彭思安)
0616110

April 9, 2020

1 Introduction

The first assignment for the course involved a chess board, and placing a knight on that board. Given a starting and ending position, how do we move the knight along the "best" path from the start to the finish? Using several of the methods described in the textbook, the task focused on comparing the performance of each one. Some criteria may include: completeness, time and space complexity, among other general observations.

1.1 Breadth-First-Search

The first algorithm to be implemented was the classic Breadth-First-Search. In general, this algorithm keeps a "frontier" data structure where to-be-expanded nodes are stored. Every iteration, we pop a node from this set and examine its children. Since we can move by either $(\pm 1, \pm 2)$ or $(\pm 2, \pm 1)$, every node has a possible 8 children.

In this implementation, we first check the validity of the specific child node(i.e. if it has already been explored), and if we have not encountered it and is withing the board bounds, we add it to the frontier list, implemented here by `std::list<Node*>`. If we already explored a node,then we already know it does not contain a shortest path, otherwise the algorithm would have halted already.

To store the individual nodes information, we use a special `struct Node`, which essentially just stores the node's correspondent x and y values in the board. Then, a quick hash function can turn the x and y values into the index in the node array.

Breadth-First Search appears to not be the most efficient algorithm to use as is, unless we add some form of heuristic when we choose which nodes to explore. Our algorithm will eventually reach the target node, however given a large enough board the time would likely be prohibitive. Since the branching factor is 8, since we can make 8 possible moves at every node, the worst-case time complexity is $O(d^8)$, where d is the depth of the solution path.

1.2 Depth-First-Search

For depth-first-search(DFS), we first carry out a thread to the end, and as we are left without valid states to move on to that we start backtracking up the chain. On average, it tends to be slower than BFS, since we can explore a large amount of nodes before getting to the target if the target is near the starting point. However, the *memory* requirements are quite lower. This is because at any point, only the nodes on the current search path are kept in memory – the ones we have to search – rather than all the nodes at a given level such as BFS.

From some of the sample tests, DFS resulted in sometimes significantly longer search paths, since it would search for paths that lead to dead ends.

Depending on the problem, DFS would probably not be the most ideal search algorithm. These two first algorithms usually have no heuristic to select nodes, so considerable time may be spent on nodes that ultimately have no bearing on the final result. Thus, uninformed algorithms in general might not be the wisest move.

An interesting experiment or topic for further discussion given enough time might be to check the actual average difference between random points on the board.

1.3 Iterative Deepening Search

Iterative Deepening Search, sometimes referred to as *Iterative Deepening Depth-First Search*, attempts to combine the useful properties of BFS and DFS in one single algorithm. First, we set a limit on the depth of the solution, and call a depth-first search on the starting node. Thus, whenever we find the solution node, it should be done using the least amount of ndoes possible.

Every time we don't find the solution node, we increase the limit and try again. This might result in the top nodes of the search tree being generated often, however the difference still turns out not to be too great, since most nodes reside in the lower levels of the search tree(assuming constant branching factor). Even though the asymptotic running time remains $O(b^d)$ (same as BFS), the memory complexity is that of DFS $O(bd)$. For uninformed strategies with an unknown solution depth, IDS might be the best choice since like DFS for a finite space, it is guaranteed to find the solution becuase of the optimality property.

For our implementation, we use a helper function, since we recursively call the helper function. When first entering the helper function, besides from just checking whehter we are at the target, we also have to check whether the limit has been reached. Other than that, we proceed in a very similar manner to DFS.