Graph Searching N-Queens and N-Puzzle

Andrew Essex
University of Colorado Colorado Springs

Introduction

Searching algorithms have been a staple since the beginning of computation as a whole. There success in applications have solidified them in history. Artificial Intelligence (AI) has leveraged their potential to solve a multitude of problems to the point where they are the basis of basic learning. Framing problems into a search and applying search algorithms has shown to be a valid heuristic approach in most cases. In this study, 4 different search algorithms were implemented and applied to two different problem sets. the algorithms: breadth-first (bf), depth-first (df), iterative deepening (id), and bidirectional (bd). The problem sets, N-Queens and N-Puzzle. Both the algorithms and problems are ultra classics in the field of computer science. The remainder of this paper will describe how the algorithms were applied and the analysis of their performance. It is assumed knowledge of the various algorithms and problems are known therefore each will not be explained in great detail. The development language of choice was python.

Search algorithms

Each search algorithm was implemented to take 3 parameters from the requirements

- initial state/node
- a goal state or function that determines goal state
- a reference to compute successors or expand a node/state

Based on the parameters listed above each search will run and at each iteration will and expand successor Nodes/states according to each specific search i.e. (bfs, dfs, etc.) until a goal node is found.

State Representation

For both problems I needed to represent different board configurations as states. Albeit each problem had their own form of states but the general structure was the same. States were comprised of an object with the following properties.

- state - a tuple representing various board configurations both puzzle and chess
- parent - a reference to a node's (state's) parent
- depth - depth of the node/state
- cost - cost of the node/state

Details on how this representation was used are problem specific and will be expanded on below

N Puzzle

The puzzle object is initialized with an integer n representing the $n \times n$ board. This allowed us to dynamically change the board size. The empty tile is represented by the number zero. Although the board state is configured as a tuple it is interpreted as a matrix with rows and columns. Before searching for a valid puzzle state, a start and goal state needed to be generated. For the goal state, based on n a tuple was created and looped through from 1 to n^2 not inclusive then a single zero was appended resulting in

$$1, 2, 3, 4, 5, 6, 7, 8, 0$$

where $n = 3$. A similar approach was used for the initial state however it was randomized such that we can further observe the performance of each search. This was accomplished by basically taking the goal state and expanding it x number of times creating a list of tuples. Then randomly selecting one from the list. This would guarantee a pseudo random start state every time. It is worth noting that the goal state and start state share the same depth, cost, and parent of None (null) upon initialization. Possible moves up, down, left, and right are represented as indices [-n, n, -1, 1] since we are treating the tuple as if it were a matrix. In order to compute successive states possible moves from tile 0 must be determined. This is done by creating a dictionary on the fly with the key being the index from the tuple and the value being a list of valid moves from said index. For example:

$$\{0 : [1, -1, 3, -3], 1 : [1, -1]....\}$$

. Knowing the possible moves from each tile index in the state and knowing the index of tile 0 from the state we can compute the valid moves tile 0 can make resulting in successive states. For a more concrete example imagine the board was configured as such

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 0 \\ 6 & 7 & 8 \end{bmatrix}$$

We know that the only valid moves are left down, and up. Tile zero's index is five so the dictionary would have an entry

$$\{5 : [-1, 3, -3]\}$$

So from here we generate states/nodes based on the dictionary using the position of 0 and compute the following.

$$\begin{bmatrix}1 & 2 & 0\\ 4 & 5 & 3\\ 6 & 7 & 8\end{bmatrix} \begin{bmatrix}1 & 2 & 3\\ 4 & 0 & 5\\ 6 & 7 & 8\end{bmatrix} \begin{bmatrix}1 & 2 & 3\\ 4 & 5 & 8\\ 6 & 7 & 0\end{bmatrix}$$

These would be the successive nodes in a search tree. This process is repeated until a goal state is found. We know when a goal state is reached by doing a simple comparison with the state/node that is being evaluated during search time Obviously the order in which the tree is created and compared at each time step is dependent on the search algorithm.

Results and Analysis

From first glance one can formulate the size of the search space is dependent on n and is $n^2!$However upon further examination and testing one can conclude that not all successor states are reachable from the initial state. In fact only half of them are reachable. Making the state space of size $n^2!/2$. This does require knowing something about the states in advance. In general each search algorithm converged on a solution in a reasonable time while $n <= 3$ the following table shows the results averaged over 10 runs of each search where $n = 3$.

Algo.	Depth	Cost	Time
bf	10	10	0.0515 secs
df	10634	10634	0.54 secs
ids	8	8	0.332 secs
bd	8	8	0.0322 secs

From the table above we can see bd search performed the best for this problem. This is not much of a surprise since during implementation bf search performance was notably impressive. Since bd is essentially bf in both directions this was expected. On average bd performed at least 50% faster than bf. df and ids at least converged when $n <= 3$ that wasn't the case when $n >= 4$. The following captures this.

Algo.	Depth	Cost	Time
bf	18	18	36.16 secs
df	∞	∞	∞
ids	cutoff	cutoff	cutoff
bd	15	15	17.9 secs

As you can see df was unable to converge resulting in infinitely searching. ids hit the cutoff limit and therefore could not find a solution. bf and bd did converge but with an exponential time increase in comparison to the smaller value of n

Values of $n >= 5$ showed significant loss of convergence. bd would still perform relatively well but on average would take more than 3 mins to find a solution. bf search would take double amount of time which supports are claims from above. The remaining search algorithms would either never find a solution or cutoff after the depth limit. I suspect this is because the search space grows at a factorial rate.

N Queens

This problem is instantiated in a similar matter as the puzzle problem. Taking in a size n and creating a start state filled with None (null) values. Again to generate a board of dynamic size. The board configuration is also represented as a tuple where the index represents the column and number at that index represents the row where the queen is placed. This was decided to eliminate added complexity in dealing with a matrix representation. The general flow goes as follows. For each search iteration generate successors. We generate successors going column by column placing a queen in each row where there is not a clash or risk of being attacked. So for example if $n = 4$ on the first iteration we have a state

$$[None, None, None, None]$$

This would generate 4 new states

$$[0, None, None, None]$$
$$[1, None, None, None]$$
$$[2, None, None, None]$$
$$[3, None, None, None]$$

Obviously we don't have to worry about clashes through the first iteration but the comparisons are still made as such. We compare equality to each row and column. Testing if a queen is already placed at that row/column. Then we also compare row + column and row - column. This compares the diagonals left and right accordingly. Again if there are no clashes a queen is placed at that column using the index. This process is repeated until there are no queens left to place. To determine a goal state we check to see if there are any columns in the tuple whose value is None (null). If not then we loop through the whole tuple again and check for any conflicts throughout the whole tuple using the method we just described. Essentially rechecking, this solution only checks for a single solution not multiple. This is obviously a brute force approach but the results were impressive.

Results and Analysis

It is obvious that the search space is dependent on the size of the board an $n \times n$ needing to place n queens give us great intuition on the upper bounds of the search space. The formula would be

$$\frac{n^2!}{n!}$$

Since the first queen can be placed on any of the n^2 places then the next queen can be placed in any of the $n^2 - 1$ places and so forth. As the size of n grows so does the number of queens to be placed. Forming a relationship the search space does not expand in such a fashion as the n puzzle problem and the results show this.

Bidirectional search was not included in these experiments because the goal state was unknown. A work around was provided however the implementation details of the problem did not adhere to that solution. The core test case checked if all queens were placed. If they were no successors were generated. This workaround also conflicts with the logic of generating successors individually and is therefore

deemed inapplicable. However, each search algorithm aside from bd performed exceptionally well throughout the experiments. Depth first (df) converged on a solution much faster than both bf and id consistently regardless of the size of n. Df on average was 300% faster than both algorithms. An interesting observation was that no matter the algorithm the depth and cost were constituent. It was directly correlated to the size of n. This would support the reason why df performed the best by far since at it's core it is to reach the furthest depth possible. The following table shows the results from running experiments where $n = 10$

Algo.	Depth	Cost	Time
bf	10	10	1.32 secs
df	10	10	0.035 secs
ids	10	10	3.72 secs
bd	n/a	n/a	n/a

The results above are astonishing. Depth first reigns superior in solving this problem. Even when $n = 24$a solution was found in 90 seconds. It is suspected that the reason the depth is the same for all searches is due to the relationship between the size of the board and the number of queens to place. The search space doesn't explode at the same rate as n puzzle.

Conclusion

In concluding this study, 4 searching algorithms were used to solve the n-queens and n-puzzle problems. Breadth-first and Bidirectional performed superior over depth first and iterative deepening in n-puzzle. This was with small values of n. Puzzles of $n >= 5$ took exponentially longer or didn't converge at all amongst all search algorithms. With n queens we saw more consistent results as n grew. A lot was learned but feel the most important thing was applying the conceptual knowledge covered in lecture and applying it practically to ultra classic computer science problems. With this fundamental understanding we hope to increase our comprehension and solve more complex problems in the future.