Implementation Report

Assignment 1: Solving the three-digit puzzle using search algorithms
COMP3308 Artificial Intelligence
Semester 1,2012

Weilong Ding

Table of Contents

Algorithm Implementations

Breadth-First Search

Breadth-first search (BFS) is a graph search algorithm that begins at the root node and explores all the neighboring nodes. Then for each of those nearest nodes, it explores their unexplored neighbor nodes, and so on, until it finds the goal.

To implement this algorithm for the 3-digit puzzle, we need to store the children nodes and expanded nodes. Here queue is chosen because children nodes are always generated from left to right or right to left in breadth-first search and queue can guarantee that these nodes can be accessed in the way they generated. In my implementation children nodes are created from left to right. The first step of the algorithm is to check if the size of the expanded node list has reached the limit (1000 nodes) or not. When the size of the expanded node list reaches the limit, searching stops immediately and gives a warning "Depth limit reached". Then the first node in fringe will be popped out and checked if the number it contains has been expanded or not (root node will not be checked), if so discard it and pop another node, else expand it. For the currently expanded node we will check if the number it contains is the goal. If so the node will be traced back to the root while it will also be pushed into a stack named path so that the path found can be printed out in a correct order. This working flow is basically obeyed by the five other search strategies with some variations.

Depth-First-Search

Depth-first search will search a tree as deep as possible before it begins to search another branch. Due this feature depth-first search is generally done using recursion. Therefore there is no reason for me not to use this efficient and natural technique. The working process of Depth-first search is basically the same such as condition and constraint checks. The main difference is that when a number has been decided not the goal number, the first possible child of this number will be created and passed as argument to another DFS call so that the whole algorithm can be done in recursion until it finds the goal and return back all the way up to the outmost call.

What worth mentioning are:

1) For depth-first-search I did not use fringe to store the nodes yet to be expanded because DFS always goes as deep as it can so that there will always be only one node in the fringe and no other created nodes waiting to be expanded. And because of the features of DFS, when the algorithm found the goal, the path it traces back to the root is the path we found from start to node, although it may not the optimal one. Stack path can guarantee that path will be printed out in a correct order.
2) In my implementation of all the algorithms, an algorithm will return true when the goal is found or there are no nodes in fringe, i.e. there are no nodes yet to be expanded because path and expanded nodes still need to be printed out even if it is the latter situation, and return false when

depth limit is reached. But in DFS there is always one node waiting to be expanded so that this standard will not work here. Therefore I add a blob of code to check if the size of expanded node list is 1. The reason why I wrote this is that the situation that there are node nodes waiting to be expanded only happens when the start node is 000 or 999 and the forbidden numbers "block" their movement and start node will always be expanded. After the sequence of adding new children if the size of expanded node list is still 1, it means no node to expanded, but we can still return true and print out the empty path and expanded list.

Iterative-Deepening Search

Iterative-Deepening search is also done with recursion and the idea is basically the same as depth-first search except the depth-first search in IDS is depth limited for each loop and IDS will keep looping until the goal is found. To achieve IDS an extra function named Depth-limited search is implemented to do the recursion and in IDS function we only need to do the loop, passing the root node and depth as arguments to DLS. After each loop the depth will increase by 1 and the new search will go deeper.

For IDS an extra data structure named tmpExpanded is declared to store the expanded node in each loop and appended to the main expanded list because we want to show all the expanded nodes not just the ones in one round.

Greedy Search

The algorithms above are all uninformed search algorithm. Now we finally come to informed search algorithm. The advantage of informed algorithms is that they know some information that they cannot get from the problem itself. Therefore the extra information will make these algorithms optimal and efficient.

Greedy search always choose the best when searching. In the implementation we can calculate the heuristic value for a node, and choose the node with lowest value.

The way we get the heuristic value is to calculate the differences of each digit between the current node and goal node sum them up and add the difference between the largest difference and the second largest difference from above. this heuristic is admissible because:

1) if all digits can be altered freely, the number of total movements should be the sum of differences between each digit of current node and goal.
2) but same digit cannot be altered in two successive moves, therefore in certain cases we need to "sacrifice" another digit to change a digit which is a "useless" move. E.g.: if we want to transform 112 to 111 but the third digit has been changed, we need to change the first digit or the second digit first, say, 122, and then 121, 111, totally 3 steps. if we can change digits freely, only 1 step needed.
3) in the h formulation, the front part (firstDigitDiff + secondDigitDiff + thirdDigitDiff) calculate the steps needed for condition 1 and the latter part calculated difference between the largest difference and the second largest difference from above which is the least "useless" step we need.

4) so far the heuristic is admissible and another constraint for the puzzle is the forbidden numbers which make the heuristic even more admissible because we need more steps to "detour" the forbidden numbers.

Due to the specificity of informed search, we need to use a Priority Queue as the fringe. the fringe will store the nodes in a non-decreasing order. what is more important is that if we got nodes with same heuristic value, the last added node should always be stored in front of the others with same value. if we only compare the heuristic value, the last added node will be stored at the end by default therefore here to cater our requirements we need to override the comparator to have a second criteria to compare the node. we first compare the heuristic, if node1 got larger value, it will be put after node2. if node1 got smaller value, it will be put before node2. if they got same value, we continue to compare their id. A smaller ID means this node is created earlier so that we can put the node with larger ID at front and vice versa. and this new comparator also obeys the requirements from javadoc.

A* Search

A* search strategy works the same way as Greedy does: always choose the best node, but with, of course, a slight difference. While Greedy search only considers of heuristic, A* also considers of the cost from the start node to the node it is going to choose. The cost from start node plus the heuristic of the node will provide an estimated cost from start node to goal node through this node, which can make the result path more optimal. Next node to be expanded is chosen based on an f value given by function f. In function f, the depth of the given node is calculated and adds it to heuristic value. Because the cost from one node to its children is always 1, so the depth of the node is also the cost from start node to this node.

Due to the extra information A* knows, it generally give a better result than the one Greedy search gives.

Hill Climbing Search

The biggest advantage of hill-climbing search is that it minimizes the memory it use when searching. Unlike other search algorithms, hill-climbing search only store one node at a time so that fringe is useless here. It will choose the neighbor with least heuristic of current node and then compare it with the current node, if the heuristic is bigger than that of current node, stop searching. instead of using fringe to store nodes, I only use a node type minH to store the node with the least heuristic. When a new child is added, compare the heuristic of minH and the new child and pick the smaller one as the new minH. After checking all the neighbors of the node, minH is checked to see if the search should stop or not.

Empirical Results

Strat egy	Path found	Nb of expan ded nodes	Max nb of node s in	Compl ete, i.e. found	Opti mal, i.e. foun

			mem ory at a given time	a path from S to G?	d the short est path from S to G
BFS	345,355,455,465,565,555	302	554	Yes	Yes
DFS	345,245,235,135,125,025,015,115,105,005,…,555	967	967	Yes	No
IDS	345,355,455,465,565,555	429	378	Yes	Yes
Gree dy	345,355,455,456,556,555	6	19	Yes	Yes
A*	345,355,455,456,556,555	6	19	Yes	Yes
Hill-climb ing	345,355,455	3	1	No	No

Discussion

From the table above we can see that most of the search algorithms found the path in spite of optimization. Breadth-First Search found an optimal path for the puzzle but the number of expanded nodes (NE) and max number of nodes (MN) in memory at given time are 302 and 554 respectively. This result is at an intermediate level among the whole results. The price for BFS to find an optimal path is to traverse the search space level by level so that the number of expanded node and the number of nodes stored in memory can be quite large. The advantage of BFS is that it is always complete-if the shallowest goal node is at some finite depth d, breadth-first search will eventually find it after generating all shallower nodes. Although the goal breadth-first search found is not necessarily the optimal one technically, in our case all actions, i.e. change one of three digits of a number, have the same cost, which guarantees that breadth-first search is optimal in 3 digit puzzle.

The result of depth-first search looks not that good as BFS even though it eventually found a path. It is obviously seen that DFS is complete in finite space because in our problem the constraints make sure that repeated states and redundant path are avoided and it will eventually expands every nodes. But it is clearly not optimal from the path we got. As the search proceeds immediately to the deepest level of the search tree, where the nodes have no successors, it may choose a non-optimal goal instead of an optimal goal at a shallower level.

Although a depth-first tree search may generate all of the $O(b^m)$ nodes in the search tree, where m is the maximum depth of any node, it only needs to store a single path from the root to a leaf node, along with the remaining unexpanded sibling nodes for each node on the path. Once a node has been expanded , it can be removed from memory as soon as all its descendants have been fully explored so that it may need less memory than BFS does in certain cases.

Iterative deepening search combines the benefits of depth-first search and breadth-first search. Like depth-first search, its memory requirements are modest: $O(b^d)$ to be precise. Like breadth-first search, it is complete when the branching factor is finite and optimal when the path cost is a non-decreasing function of the depth of the node. Due to the feature it inherits from BFS IDS is optimal and complete in finite space as well. It also uses less memory than BFS does. In general, iterative deepening is the preferred uninformed search method when the search space is large and the depth of the solution is not known.

Compared to the uninformed search algorithm we analyzed above, informed search algorithms are obviously far more efficient. Besides the information within the problem, informed search are provided with extra information to make the algorithm faster and optimal. For greedy search although it finally found an optimal path, it is not optimal because tries to expand the node that is closest to the goal, on the grounds that this is likely to lead to a solution quickly. Even if it finds a path with minimal cost, the path may not be an optimal one because greedy search only choose what is the best so far, not based on general situation. This shows why the algorithm is called greedy.

And greedy search is also not complete because it may encounter a dead end and the search will be unable to continue. In spite of these shortcomings greedy search still can be a quite efficient algorithm with a good heuristic function.

A* search, although as one of best-first search as greedy search, is optimal if its heuristic is admissible. An admissible heuristic is one that never overestimates the cost to reach the goal. The heuristic A* uses in this problem, as described above, is admissible. From the table we can see the results from greedy search and A* search are the same. But A* is search is complete, optimal and optimally efficient. For 3-digit puzzle all actions have same cost, which cannot distinguish greedy search and A* search very well. A* search will appears to be more powerful in which the action costs are different.

Although the features of A* search are rather satisfying, unfortunately it does not mean that A* is the answer to all our searching needs. The catch is that, for most problems, the number of states within the goal contour search space is still exponential in the length of the solution.

Hill-climbing search is the only algorithm that is not complete in 3-digit puzzle. The biggest advantage of hill-climbing search is that it's extremely memory-efficient because it only stores one node at any stage. That is why this kind of searches is called local search. Besides little memory they use, local search can often find reasonable solutions in large or infinite state spaces for which systematic algorithms are unsuitable. Hill-climbing search is sometimes called greedy local search because it grabs a good neighbor state without thinking ahead about where to go next. Hill-climbing often makes rapid progress toward a solution because it is usually quite easy to improve a bad state, but they often fail to find a goal when one exists as well because they can get stuck on local maxima like it did in our problem. So the success of hill-climbing depends very much on

the shape of the state-space land-scape: if there are few local maxima and plateau, hill-climbing will find a good solution very quickly.

Reflection

From this assignment I got a great opportunity to implement these search algorithms by myself and it does help me understand how these search algorithm work and in which cases I should use them. Although there were a few obstacles when I tried to implement them, luckily they are all being solved and this definitely takes my understanding of these algorithms to the next level. There are also some interesting challenges like carrying out my own heuristic for informed search. This process is kind of a brainstorming, which I pretty enjoy it, and when I found my heuristic works I did have a great feeling of achievement.