Implementation Report

Assignment 1: Solving the three-digit puzzle using search algorithms

COMP3308 Artificial Intelligence

Semester 1,2012

# 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 an "useless" move. Eg: 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 obey 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

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Strategy | Path found | Nb of expanded nodes | Max nb of nodes in memory at a given time | Complete, i.e. found a path from S to G? | Optimal, i.e. found the shortest 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 | 1522 | Yes | No |
| IDS | 345,355,455,465,565,555 | 429 | 1675 | Yes | Yes |
| Greedy | 345,355,455,456,556,555 | 6 | 19 | Yes | Yes |
| A\* | 345,355,455,456,556,555 | 6 | 19 | Yes | Yes |
| Hill-climbing | 345,355,455 | 3 | 1 | No | No |

# Discussion

# Reflection