2048-solver Project report

Samy Shehata 22-3798
Omar Mahmoud
Karim Tarek
October 16, 2014

1 THE 2048 PROBLEM

This project is an Artificial Intelligence solver for 2048 game. 2048 is a board game with 4*4 blocks. Intially the board has two random blocks having the value of 2, the rest of the grid is empty. The game has four operaters : Left, Up, Right and Down. Each one of these operators moves all the blocks on the board in a certain direction. If two blocks with the same values collide, a new block is generated with the a value equal to their sum and the old blocks are removed. After each move a block with the value 2 will be placed randomly in any empty place in the board. The goal of the game is to reach a block with the value 2048. The user will lose the game if no moves will have an effect on the board. For the purpose of this project we only consider a more relaxed version of the problem. After each move a block having the value 2 is placed in the upper-left corner in the board, if the upper-left corner is not empty then it will be placed in the next corner moving clockwise. If there are no empty corners nothing will be added. This was done to eliminate the factor of randomness in the problem as the solver uses search-based algorithims.

2 IMPLEMENTATION

2.1 OVERVIEW

SEARCH TREE NODE A node is an abstract data type modelled by the struct Node with the following fields:

- Depth: The depth of the node in the search tree starting with depth 0 for the root.

- Cost: The cost of reaching the node starting from the root.
- Parent: A pointer to the parent node in the tree, nil for the root.
- Operator: The operator applied to the parent in order to reach the node.
- State: The state of the problem after applying the node operator.

SEARCH PROBLEM A search problem is represented by the class Search-p. It has the following members:

- S0: The depth of the node in the search tree starting with depth 0 for the root.
- Operators: a list of possible operators that can be applied to the problem state to reach the next state in the state space.
- Goal-test: a function that can be applied to a search tree node which returns true if the node is a goal node I.e the state of the node is the target state.
- Path-cost: a funcation that is applied to the cost of a parent node and the cost of applyng an operator to calculate the cost of reaching a new node.

The search-p class implements the following methods:

FUNCTION GENERAL-SEARCH
Input

- Prp: A search-p object representing the search problem.
- qing-fun: A function that handles the search strategy.

Output

- Sol: A list of operators that form the found sol (if it exists).
- Expand-count: The number of node expanded during search.
- Sol-cost: The total cost of the solution returned.

General search creates a queue with only one node containing the initial state of the problem and then uses the the qing-fun to recursivly expand nodes and search for the goal node. Returns nil on failure of finding a solution.

2048 PROBLEM The 2048 problem is defined by a the Grid class with the following members:

- Blocks: a square two dimensional array representing the board grid. The array holds 0s for empty blocks and the value of the blocks otherwise.
- Size: the size of the grid. Since the grid is always square, blocks are defined as a size * size array.
- Maximum: the current largest block in the grid.

The state of the 2048 problem can be affected by four operators represented by the following member functions:

- Left:Collects all grid blocks to the left.
- Right: Collects all grid blocks to the right.
- Up: Collects all grid blocks upwards.
- Down: Collects all grid blocks downwards.

2.2 DETAILS

Our implementation is split into three files.

2.2.1 GRID.LISP

This file contains all functions specific to the 2048 game implementation.

FUNCTION GENGRID
Input

- void

Output

- Grid

Generates a new grid object and and adds two blocks with the value 2 in two random locations.

FUNCTION GRID-DISPLAY
Input

- Grid

Output

- void

prints the grid blocks as a square matrix.

FUNCTION GRID-SPAWNBLOCK
Input

- Grid
- X: The col index of the grid
- Y: The row index of the grid

Output

- void

Adds a new block with the value 2 at position (x y) of the grid.

FUNCTION GRID-EMPTYBLOCKP
Input

- Grid
- X: The col index of the grid
- Y: The row index of the grid

Output

- boolean

Returns true if the block at position (x, y) is empty.

FUNCTION GRID-UPDATE
Input

- Grid

Output

- Void

Distructive function the adds new block to the array. This function is called after every operator is applied to generate the new block. It checks the four corners for the grid in order staring from the top left corner and adds the block in the first empty corner.

FUNCTION GRID-FIND-MAX
Input

- Grid

Output

- Void

loops over the grid blocks and finds the current maximum block. Sets the class member "maximum"

FUNCTION EQLP
Input

- Grid1
- Grid2

Output

- boolean

returns true if the two grids are identical.

FUNCTION GRID-RIGHT, GRID-LEFT, GRID-UP, GRID-DOWN
Input

- Grid

Output

- Grid
- Cost: The cost of applying the operator.

Four functions representing the four operators that can be applied on a grid. Operators work in the same way:

1. Create a new grid.
2. Set the grid blocks to the result of applying the operator to the old grid
3. Check if the new grid is equal to the old grid. If equal return null otherwise go to 4.
4. Call Grid-update to add a new block to the new grid.
5. Grid-find-max to find the new largest block.
6. Return the new grid and the score acqured from applying the operator.

FUNCTION COLLECT-RIGHT, COLLECT-LEFT
Input

- List

Output

- List

Helper functions to the operator functions each taking a list and outputing another list after collecting all elements right and left respectively and pairwise merging equal elements (by calling pairwise merge). The list is then padded with zeroes on the edge. These functions are applied on each row for Grid-right and Grid-left. Grid-up and Grid-down use the same function while rotating the grid first.

FUNCTION PAIRWISE-MERGE
Input

- List

Output

- List

Helper function to collect-right and collect-left functions. Recursively filter out 0s and merge each two equal elements from left to right.
e.g 2 0 2 4 -> 4 4

2.2.2 MAIN.LISP

This is the main file that contains the ADTs and the general search functions.

FUNCTION SEARCH
Input

- Grid
- M: The target value of the search problem.
- Strategy: The search algorithim used.
- Visualise: A boolean whether to visualise solution or not.

Output

- List: Operations that lead to the goal state. Null if goal not reached.
- Expanded-count: Number of nodes expanded.
- Solution-Cost: cost of the returned solution

The function creates search-p object from the input grid and M. It then passes the object to general-search function to find a solution. If visualise is set, the function calls show-solution function to display the steps of the found solution

FUNCTION GENERAL-SEARCH
Input

- prp: A search problem
- qing-fun : A function the determines the search strategy to find a solution

Output

- Sol: a list of operators that form the found sol (if it exists).

- Expanded-count: Number of nodes expanded.
- Solution-Cost: cost of the returned solution

General search creates a queue with only one node containing the initial state of the problem and passes it to search-helper function which returns the goal node found. The function passes the goal-node to form-solution function which returns a list of the operators forming the solution starting from the root. Returns nil on failure of finding a solution.

FUNCTION SEARCH-HELPER
Input

- prp: A search problem.
- queue: A queue of the current nodes awaiting expansion
- qing-fun : A function the determines the search strategy to find a solution

Output

- goal-node: The goal node found from which the solution can be reconstructed
- Expanded-count: Number of nodes expanded.

Iteratively expands the nodes in the queue and adds the generated children using the quing function. First the function checks if the head of the queue is goal node. If true the function returns it with the number of nodes expanded. Otherwise the function passes the head of the queue to the Expand function which returns a list with the children of the node. Then the function applies quing-fun to both the current queue and the children list. The function fails when the queue is empty.

FUNCTION FORM-SOLUTION
Input

- n: The goal node.

Output

- op-list: List of operaters that start from the root to form the solution.

The function check if n is the root. If this is the case, op-list will be returned. Else the operater of n will be concatenated to op-list and the function will be called recursivly on the parent of n and the new op-list.

FUNCTION FORM-SOLUTION

Input

- s0: The initial state.
- Sol: List of operators to reach the goal state.
- Print-fun: Function to be used to display the state.

Output

- void

Iterates on the list of operaters uplying them to the initial state till reaching the goal state. Uses print-fun to print each new state.

2.3 SEARCH.LISP

This file has the functions that implements the different search strategies.

All search strategies are implemented as functions that take a queue and a new list. The queue is the set of nodes to be expanded and the new list is the nodes newly generated from expanding a node.

BREADTH FIRST SEARCH (BF) The new list gets appended to the end of the queue.

DEPTH FIRST SEARCH (DF) The new list gets appended to the beging of the queue.

ITERATIVE DEPTH (ID) Applys Df until a cut-off depth then the depth is incremented and Dfs is restarted. ID terminates when no nodes generated exceed the cutoff depth.

A* (AS1, AS2) Applies a merge sort between the queue and the new list according to a given heuristic. A* Heurisitcs consider the cost of the path already traversed as well as the expected cost to the goal.

GREEDY (GR1, GR2) Applies a merge sort between the queue and the new list according to a given heuristic.

2.4 HEURISTICS

Heuristics are implemented as a comparison function between two nodes n1, n2. The Heuristic should return true if n1 is “better” than n2. This is used by merge sort to order the nodes better to worse.

H1 Consideres the number of doubles needed to get the current maximum block of the grid to the target block. This heuristics is admissible, however, it is a poor estimation as it returns values much smaller than the actual cost.

H2 Consideres the score that will be accumilated from doubling the current maximum block till it reaches the largest block. This heurisitic is admissible and it dominates h1 which is why it is a better estimation of the expected cost to reach the node.

H3 Consideres the number of combination moves that are needed to reach the target block, assuming that only the current maximum block exists in the grid. Not admissible as it ignores the other blocks existing in the grid which may reduce the actual cost of reaching the node.

3 Performance Comparison

For our implementation the best performing search strategy is depth first search. This is due certain properties of the problem and the implementation itself. First we note that to reach a goal of M=32, requires a solution of at least 16 moves. Since the moves are sequential, they must come from consecutive depths in the search tree. This is the main advantages of depth first search, as it reaches higher depth in the shortest time. In contrast breadth first search wastes time searching in higher levels of the tree, even though they can not contain the solution. Considering the M=32 example, a BFS would have to expand 4^{32} before reaching a depth where may exist. Note that there is no guarantie that the search will not need to go to even further depths to find the solution. This means that the number of nodes expanded by BFS is several orders of magnitude smaller than the number of nodes expanded by DFS. Other than the time complexity BFS suffer from a memory problem as the entire tree up to the solution depth is expanded. This makes BFS more likey to exhausted the lisp heap. Ierative deepining suffers from an even worse time complexity than BFS, however it has a slightly better memory cost. Naturally with no infinite branches (any operator will at some point be non effective), there is no reason to use ID instead of BFS.

When comparing DFS to heuristic based methods, we find that DFS still has better perfomance.While theoratically this should not be the case, the implementation of the priority queue to be used in greedy and A* seaches greatly degrades the performace. The priority queue uses a recursive merge sort function to maintain ordering of nodes. This can be very slow as the depth, and by extension the queue, becomes large. It is aslo likely that as the algorithm searches for higher order solutions (higher M), it will suffer from a stack overflow. For this implementation, greedy search out performes A* search. This is due to the definition of the cost. In our definition, the cost increases as we make more combinations of blocks which conflicts with the fact that combining blocks gets us closer to the solution. Because of this, A* spends more time exploring solutions that make the least combinations as they have better cost, even though they do not get it closer to the goal.

In conclusion we state that DFS is currently the best strategy, specific to our implementation, for solving the 2048 problem. The performance of heuristic based searches can be improved by a better implementation for the priority queue, possibly using non recursive heab sort.A* specifically can be improved with a better definition of the cost, such that it does not conflict with the direction of reaching the goal node.