Report for Programming Problem 1 - 2048

Team:

Student ID: 2018293728 Name: Miguel Rabuge
Student ID: 2018283166 Name: Pedro Rodrigues

1. Algorithm description

1. **Pre-processing:**
   For a given board to be solvable it needs to verify that the sum of all the digits must be a power of two, otherwise it will be impossible to merge all tiles of a given board ending up with only one. This condition is tested before the start of the backtracking itself in order to automatically exclude boards that would only waste computer processing time not reaching a feasible solution.

2. **Backtracking:**

   2.1. Base Cases:

   2.1.1. One tile left in the Board (Accept Case)
   This base case is the most basic one. If there is only one tile in the board, then we have reached a solution, which might not be an optimal one.

   2.1.2. Maximum Recursion Depth is reached (Reject Case)
   This base case is caused by a recursion depth threshold given by the problem input. The point is to drop the current branch if the threshold is reached.

   2.1.3. Speed-Up Tricks (Reject Conditions)

   2.1.3.1. Best Solution Variable
   Conceptually, this trick consists in starting our “best solution” variable by assigning it to +infinite, since we want to minimize the number of moves to reach a solution. From this point on, every time that we get to the 2.1.1. base case, we update this “best solution” variable if the found solution is lower than the current one. Any sequent branches that use more moves than our current best are dropped.

   2.1.3.2. Board comparison after a move
   This optimization tests if a move made any changes to the board. If it did not, the branch is dropped.

   2.1.3.3. Jam Checking
   The “Jam checking” consists in checking if a jam is

happening, in which the boards are not equal, which would be caught by the 2.1.3.2. optimization but they are horizontally or vertically equivalent, therefore being dropped.

2.2. Recursive Step:
The recursive step consists in making a move (Left, Right, Up, Down), returning true if any branch returns true, or false if every branch returns false.

2. Data structures

In the case of this problem the data structures used do not need to be that complex because the only required information that needs to be stored in a given moment is the state of our game board. This game board is a square having a dimension of NxN being N the number of rows/columns. The matrix data structure seems to be the perfect fit for this case.

3. Correctness

1. **Pre-Processing:** In 2048, every tile is a power of 2 and only equal numbers merge with each other, creating a higher power:
   $2^{(k+1)} = 2^k + 2^k, \quad k = 1,2,..,n$
   With that being said, in a solution, the last merged tile has to be a power of 2.

2. **Maximum Recursion Depth is reached:** It is guaranteed by the problem statement that no optimal solution is found after this point. Formally, let D be the maximum recursion depth and C the current recursion depth. If C is equal to D, the branch is dropped.

3. **Best Solution Variable:** Even if it found a solution, it will not be better than our current one. Formally, let B be the best (lowest) solution found so far and C the move count so far in a given branch. If C == B, the branch is dropped since no better (lower) solution will be found.

4. **Board comparison after a move:** Even if it found a solution, there will be a better one, if you just remove this motionless move. Let X be the current state of the board, Y a feasible solution from the state X. Let m1, m2 and m3 be sorted moves to get from X to Y. Let also S1, S2 be intermediate states and C the solution move count. The solution path is: (X) —m1–> (S1) —m2–> (S2) —m3–> (Y). But if S2 is exactly equal to S1, then the m2 move is redundant and can be safely removed from the solution path. This way, a solution is still found, and it is a better one, since it requires C - 1 moves.

5. **Jam Checking:** Even if it found a solution, there will be a better one, if you just remove the jam moves. Let X be the current state of the board, Y a feasible solution from the state X. Let m1, m2 and m3 be sorted

moves to get from X to Y, where m1 and m2 are inverse moves, i.e. RIGHT and LEFT or UP and DOWN or vice-versa. Let also S1, S2 be intermediate states and C the solution move count. The solution path is: (X) —m1–> (S1) —m2–> (S2) —m3–> (Y). But if S2 is horizontally or vertically equivalent to X, then the m1, m2 moves are redundant and can be safely removed from the solution path. This way, a solution is still found, and it is a better one, since it requires C - 2 moves.

4. Algorithm Analysis

We can measure the temporal performance of an algorithm by taking into account the number of recursive calls in the worst case scenario (that is, the rejection conditions never being triggered). Obviously the number of operations might depend on the board to be evaluated as some boards might be accepted/rejected with little or no computation time required but others would need to check features to yield the same conclusion. In terms of the base case of the recursive process we can say the checking process of both accepting or rejecting a board can be done with a number of operations that contribute with a given constant time “c” to the algorithm.
Being T(n) the temporal cost of the recursive call of our backtracking function with only n moves left and “c” an arbitrary constant representing the additional computational cost of a given recursive call, the following demonstration attempts to prove the temporal complexity of the algorithm by unrolling the recursive process:

$$T(n) = 4 * (T(n - 1) + c)$$
$$= 4 * (4 * (T(n - 2) + c) + c)$$
$$= 4 * (4 * (4 * (T(n - 3) + c) + c) + c) = (...)$$

We can see that the following expressions follow a progression where in each recursion the backtracking function T(N) will have smaller value of N (N = n, n - 1, n - 2, … , n - k, …, 0) and will always be prepended of a higher power of 4 with each recursive call. The number of constant terms “c” will follow a geometric progression, which at the end of the recursive process will yield a sum given by the formula: Sn = (r^n -1) / (r - 1) with r = 4 and n = 1, 2, …, N. This being said we can conclude that temporal worst case complexity of our algorithm after k steps is given by the following expression.

$$T(n) = 4^k * T(n - k) * ((4^k - 1) / 3) * c \quad \text{--> } O(4^n)$$

In terms of spatial complexity, according to our implementation we need N^2 auxiliary space to store the state of the board in any given moment. Considering case where we need to do K recursive calls and considering that each recursive call needs also an additional constant amount of space “c” to store variables, etc…:

$$S(N) = K * N^2 + c \text{ -> } O(N^2)$$

5. References

[1] https://en.cppreference.com/w/
[2] T. H. Cormen, et a.l Introduction to Algorithms, 3rd ed., 2009.
[3] J. Erikson, Algorithms, 2019