UNIVERSITY
OF LODZ

Faculty of Mathematics and Computer Science

The $(n^2 - 1)$-Puzzle

Artificial Inteligence in Computer Games
Final Project

Double Degree in Mathematics and Computer Science (University of Granada, Spain)

Author:
Leandro Jorge Fernández Vega

December 28, 2024

Contents

1 Goals

- Apply the mathematical Group Theory to an AI problem.
- Get familiar with working with data trees and complex data structures.
- Offer a game-solving algorithm for the famous *15-Puzzle* and generalize it to any board size.

2 Introduction

The construction of an algorithm which is able to solve the famous *15-Game* is proposed.

The *15-Puzzle* is a board game which consists of 15 numbered tiles and an empty one. The goal is to put every tile in order by sliding the tiles into the empty space.

Figure 1: Example of a solved *15-Puzzle*.

3 Mathematical Analysis

Let $n \in \mathbb{N}$, $n \geq 2$. Our goal is to provide an algorithm which solves the $(n^2 - 1)$-*Puzzle*.

The first thing we need to consider is solvability. Can every situation on the board lead to a solution?

Theorem 1 (Johnson and Story (1879)). *On boards of size $m \times n$, where $m, n \geq 2$, all even permutations are solvable.*

It can be proven by induction on m and n, starting with $m = n = 2$. This theorem proves that every situation on the board can be generated by 3-cycles.

Corollary 1. *A board of size n^2 can be represented by the Alternating Group A_{n^2-1}. This means that the total possible states which lead to a solution are reduced to the number $|A_{n^2-1}| = \frac{(n^2-1)!}{2}$*

On the other hand, we can also consider which is the solution with the fewest number of slides. This is, in fact, an NP-Hard problem, although there is a polynomial-time constant-factor approximation.

It is known that the *15-Game* can be solved in no more than 80 moves, while the *8-Game* in no more than 31.

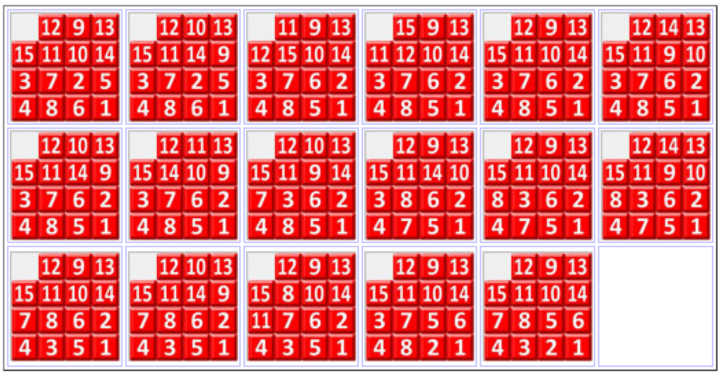

Figure 2: The 17 positions of the *15-Puzzle* which need 80 moves.

4 Computational Analysis

4.1 Board Generation

We first need to think how to generate a solvable situation for our game player. Knowing that every solvable distribution of the board is generated by 3-cycles, it is intelligent to start from the puzzle's idle position and apply any number of 3-cycles to shuffle it, obtaining a solvable distribution.

4.2 Algorithm Selection

On the other hand, it is necessary to think about what algorithm solves the problem in a more efficient way. However, we have to know that dealing with permutations and exploring a tree of possible solutions frequently involve the use of non-polynomial algorithms.

In this case, an easy option for our problem would be the A* Algorithm.

4.2.1 Properties

- Completeness: If a solution exists, A* will find it.
- Optimality: A* always finds the optimal solution regarding cost. In our case, it finds the solution with the least number of movements, although not in a polynomial time.
- It uses a function to calculate costs: Let $f(n) = g(n) + h(n)$, where g is a real-cost function, and h an heuristic one, which estimates the cost to a possible solution.
- In this case, both time and space complexities are factorial in the worst case, as we are dealing with permutations.

4.2.2 Structures

- Explored Set: Nodes which have already been visited. It is frequent to use either a set, an unordered set or an unordered map. For this example, a simple set gives better results.
- Frontier Set: Nodes which are to be explored. The structure used is a priority queue which sorts based on cost.

4.2.3 Heuristic Selection

The heuristic function determines the number of nodes which are expanded. As a result, the selection of this function is an important step in the process. Heuristics also have some characteristics. Let h be an heuristic function.

Definition 1. *We say that h is admissible* $\iff$

$$h(n) \leq h^*(n) \ \forall n \text{ node}$$

where $h^*(n)$ is the true cost to reach node n. In other words, the heuristic does not overestimate costs.

Definition 2. *We say that h is monotonic (or consistent)* $\iff$

$$h(n) - h(n') \leq c(n, n') \ \forall n, n' \text{ nodes}$$

where $c(n, n')$ is the cost of going from node n to node n'. This guarantees the optimal solution is found without the need to update costs and search in the frontier set.

Definition 3. *Let h_1 and h_2 be heuristics. We say that h_2 dominates h_1* $\iff$

$$h_2(n) \geq h_1(n) \ \forall n \text{ node}$$

In practice, this means h_1 will expand, at least, the same number of nodes as h_2.

Proposition 1. *Monotonicity* $\implies$ *Admissibility.*

Theorem 2. *Let h be an admissible heuristic used by the A* Algorithm. Then, the A* Algorithm is complete and optimal regarding cost.*

Knowing this, we can now decide which heuristic to use.

Manhattan Distance One really appropriate heuristic for this problem is the Manhattan Distance, which is defined as:

$$d_{\text{Manhattan}} = \sum_{i=1}^{n} \sum_{j=1}^{n} |x_i^j - y_i^j|$$

where x_i^j is the tile i, j on the board and y_i^j is the position where x_i^j should be.

It will measure how far each tile is from its correct position, and then add all these values.

Inverse Distance This heuristic builds upon linear conflicts and uses the idea of inversions.

We define an inversion to be when a tile appears before another tile with a smaller number. The blank has no number and cannot contribute to inversions.

Considerations:

- When moving a tile horizontally, the total number of inversions never changes. This is due to the blank not affecting inversions.
- When moving a tile vertically, the total number of inversions can change by only -3, -1, +1, and +3.
  - Case 1: the three skipped tiles are all smaller (or larger) than the moved tile. Moving the tile will either add or fix three inversions, one for each skipped tile. So, the total number of inversions changes by +3 or -3.
  - Case 2: two of the tiles are larger and other is smaller (or vice versa). In this case, there's going to be a net change of +1 or -1 inversions.
- One vertical move can fix at most three inversions. If we assume the minimum number of vertical moves needed to fix the inversions, that results in $\lfloor n\_inversions/3 \rfloor$. If there is a remainder, the remaining inversions can be solved with at least one vertical move per remaining inversion.

As a result, going through the board left-to-right, top-to-bottom, we can define

$$\text{vertical} = \lfloor \frac{\text{n\_inversions}}{3} \rfloor + \text{n\_inversions}\%3$$

Analogously, we can define the horizontal count of inversion, but noticing we will go through the board top-to-bottom, left-to-right.

Finally, the heuristic is defined as:

$$d_{\text{Inverse}} = \text{vertical} + \text{horizontal}$$

Walking Distance We now define the walking distance as the sum of the Manhattan Distance and the Inverse Distance.

$$d_{\text{Walking}} = d_{\text{Manhattan}} + \alpha \cdot d_{\text{Inverse}}$$

where α is a weighting factor.

Proposition 2. *The Manhattan Distance for the $(n^2 - 1)$-Puzzle problem is a monotonic heuristic.*

Proposition 3. *The Inverse Distance for the $(n^2 - 1)$-Puzzle problem is an admissible heuristic.*

Proposition 4. *The Walking Distance for the $(n^2 - 1)$-Puzzle problem is an admissible heuristic for an appropriate value of α, which dominates the Manhattan Distance and the Inverse Distance Heuristics. In particular, for $\alpha = 2$, this is true.*

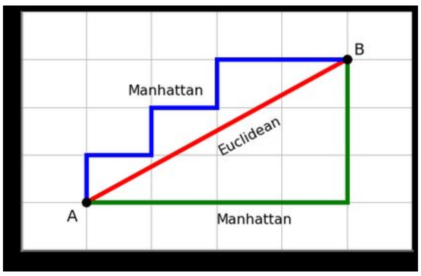

Figure 3: Comparison between the Euclidean and Manhattan distances.

5 Implementation

The implementation of the game has been done in C++11.

In this section, every function is defined in its respective header file. All these definitions have been omitted to avoid redundancy. Only the most relevant aspects of these headers are shown.

5.1 Generator

```
//generator.h
#ifndef GENERATOR_H
#define GENERATOR_H

#include <vector>
#include <iostream>

using namespace std;

const int N_PERMUTATIONS = 100; ///< Number of permutations to generate
const int LOWER_BOUND = 2; ///< Lower bound for board size
const int UPPER_BOUND = 4; ///< Upper bound for board size
#endif // GENERATOR_H
```

```cpp
//generator.cpp
#include "../include/generator.h"

using namespace std;

/**
* @brief Function to get a random number from the vector
    without repetition.
* @param v The vector from which to get the random
   number
* @return A random number from the vector
*/
int getNum(vector<int>& v) {

    // Size of the vector
    int n = v.size();

    // Generate a random number

    // Make sure the number is within the index range
    int index = rand() % n;

    // Get random number from the vector
    int num = v[index];

    // Remove the number from the vector
    swap(v[index], v[n-1]);
    v.pop_back();

    // Return the removed number
    return num;
}
```

```
/**
 * @brief Function to generate a random vector of n different numbers.
 * @param n The number of different numbers to generate
 * @return A vector of n different random numbers
 */
vector<int> generateRandom(int n) {
    vector<int> v(n-1);

    // Fill the vector with the values
    // 1, 2, 3, ..., n-1
    for (int i = 1; i < n; i++) v[i-1] = i;

    vector<int> u;

    for(int i = 0; i < 3; i++) u.push_back(getNum(v));

    return u;
}
```

```cpp
/**
 * @brief Function to generate a permutated matrix.
 * @param n The size of the matrix (n x n)
 * @return A permutated matrix of size n x n
 */
vector<vector<int>> generateBoard(int n){

    srand(time(NULL));

    int n_squared = n*n;
    vector<int> permutation(n_squared);

    //Fill the board with the numbers from 0 to n^2-1
    for(int i = 0; i < n_squared; i++) permutation[i] = i;

    //Do the permutations
    for(int i = 0; i < N_PERMUTATIONS; i++){

        vector<int> swapper = generateRandom(n_squared);
        int swap = permutation[swapper[0]];

        permutation[swapper[0]] =  permutation[swapper[1]];
        permutation[swapper[1]] = swap;

        swap = permutation[swapper[2]];
        permutation[swapper[2]] = permutation[swapper[0]];
        permutation[swapper[0]] = swap;
    }

    //Constructing the new permutated matrix
    int index = 0;
    vector<vector<int>> matrix(n, vector<int>(n));
    for(int i = 0; i < n; i++){
        for(int j = 0; j < n; j++){
            matrix[i][j] = permutation[index++];
        }
    }

    return matrix;
}
```

```cpp
/**
 * @brief Function to show a matrix.
 * @param matrix The matrix to show
 */
void showMatrix(const vector<vector<int>>& matrix){
    int n=matrix.size();
    for(int i = 0; i < n; i++){
        for(int j = 0; j < n; j++){
            cout << matrix[i][j] << "\t";
        }
        cout << endl << endl;
    }
}
```

5.2 Solver

```
    //A_star.h
    #ifndef A_STAR_H
    #define A_STAR_H

    #include <queue>
    #include <vector>
    #include <list>
    #include <unordered_set>
    #include <set>
    #include <cmath>
    #include <string>

    using namespace std;
/**
* @brief Enum to represent the possible moves in the puzzle.
*/
enum class Action {
    UP,     ///< Move the empty tile down
    DOWN,   ///< Move the empty tile up
    LEFT,   ///< Move the empty tile right
    RIGHT   ///< Move the empty tile left
};
```

```
/**
* @brief Node structure to store the state of the board, the cost, and the moves made.
*/
struct node {

    vector<vector<int>> board; ///< The current state of the board
    int cost; ///< The real cost to reach this node
    pair<int, int> pos_0; ///< The position of the empty tile on the board
    list<Action> moves; ///< The list of moves made to reach this node

    /**
    * @brief Default constructor for the node.
    */
    node() {
        this->cost = 0;
    }

    /**
    * @brief Parameterized constructor for the node.
    * @param board The state of the board
    * @param cost The cost to reach this node
    * @param pos_0 The position of the empty tile
    * @param moves The list of moves made to reach this node
    */
    node(const vector<vector<int>>& board, int cost, const pair<int, int>& pos_0, const list<Action>& moves) {
        this->board = board;
        this->cost = cost;
        this->pos_0 = pos_0;
        this->moves = moves;
    }
```

```cpp
    /**
    * @brief Copy constructor for the node.
    * @param other The node to copy from
    */
    node(const node& other) {*this=other;}

    /**
    * @brief Overloading the assignment operator to copy the values of a node.
    * @param other The node to copy from
    * @return A reference to the assigned node
    */
    node& operator=(const node& other) {
        if (this != &other) {
            this->board = other.board;
            this->cost = other.cost;
            this->pos_0 = other.pos_0;
            this->moves = other.moves;
        }
        return *this;
    }

    /**
    * @brief Overloading the equality operator to compare two nodes.
    * @param other The node to compare with
    * @return True if the boards are equal, false otherwise
    */
    bool operator==(const node& other) const {return this->board == other.board;}

    /**
    * @brief Overloading the less-than operator to compare two nodes.
    * @param other The node to compare with
    * @return True if this node is less than the other node, false otherwise
    */
    bool operator<(const node& other) const {return this->board < other.board;}
};
```

```cpp
/**
 * @brief Functor for the priority queue to compare nodes based on their cost.
 */
class Comparer {
public:
    /**
     * @brief Comparison operator to compare two nodes.
     * @param a The first node
     * @param b The second node
     * @return True if the cost of node a is greater than the cost of node b, false otherwise
     */
    bool operator()(const node& a, const node& b) {
        return a.cost + WalkingDistanceHeuristic(a.board) > b.cost + WalkingDistanceHeuristic(b.board);
    }
};
```

The following code is the implementation of a hash function in case we want to use an unordered set to manage the explored set of nodes. However, in this case, the explored set is managed with a normal set structure.

```
/**
 * @brief Hash function for the unordered_set to hash
   nodes.
 */
class Hash {
    public:
        /**
         * @brief Hash function to generate a hash value
    for a node.
         * @param node The node to hash
         * @return The hash value of the node
         */
        size_t operator()(const node& node) const {
            size_t hashValue = 0;
            hash<int> hashInt;  // Hash Function for
   integers

            for (int i = 0; i < node.board.size(); i++)
   {
                for (int j = 0; j < node.board[i].size()
   ; j++) {
                    hashValue ^= hashInt(node.board[i][j
   ]) + 0x9e3779b9 + (hashValue << 6) + (hashValue >> 2)
   ;
                }
            }
            return hashValue;
        }
};
#endif // A_STAR_H
```

```cpp
//A_star.cpp
#include "../include/A_star.h"

using namespace std;

/**
 * @brief Function to calculate the Manhattan distance heuristic.
 * @param board The state of the board
 * @return The Manhattan distance heuristic value
 */
int ManhattanDistanceHeuristic(const vector<vector<int>>& board){

    int cost = 0;
  int n = board.size();
    for(int i=0; i<n; i++){
        for(int j=0; j<n; j++){
            if(board[i][j] != 0)
                cost += abs(i - board[i][j]/n) + abs(j - board[i][j]%n);
        }
    }
    return cost;
}
```

```
/**
 * @brief Function to calculate the Inversion distance heuristic.
 * @param board The state of the board
 * @return The Inversion distance heuristic value
 */
int InversionDistanceHeuristic(const vector<vector<int>>& board){

    int n= board.size();
    int inversion_count = 0;
    for(int i=0; i<n; i++){
        for(int j=0; j<n-1; j++){
            if(board[i][j] != 0){
                if(board[i][j] > board[i][j+1]) inversion_count++;
                if(board[j][i] > board[j+1][i]) inversion_count++;
            }
        }
    }

    return inversion_count/3 + inversion_count%3;
}
```

```
/**
 * @brief Function to calculate the Walking distance heuristic.
 * @param board The state of the board
 * @return The Walking distance heuristic value
 */
double WalkingDistanceHeuristic(const vector<vector<int>>& board, double weight){

  return ManhattanDistanceHeuristic(board) + weight*InversionDistanceHeuristic(board);
}
```

```cpp
/**
 * @brief Function to find the empty tile on the board.
 * @param board The state of the board
 * @return The position of the empty tile
 */
pair<int,int> findEmptyTile(const vector<vector<int>>&
    board){

  int n = board.size();
  for(int i=0; i<n; i++){
    for(int j=0; j<n; j++){
      if(board[i][j] == 0)
        return pair<int,int>(i,j);
    }
  }
}
```

```cpp
/**
 * @brief Function to determine if a board is a solution
    .
 * @param board The state of the board
 * @return True if the board is a solution, false
    otherwise
 */
bool isSolution(const vector<vector<int>>& board){

    int n = board.size();
    bool isSolution = true;

    for(int i=0; i<n && isSolution; i++){
        for(int j=0; j<n && isSolution; j++){
            if(board[i][j] != i*board.size() + j)
                isSolution = false;
        }
    }

    return isSolution;
}
```

```cpp
/**
 * @brief Function to permutate the tiles of the board.
 * @param board The state of the board
 * @param pos_0 The position of the empty tile
 * @param action The action to perform
 * @return The new position of the empty tile
 */
pair<int,int> permutateBoard(vector<vector<int>>& board,
    const pair<int,int>& pos_0, const Action& action){

  int i = pos_0.first;
  int j = pos_0.second;

  switch(action){
    case Action::DOWN:
      if(i!=0){
        swap(board[i][j], board[i-1][j]);
        i=i-1;
      }
      break;
    case Action::UP:
      if(i!=board.size()-1){
        swap(board[i][j], board[i+1][j]);
        i=i+1;
      }
      break;
    case Action::RIGHT:
      if(j!=0){
        swap(board[i][j], board[i][j-1]);
        j=j-1;
      }
      break;
    case Action::LEFT:
      if(j!=board.size()-1){
        swap(board[i][j], board[i][j+1]);
        j=j+1;
      }
      break;
  }

  return pair<int,int>(i,j);
}
```

```
/**
* @brief Function to generate a child node by applying
  an action to the current node.
* @param current_node The current node
* @param action The action to apply
* @return The generated child node
*/
node Apply(const node& n, const Action& action){

    vector<vector<int>> board=n.board;
    int cost = n.cost;
    pair<int,int> pos_0 = n.pos_0;
    int i = pos_0.first;
    int j = pos_0.second;
    list<Action> moves = n.moves;

    if ((action == Action::UP && i != board.size() - 1)
   ||
        (action == Action::DOWN && i != 0) ||
        (action == Action::LEFT && j != board.size() -
   1) ||
        (action == Action::RIGHT && j != 0)) {
        pos_0 = permutateBoard(board, pos_0, action);
        cost++;
        moves.push_back(action);
    }

    return node(board, cost, pos_0, moves);
}
```

```
/**
 * @brief A* algorithm to solve the puzzle.
 * @param board The initial state of the board
 * @return The list of actions to solve the puzzle
 */
list<Action> A_star(const vector<vector<int>>& board){

    node current_node(board, 0, findEmptyTile(board), list<Action>());
    priority_queue<node,vector<node>,Comparer> frontier;
    set<node> explored;
  list<Action> moves;

  bool solutionFound = isSolution(current_node.board);
  frontier.push(current_node);

  while(!solutionFound && !frontier.empty()){
    frontier.pop();
    explored.insert(current_node);

    if(isSolution(current_node.board)) solutionFound = true;

    if(!solutionFound){
      // Generate child UP
      node child_up = Apply(current_node, Action::UP);
      if(explored.find(child_up) == explored.end()) frontier.push(child_up);
    }

    if(!solutionFound){
      // Generate child DOWN
      node child_down = Apply(current_node, Action::DOWN);
      if(explored.find(child_down) == explored.end()) frontier.push(child_down);
    }

    if(!solutionFound){
      // Generate child LEFT
      node child_left = Apply(current_node, Action::LEFT);

```

```
      if(explored.find(child_left) == explored.end()) frontier.push(child_left);
    }

    if(!solutionFound){
      // Generate child RIGHT
      node child_right = Apply(current_node, Action::RIGHT);
      if(explored.find(child_right) == explored.end()) frontier.push(child_right);
    }

    if (!solutionFound and !frontier.empty()){
      current_node = frontier.top();
      while(!frontier.empty() && explored.find(current_node) != explored.end()){
        frontier.pop();
        if(!frontier.empty())
          current_node=frontier.top();
      }
    }

    if(solutionFound) moves=current_node.moves;
  }

    return moves;
}
```

```cpp
/**
 * @brief Function to convert the string to actions.
 * @param action The string to convert
 * @return The action representation of the string
 */
Action stringToAction(char action) {
    switch (action) {
        case 'w': return Action::UP;
        case 's': return Action::DOWN;
        case 'a': return Action::LEFT;
        case 'd': return Action::RIGHT;
    }
}
```

```cpp
/**
 * @brief Function to convert the actions to string.
 * @param action The action to convert
 * @return The string representation of the action
 */
string actionToString(const Action& action) {
    switch (action) {
        case Action::UP:    return "UP";
        case Action::DOWN:  return "DOWN";
        case Action::LEFT:  return "LEFT";
        case Action::RIGHT: return "RIGHT";
    }
}
```

5.3 Main

```
#include "../include/generator.h"
#include "../include/A_star.h"

using namespace std;

int main() {

    int n; //Length of the board

    //Presentation
    cout << "Welcome to the (n^2-1)-puzzle solver!!!!!"
  << endl;

    do{
        cout << "Introduce a board size between " <<
  LOWER_BOUND << " and " << UPPER_BOUND << ": ";
        cin >> n;
    }while(n<LOWER_BOUND || n>UPPER_BOUND);


    //Generate the board
    cout << "You chose the board size: " << n << "x" <<
  n << endl;
    if(n==4) cout << "You will play the famous 15-puzzle
  !!!!" << endl;
    cout << "Your puzzle to solve is: " << endl << endl;


    vector<vector<int>> board;

    //Generate a board that is not a solution
    do{
        board = generateBoard(n);
    }while (isSolution(board));

    pair<int,int> pos_0 = findEmptyTile(board);
    showMatrix(board);




```

```
    int option;
    do{
        cout << "Press 0 to solve the puzzle by yourself or 1 to solve it automatically: ";
        cin >> option;
    }while(option!=0 && option!=1);
    cout << endl;

    //Manual Solver
    if(option == 0){

        char move;

        do{
            cout << "You chose to solve the puzzle by yourself!!!!" << endl;
            cout << "Use the following keys to move the empty tile:" << endl;
            cout << "W: Move the empty tile down" << endl;
            cout << "S: Move the empty tile up" << endl;
            cout << "A: Move the empty tile right" << endl;
            cout << "D: Move the empty tile left" << endl;
            cout << "Q: Quit the game" << endl;
            cout << endl;
            showMatrix(board);

            cout << "Select a key: ";
            cin >> move;
            cout << endl;

            if(move =='w' || move=='a' || move=='s' || move=='d'){
                Action move_s=stringToAction(move);
                pos_0 = permutateBoard(board, pos_0, move_s);
                actionToString(move_s);
            }
            else if (move!='q') cout << "Select a valid key!!!!" << endl << endl;
        }while(!isSolution(board) && move != 'q');
```

```

        //Not playing anymore
        if(move=='q') cout << "You quit the game!!!!" <<
    endl;
        else cout << "The puzzle has been solved!!!!" <<
    endl;
    }

    //Automatic Solver
    else{

        cout << "You chose to solve the puzzle
   automatically!!!!" << endl;
        cout << endl << "Solving the puzzle..." << endl;

        list<Action> moves = A_star(board);

        cout << "The solution is: " << endl << endl;

        showMatrix(board);

        for(auto it = moves.begin(); it != moves.end();
   it++){
            cout << actionToString(*it) << endl;
            pos_0 = permutateBoard(board, pos_0, *it);
            showMatrix(board);
        }

        cout << endl << "The puzzle has been solved!!!!"
    << endl;
    }

    cout << "
   ///////////////////////////////////////////////////
   " << endl;

    return 0;

}
```

6 Conclusions

- How different heuristics can alter the execution time of an algorithm.
- How complex it is to provide a good solution to a problem based on heuristics.
- The need to get familiar with data structures and their particularities.

References

Rafael Cobos. Búsqueda con información. http://arantxa.ii.uam.es/~rcobos/teaching/esp/ia/busqueda-con-informacion.pdf. Accessed: 2024-12-13.

Michael Kim. Puzzle blog. https://michael.kim/blog/puzzle. Accessed: 2024-12-13.

Kociemba. Fifteen solver. https://kociemba.org/themen/fifteen/fifteensolver.html. Accessed: 2024-12-13.

Tristan Penman. N-puzzle demo. https://tristanpenman.com/demos/n-puzzle/. Accessed: 2024-12-13.

Wikipedia contributors. 15 puzzle: Varieties of the 15 puzzle. https://en.wikipedia.org/wiki/15_puzzle#Varieties_of_the_15_puzzle. Accessed: 2024-12-13.