Content:

1. Introduction

2. Problem analysis

3. Solution design

3.1. Solvability of a puzzle

3.1.1. An example of solvable and unsolvable state

3.1.2. Inversion distance

3.2. Searching algorithms

3.2.1. Uninformed search

3.2.1.1. BFS

3.2.1.2. DFS

3.2.1.3. Rejection of BFS and DFS

3.2.2. Informed search

3.2.2.1. Heuristic value and heuristic functions

3.2.2.2. A\* algorithm

3.2.2.3. Iterative deepening A\* (chosen)

3.2.3. Heuristic functions:

3.2.3.1. Run-time heuristic functions

3.2.3.1.1. Misplaced tiles

3.2.3.1.2. Manhattan distance

3.2.3.1.3. Linear conflicts

3.2.3.1.4. Inversion distance

3.2.3.2. Pre-calculated heuristic functions

3.2.3.2.1. Walking distance

3.2.3.2.2. Disjoint pattern database (chosen)

3.2.3.2.3. Dynamic pattern database

4. Implementation

4.1. Overview

4.2. Pattern and cache parser

4.3. Expansions of states

5. Result and discussion

5.1. Testing

5.2. Discussion

6. Conclusions and recommendations

6.1. Conclusions

6.2. Recommendations

7. References

1. Introduction:

The 15-puzzle has been one of the most interesting problems in mathematics and computer science. Despite the simple rules, this game has a lot of possible solutions and a wide range of situations that make it very difficult to solve. There are also alternative version of this game: 8-puzzle, 24-puzzle, … They can all belong to a unified problem called n-puzzle, which is NP-hard and scales significantly with each higher n.

2. Problem analysis:

The task of the problem is to move the blank tile so that all tiles come back to the correct position. The 15-puzzle is NP-hard, therefore it is very hard to find an efficient solution in a short amount of time. In this project’s implementation, the puzzle will be presented by a one-dimensional array for better performance, so the goal can be implemented as:

[1, 2, 3, 4, 5, 6, 7, 9, 10, 11, 12, 13, 14, 15, 0]

The blank tile is a zero, and every time the blank tile moves, the 0 will swap its position with another tile. Moving up with make 0 step back 4 tiles, moving down steps further 4 tiles, moving left steps back 1 tile, and moving right steps further 1 tile. There are also some positions that 0 can not move to specific directions.

3. Solution Design:

3.1. Solvability of a puzzle:

3.1.1. An example of solvable and unsolvable states:

On theory, there are 16! = 20,922,789,888,000 possible states of a 15 puzzle. However, not all states are possible to be solved.

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 |
| 9 | 10 | 11 | 12 |
| 13 | 15 | 14 |  |

This state of Sam Loyd is unsolvable.

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 5 | 9 | 13 |
| 2 | 6 | 10 | 14 |
| 3 | 7 | 11 | 15 |
| 4 | 8 | 12 |  |

This state (also referenced as “devil’s configuration”) is solvable.

Therefore, an algorithm to detect whether a state can be solved is required. For this, a heuristic approach will be discussed briefly, which is inversion distance.

3.1.2. Inversion distance:

A puzzle state can be converted into a one-dimensional array by appending each tile to the array from left to right, up to bottom (blank tile is represented as 0).

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 5 | 9 | 13 |
| 2 | 6 | 10 | 14 |
| 3 | 7 | 11 | 15 |
| 4 | 8 | 12 |  |

This state has the array presentation as following:

[1, 5, 9, 13, 2, 6, 10, 14, 3, 7, 11, 15, 4, 8, 12, 0]

For each tile, its inversion distance is defined as the number of tiles that go after it but have smaller value in the array (0 is excluded from others’ inversion distances and has no inversion distance itself). For example, the inversion distances of all tiles in the upper example:

Tile 1: id = 0

Tile 5: id = 3 (5 goes before 2, 3, 4)

Tile 9: id = 6 (9 goes before 2, 6, 3, 7, 4, 8)

Tile 13: id = 9 (13 goes before 2, 6, 10, 3, 7, 11, 4, 8, 12)

Tile 2: id = 0

Tile 6: id = 2 (6 goes before 3, 4)

Tile 10: id = 4 (10 goes before 3, 7, 4, 8)

Tile 14: id = 6 (14 goes before 3, 7, 11, 4, 8, 12)

Tile 3: id = 0

Tile 7: id = 1 (7 goes before 4)

Tile 11: id = 2 (11 goes before 4, 8)

Tile 15: id = 3 (15 goes before 4, 8, 12)

Tile 4: id = 0

Tile 8: id = 0

Tile 12: id = 0

Consequently, the inversion distance of a state is the sum of inversion distances of its tiles (0 is excluded). For the example, the state will have the inversion distance of 0 + 3 + 6 + 9 + 0 + 2 + 4 + 6 + 0 + 1 + 2 + 3 + 0 + 0 + 0 = 36. Consider n as the side length of the puzzle, here is how the solvability of a state is determined:

- For n being odd: The inversion distance must be divisible by 2.

- For n being even:

* If the inversion is divisible by 2, the blank tile must lie in a row with even ordinal number (start counting from 1).
* If the inversion distance is not divisible by 2, the blank tile must lie in a row with odd ordinal number (start counting from 1).

In the example, the inversion distance is 36 which is divisible by 2, and the blank tile lies in the 4th row (an even ordinal number), so this puzzle is solvable. Looking the at the state of Sam Loyd, the inversion distance is 1 (only 15 goes before 14) but the blank tile is in a row with even ordinal number(4th), hence makes this unsolvable.

3.2. Searching algorithms:

3.2.1. Uninformed search:

3.2.1.1. Breadth-first sreach (BFS):

BFS belongs to the brute-force approaches, which cares about exploring all possible states. To implement BFS, a FIFO (first-in first-out) data structure is required, and here the chosen structure will be a queue. The basic principle of BFS is that each time when a state is dequeued from the queue, all possible successors of that state will be put to the queue, which will make the states sharing the same depth as the removed one the next candidates for the upcoming iteration. This procedure will continue until a solution is found. Here is the pseudo-code of BFS for 15-puzzle:

BFS (Root):

Initialize a Queue with Root

While Queue is not empty:

S = Queue.dequeue()

If S is the goal state, then return the path to S

For each possible successor of S:

If the successor has not been explored, then put it in Queue

If out of loop, alert there is no solution

One can see that BFS can guarantee the shortest solution found; however, one major disadvantage is that this algorithm needs to store all the possible states expanded, which will create a significant burden on memory.

3.2.1.2. Depth-first search (DFS):

DFS is also a member of brute-force search algorithms. The implementation of DFS will need a LIFO data-structure (last-in first-out), so Stack will be chosen for carrying out the algorithm. The basic principle of DFS is that each time a state is popped out from the Stack, all its possible successors will be put into the Stack resulting in these expansions will be the next candidates for the next iteration. This procedure will continue until a solution is found. Here is the pseudo-code of DFS for 15-puzzle:

DFS (Root):

Initialize a Stack with Root

While Root is not empty:

S = Root.pop()

If S is the goal state, return the path to S

For each possible successor of S:

If S has not been explored, the put it in Stack

If out of loop, alert that there is no solution

While DFS may assure to keep the memory taken at a reasonable cost, the time to find the solution can increase dramatically if the states that lead to much longer solutions or a dead-end is put at the top of the stack after iterations.

3.2.1.3. Rejection of BFS and DFS:

Both BFS and DFS require a massive processing power or memory since they all treat the states at the same, no state is better than the other since no additional information about them is provided (uninformed), so all possible states must be tried out to deliver a solution. These searching algorithms may seem reasonable for the smaller version of 15-puzzle, which is 8-puzzle having 9! = 362,880 possible states; however, with the monstrous theoretical number of the former, BFS and DFS are unquestionably shown to be inferior and inefficient.

3.2.2. Informed search:

3.2.2.1. Heuristic value and heuristic functions:

Heuristic value is a measurement to judge how close the current state is to the goal. The heuristic value of the goal is 0 and the lesser the heuristic value, the better that state is compared to others and the closer it is to the goal. Consequently, a heuristic function is a procedure that takes a state as the input and evaluates the heuristic value of that state. The heuristic functions will make a crucial improvement over uninformed search as they provide additional information about the states and help to choose the best ones to investigate further.

3.2.2.2. A\* algorithm:

A\* (called “a star”) algorithm is developed by Peter Hart in 1968. This idea of this algorithm is to find the states with best evaluated values (symbolized as “f”) and focuses on exploring them first. To implement A\*, a min priority queue is required. A min priority queue will always push the items with the lowest values deduced from comparator to the top, which ensures the next enqueuing will investigate the best states. The f value of each state is the sum of the result of heuristic function taking it as an input and the depth of the state in the expansion tree. The lower the f value, the better the state compared to others. Here is the pseudo-code of A\*:

A\* (Root):

Initialize a Path with Root

Initialize a PriorityQueue with Root, with a comparator of f value

While PriorityQueue is not empty:

S = PriorityQueue.dequeue()

If S is the goal, return the path to S

If S has been explored:

Replace S in the path if the depth of S is lower than that of explored one

Continue (to skip this iteration)

Else:

Add S to path to goal

For all possible successor of S:

Put the successor in PriorityQueue

If out of loop, alert there is no solution

A major upgrade can be seen from A\* compare to BFS and DFS is that it only expands the states with the lowest f values leaving the worse states to the last to explore, therefore not only decrease the expected time to reach the goal but also occupies less memory. However, despite those great improvements, the A\* algorithm still causes a massive burden on memory because of the theoretical number of possible states in 15-puzzle problem.

3.2.2.3. Iterative Deepening A\* (IDA\*):

IDA\* was first mentioned by Richard Korf in 1985. IDA\* works nearly identically as A\*; however, this algorithm introduces a concept of bound, a number that when the f of current investigated state is larger than it, the algorithm will remove the state from path and revert to the previous candidates. With this feature, IDA\* despite being slower than A\* (by a small margin) saves a lot of memory since it will reconstruct the path instead of saving everything in memory. Here is the pseudo-code of IDA\* with two functions search and ida\_star:

search (Path, g, Bound):

S = Path.peek()

f = g + h(S) // g refers to the depth, h is the heuristic function

If f > Bound, then return f

If S is goal, then return True

min = inf

For each successor of S:

If successor is not in Path:

Put successor in Path

t = search (path, g + 1, bound) // The cost of moving is 1

If t = True, then return True

If t < min, then return t

Return min

ida\_star (Root):

Initialize Bound with h(Root)

Initialize Path as a Stack with Root

While True:

t = search (Path, 0, Bound)

If t = True, return Path

If t = inf, alert there is no solution

Bound = t

For the memory constraint, the chosen searching algorithm in this puzzle solver’s implementation will be IDA\*.

3.2.3. Heuristic functions:

3.2.3.1. Run-time heuristic functions:

In-time heuristic function is a function that will evaluate the heuristic value of a state at the run-time. The following will be introduced to the readers: Misplaced tiles, Manhattan distance, linear conflicts, and inversion distance, which has been mentioned earlier.

3.2.3.1.1. Misplaced tiles:

Misplaced tiles is a very simple heuristic which only counts the number of tiles that are not in correct position. For example:

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 5 | 9 | 13 |
| 2 | 6 | 10 | 14 |
| 3 | 7 | 11 | 15 |
| 4 | 8 | 12 |  |

This state compared to the goal:

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 |
| 9 | 10 | 11 | 12 |
| 13 | 14 | 15 |  |

will have h = 12 (as 5, 9, 13, 2, 10, 14, 3, 7, 15, 4, 8, 12 are placed wrong)

However, the heuristic value of this approach is too low, too many states will share the same heuristic number, therefore it is not plausible to use this as our heursitic function.

3.2.3.1.2. Manhattan distance:

Manhattan distance can be understood as the shortest path of a tile to its correct position assuming there is no tile blocking its path. For example:

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | C |
|  |  |  |  |
|  |  |  |  |
| 4 |  |  |  |

As you can see, the C represents the correct position of the tile 4. The shortest path for 4 to reach its goal is 6 (3 up, 3 right).

The Manhattan distance of a state is the sum of all the Manhattan distances of all tiles. For example:

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 5 | 9 | 13 |
| 2 | 6 | 10 | 14 |
| 3 | 7 | 11 | 15 |
| 4 | 8 | 12 |  |

h (5) = 2 (1 down, 1 left)

h (9) = 4 (2 down, 2 left)

h (13) = 6 (3 down, 3 left)

h (2) = 2 (1 up, 1 right)

h (10) = 2 (1 right, 1 down)

h (14) = 4 (2 down, 2 left)

h (3) = 4 (2 right, 2 up)

h (7) = 2 (1 up, 1 right)

h (15) = 2 (1 down, 1 left)

h (4) = 6 (3 right, 3 up)

h (8) = 4 (2 up, 2 right)

h (12) = 2 (1 up, 1 right)

Therefore, the Manhattan distance for this tile is 40 as sum of all h. However, the Manhattan distance is not yet a good enough heuristic function to implement.

3.2.3.1.3. Linear conflicts:

Linear conflict is an improvement added to Manhattan distance as it concerns the blocking interactions between tiles belonging to the same line (can either be row or column). Proposed by Othar Hansson in 1992, he and his group defined that linear conflicts happens when there are two tiles i and j in the correct line, i is to the left of j and the goal position of i is to the right of j. Here is an illustration:

The goal position of tile 5

The goal position of tile 8

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
| 8 |  | 5 |  |
|  |  |  |  |
|  |  |  |  |

Tile 8 and 5 are all in the correct row; tile 8 is to the left of tile 5 and the goal position of tile 8 is to the right of the goal position of tile 5, in which tile 8 currently stands. In this situation, one can either remove tile 5 one step up or down for 8 to move to its correct position, or the other way around to resolve this linear conflict. The linear conflicts of lines is how many tiles in minimum must be removed in order to resolve all the linear conflicts. The linear conflicts of a state is the sum of all linear conflicts of all rows and columns. The sum of Manhattan distance and linear conflicts of a state is a decent heuristic value; however, it is not good enough compared to the wide range of states of 15-puzzle.

3.2.3.1.4. Inversion distance:

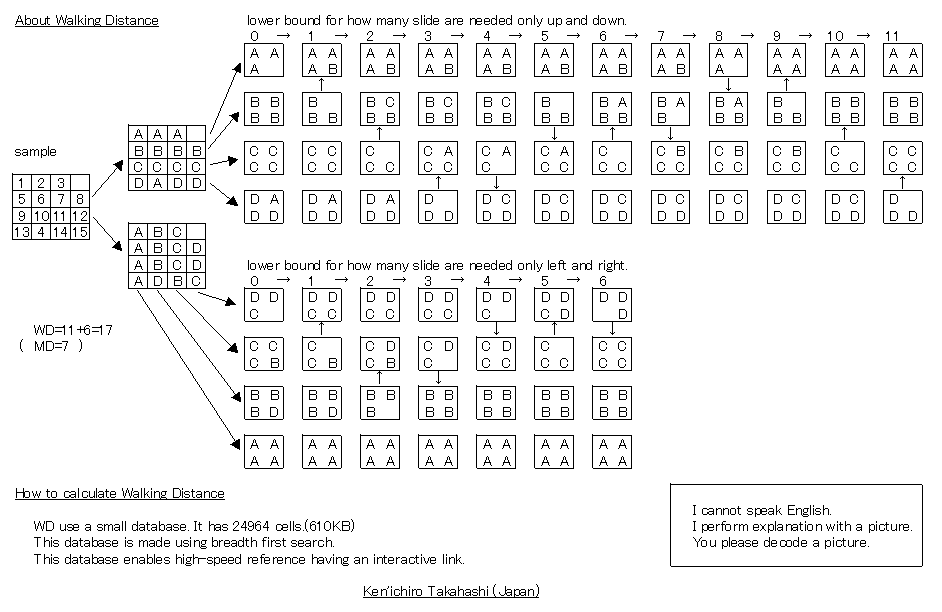
This heuristic approach is created by a Japanese engineer, Kenichiro Takahashi. The inversion distance is usually paired with Manhattan distance and the maximum of two will be chosen as the heuristic value of the state. Like previous heuristic functions, this can not be applied to 15-puzzle if a fast solution is required.

3.2.3.2. Pre-calculated heuristic functions:

Pre-calculated heuristic functions will first calculate all relevant states with their heuristic values and save them into a file. Later, when evaluate a state, it will look up that state in the file and return the pre-calculated heuristic value. This is better than run-time because look up a large set is more efficient in time compared to calculated in time of running and will take up significantly less memory.

3.2.3.2.1. Walking distance:

This technique is also developed by Kenichiro Takahashi. Walking distance will perform vertical and horizontal divisions. For horizontal division, the state is divided into rows, and heuristic value for this division is the minimum of steps for the blank tile to move so that all tiles are in their correct rows regardless of their order in that row. The same applies for vertical division but the state is divided into columns and the tiles must be in correct columns as the end. The heuristic value of a state is the sum from both divisions. Here is a picture from the author illustrating the algorithm:



From the goal state, we can implement a BFS search to generate all situations. There are total 24964 distinct entries to store, and the maximum walking distance value is 70. However, this only works for simple to below average states in terms of difficulty. This heuristic function is proven not to work effectively with hard puzzles:

|  |  |  |  |
| --- | --- | --- | --- |
|  | 11 | 6 | 13 |
| 15 | 3 | 7 | 9 |
| 12 | 5 | 4 | 2 |
| 1 | 8 | 10 | 14 |

An average-difficult state:

Time taken: 1h19m55s, steps: 63

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 5 | 9 | 13 |
| 2 | 6 | 10 | 14 |
| 3 | 7 | 11 | 15 |
| 4 | 8 | 12 |  |

The devil’s configuration (as mentioned and measured by author):

Time taken: About 21h33m, steps: 72

3.2.3.2.2. Disjoint (statically-additive) pattern database:

This approach devides the state into distict component parts called patterns which contain not overlaped tiles. The heuristic value will the sum of minumum steps the blank tile must move for the patterns to reach their goal positions. Here is an example:

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 2 | 3 | 4 |
| - | - | - | - |
| - | - | 0 | - |
| - | - | - | - |

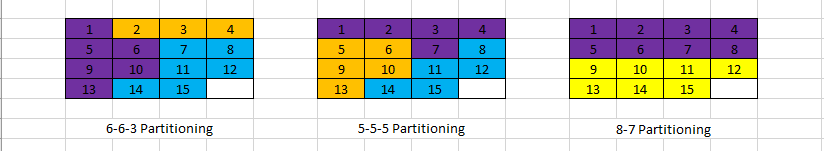
There is one pattern: A row of 1, 2, 3, 4, the symbol ‘-‘ indicates that this tile is irrelevant to this pattern. This is a complete pattern, so the heuristic value is 0. The 0 represents the tile that moves, and since it is irrelevant to the pattern, for instance, this state also share the same value with the mentioned one:

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 2 | 3 | 4 |
| 0 | - | - | - |
| - | - | - | - |
| - | - | - | - |

The pattern still holds, so the heuristic value of this is still 0. Assume there is one situation:

|  |  |  |  |
| --- | --- | --- | --- |
| 0 | 2 | 3 | 4 |
| 1 | - | - | - |
| - | - | - | - |
| - | - | - | - |

The pattern does not hold anymore, so 0 can just move down to complete the pattern, and therefore the minmum step is 1, thus the heuristic value of this state is 1. The author, Richard Korf, mentioned 3 possible pattern divisions:



Assume the purple tiles are in pattern A, yellow ones are in pattern B, blue ones are in pattern C, here are the statistics on generating all configurations (“take” means insert into database):

- The general formulas, consider n is the number of tiles for each pattern:

* Examine: 16Pn + 1
* Take: 16Pn

- For 6-6-3 partitioning:

* A and C: Examine 57657600 states, take 5765760 entries.
* B: Examine 43680 states, take 3360 entries.

- For 5-5-5 partitioning:

* A, B, and C: Examine 5767760 states, take 524160 entries.

- For 8-7 partitioning:

* A: Examine 4151347200 states, take 518918400 entries.
* B: Examine 518918400 states, take 57657600 entries.

Despite the 8-7 and 6-6-3 partitioning may yeild better heuristic values, due to hardware limitations, the 5-5-5 partitioning is chosen to implement this puzzle solver.

3.2.3.2.3. Dynamic (dynamically-additive) pattern database:

While the disjoint version focuses on dividing the state into seperated patterns, the dynamic version will generate all possible states for each pair of tiles instead of patterns. For example, consider (x, y) is all possible states for tile x and tile y, then these are entries will be saved: (1,2), (1,3), …, (2, 3), … The author, Ariel Felner, also includes triple (x, y, z) and quadruple (x, y, z, t). The dynamic pattern database is better at scaling problems such as 24-puzzle; however, performing this would require very large three-dimensional databases, and its power would be redundant for this project.

4. Implementation:

4.1. Overview:

- Approach: IDA\* + Disjoint pattern database 5-5-5.

- Use 3 json files as look up materials: pdA, pdB, pdC.

4.2. Pattern and cache parser:

In order to look up in the database, a parser is needed to convert the state into the pattern string. For this, if a pattern is divided into relevant tiles and irrelevant tiles, then reading from left to right, up to bottom, this is the form of pattern string:

[relevant tile]{[‘i’ + number of irrelevant tiles in between][‘.’ + relevant tile]}

The relevant tile will be followed by a dot if it does not stand first in the string, and the number of irrelevant tiles is proceeded by the letter ‘i’.

For example:

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | - | - | - |
| 0 | 4 | - | - |
| - | 2 | - | - |
| - | - | 3 | - |

Pattern string: 1i4.4i3.2.i4.3i1

Read as: 1 🡪 4 irrelevant tiles 🡪 4 🡪 3 irrelevant tiles 🡪 2 🡪 4 irrelevant tiles 🡪 3 🡪 1 irrelevant tile

|  |  |  |  |
| --- | --- | --- | --- |
| - | - | 3 | 4 |
| - | - | - | - |
| - | 1 | - | 2 |
| - | - | 0 | - |

Pattern string: i2.3.4i5.1i1.2i4

Read as: 2 irrelevant tiles 🡪 3, 4 🡪 5 irrelevant tiles 🡪 1 🡪 1 irrelevant tile 🡪 2 🡪 4 irrelevant tiles

In the procedure of generating the database, the pattern string must count 0 as a part of relevant tiles in order to avoid duplicates. This string having 0 is seperated from pattern string and is called cache string. Two states may share the same pattern string but must have different cache strings (however, the look up only concerns the pattern string). For example:

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 2 | - | - |
| 0 | - | - | - |
| - | - | 3 | 4 |
| - | - | - | - |

Pattern string: 1.2i8.3.4i4

Cache string: 1.2i2.0i5.3.4i4

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 2 | - | - |
| - | - | - | - |
| - | - | 3 | 4 |
| - | 0 | - | - |

Patern string: 1.2i8.3.4i4

Cache string: 1.2i8.3.4i1.0i2

This implementation is reflected in this line of code:

In \_to\_cs function in file charge\_pd.py (0 is treated the same as re – relevant tiles):

for tile in data:  
 if tile in re or tile == 0:

However, to\_ps function in both charge\_pd.py does not include 0:

for tile in data:  
 if tile in re:

\_to\_cs is shorten for ‘to cache string’, to\_ps is shorten for ‘to pattern string’

If a pattern string is encountered the second time while generating, then the heuristic value will not be updated, since the first met entry will always have the lowest value. Moreover, this generator use BFS to go through all states and implement recursion with a parameter g presenting depth (or heuristic value in this case), the value of g is not increased after each call of recusion, but only increased once in a recursive call if and only if there are new entries added to the pattern string dictionary (The pattern string is saved in a dictionary like {pattern string 1: g1, pattern string 2: g2, …}). Here is a snippet code for moving up in charge\_pd.py

# Condition to go up : z not in {0, 1, 2, 3} -> z > 3  
if z > 3:  
 up = deepcopy(data)  
  
 up[z] = up[z - 4]  
 up[z - 4] = 0  
  
 up\_cs = \_to\_cs(up)  
  
 # Check cache set: If exists then do nothing, if not than add to set and queue, check dict  
 if up\_cs not in cache\_set:  
 cache\_set.add(up\_cs)  
 pass\_queue.put(up)  
  
 # Check dictionary: If exist then do nothing, if no than add to dict  
 up\_ps = to\_ps(up)  
 if up\_ps not in pattern\_dict:  
 if not g\_increased:  
 g += 1  
 g\_increased = True  
 pattern\_dict[up\_ps] = g

If the cache string of up does not exist in cache set, then it will be put into the queue, and this cache string will be included in the set. Next, the pattern string is checked, the entry will only be added to the pattern string dictionary if it has not existed. The g\_increased variable also ensures that g is only increased once through out all the states of the same depth.

4.3. Expansion of states:

To check if a state has been visited, a set is implemented. The key form for all states will be a string of all tiles joined by a dot. For example, the goal state will have the following key: “1.2.3.4.5.6.7.8.9.10.11.12.13.14.15.0”. In original IDA\* implementation, there will be a function that return a list of successors from states, then this will be looped through, each successor will be checked if it has been processed before (by using set). However, the expand function can include another variable called ‘keys’, which is a set of keys of states have been visited. Since the expand function will return a list, then only successors whose keys are not in the ‘keys’ set are appended to the list, thus lower the occupied memory. The moving operations of blank tiles can be simplified as so (z is the index of 0 in this array, z’ is the new index):

- Move up: z not in {0, 1, 2, 3} 🡪 z > 3, z’ = z – 4

- Move down: z not in {12, 13, 14, 15} 🡪 z < 12, z’ = z + 4

- Move left: z not in {0, 4, 8, 12} 🡪 z % 4 != 0, z’ = z – 1

- Move right: z not in {3, 7, 11, 15} 🡪 z % 4 != 3, z’ = z + 1

This implementation is relfected in this snippet of code in function “expands” in ops.py:

def expands(data, keys):  
 z = find\_zero(data)  
  
 mlist = []  
  
 if z > 3: # Can not move up if z in {0, 1, 2, 3} -> z > 3  
 up = deepcopy(data)  
  
 up[z] = up[z - 4]  
 up[z - 4] = 0  
  
 if get\_key(up) not in keys:  
 mlist.append(up)

As you can see, the expand function requires another paratemer which is keys and returns a list of successors of data (state) (which is mlist in this situation). The deep copy will only be made if z > 3 meaning z can move up, and up will not be appended to the list if the key exists in the set. Here is the get\_key implementation in ops.py:

def get\_key(data):  
 return ".".join(map(str, data))

5. Result and discussion:

5.1. Testing:

11 puzzles are tested with some input from an online generator: [15 puzzle (jaapsch.net)](https://www.jaapsch.net/puzzles/javascript/fifteenj.htm)

Here are the results:

|  |  |  |
| --- | --- | --- |
| State | Steps | Time (seconds) |
| 1, 5, 9, 13, 2, 6, 10, 14, 3, 7, 11, 15, 4, 8, 12, 0 | 100 | 0.06 |
| 9, 7, 5, 6, 8, 1, 14, 0, 11, 2, 10, 3, 4, 12, 15, 13 | 90 | 0.03 |
| 0, 11, 6, 13, 15, 7, 9, 12, 5, 4, 2, 1, 8, 10, 14 | 98 | 0.05 |
| 0, 1, 3, 9, 13, 10, 11, 7, 8, 14, 2, 12, 6, 5, 4, 15 | 62 | 0.02 |
| 13, 12, 8, 7, 2, 1, 6, 14, 3, 9, 10, 11, 5, 0, 15, 4 | 80 | 0.03 |
| 1, 2, 14, 3, 9, 6, 0, 13, 5, 11, 12, 10, 15, 4, 7, 8 | 65 | 0.03 |
| 5, 7, 0, 10, 11, 3, 6, 4, 12, 1, 9, 13, 8, 2, 14, 15 | 74 | 0.05 |
| 6, 11, 14, 9, 3, 2, 12, 8, 13, 4, 1, 7, 5, 10, 15, 0 | 84 | 0.05 |
| 2, 1, 12, 13, 8, 4, 15, 5, 6, 3, 7, 11, 9, 14, 10, 0 | 74 | 0.05 |
| 6, 2, 7, 11, 14, 10, 15, 5, 12, 8, 13, 9, 3, 4, 8, 1 | 95 | 0.04 |
| 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 0 | Unsolvable | Unsolvable |

5.2. Discussion:

The pattern database can solve a state in an incredibly short time. For devil’s configuration (the first test), this approach only took 0.06 seconds compare to over 22 hours of walking distance. For the third test (a medium level that has been mentioned before), the pattern database only needed 0.05 seconds compared to over 1 hour of walking distance. However, one must see that this solution generates more steps in between. For the devil’s configuration, the optimal solution would be 77 compared to 100 of pattern database, and for the third test, 98 steps were generated compared to only 63 of walking distance. Furthermore, Kenichiro Takahashi also mentioned that all states can be solve within the bound of 80 moves. Since the IDA\* uses the heuristic function to determine which is the best state to continue, for improvement, one may choose to implement another disjoint partitioning or the dynamic version for better heuristic value, which may result in smaller step number.

6. Conclusions and recommendations:

6.1. Conclusions:

This puzzle solver can solve a state in a very short amount of time (less than 0.1 seconds); however, it does not bring an optimal solution, but rather sub-optimal at best since there are many redundant steps in between. In conclusion, time is a trade off for optimal solutions in this case.

6.2. Recommendations:

If the limitation of hardware is not a concern, one can try and implement another disjoint partitioning or the dynamic pattern database in order to get a more optimal solution. Additionally, A\* algorithm can also be used in place of IDA\* since this searching style will explore all the best states as soon as possible thanks to the min priority queue.

7. References:

- Solvability of a puzzle: [n-puzzle : Tìm hiểu về cách giải bài toán (aptech.vn)](https://aptech.vn/kien-thuc-tin-hoc/n-puzzle-tim-hieu-ve-cach-giai-bai-toan.html)

- Misplaced tiles (Hamming distance) and Manhattan distance: [8-Puzzle Programming Assignment (princeton.edu)](https://www.cs.princeton.edu/courses/archive/spring18/cos226/assignments/8puzzle/index.html)

- Linear conflicts: [Criticizing solutions to relaxed models yields powerful admissible heuristics - ScienceDirect](https://www.sciencedirect.com/science/article/abs/pii/002002559290070O)

- Inversion distance and Walking distance: [Juro Pöllö :: STP Corner (archive.org)](https://web.archive.org/web/20141224035932/http:/juropollo.xe0.ru/stp_wd_translation_en.htm)

- Disjoint pattern database: [Disjoint pattern database heuristics - ScienceDirect](https://www.sciencedirect.com/science/article/pii/S0004370201000923)

- Dynamic pattern database: [Additive pattern database heuristics | Journal of Artificial Intelligence Research (acm.org)](https://dl.acm.org/doi/10.5555/1622487.1622496)

- A\* algorithm: [A\* search algorithm - Wikipedia](https://en.wikipedia.org/wiki/A*_search_algorithm)

- IDA\*: [Iterative deepening A\* - Wikipedia](https://en.wikipedia.org/wiki/Iterative_deepening_A*)