A replication of the work of "Disjoint pattern database heuristics" paper on 24-puzzle

Reza Ghanbari

Introduction

The 24-puzzle, also known as the 5x5 sliding tile puzzle, is a classic combinatorial optimization problem that has been studied extensively in the search community. The goal of the puzzle is to rearrange a set of 24 tiles, each labeled with a number from 1 to 24, into a specific configuration by sliding them around on a 5x5 grid. The puzzle is notorious for its complexity, with almost 10^25 possible states (Korf and Felner 2002), making it a challenging problem for search algorithms to solve.

One technique that has been used successfully in solving the 24-puzzle is the use of pattern databases (Culberson and Schaeffer 1998). A Pattern database (PDB) is a precomputed table that stores the optimal distances between a set of pattern states and the goal state in the abstract state space. The idea behind PDBs is to use them as a heuristic function to guide the search process towards the goal state by estimating the distance from the current state to the goal state.

The effectiveness of PDBs depends on the choice of patterns used to create the PDBs. Generally, the more patterns used, the better the performance of search, as different PDBs capture the cost of moving tiles in more cases and generate a better estimation of the true cost. However, as the number of PDBs increases, the amount of memory needed to keep the PDBs becomes so large that they become impractical to use. Moreover, One important thing to note about PDBs is that the values in the tables cannot be simply added together. This is because the heuristic values from overlapping patterns can lead to an overestimation of the distance from the current state to the goal state. This is known as the double counting problem, and it can lead to suboptimal solutions or even failure to find a solution in some cases. This is where the idea of disjoint pattern databases (Korf and Felner 2002) comes into play.

Disjoint pattern databases (DPDBs) are a technique used for solving sliding-tile puzzles. DPDBs involve partitioning the tiles into groups that do not overlap, such that no tile belongs to more than one group. For each group, precomputed tables are generated, which store the minimum number of moves required to solve the tiles in that group. Each of these tables is referred to as a disjoint pattern database. During the search process, an index is computed for each group based on the current positions of the tiles, and the corresponding value is retrieved from the table. These values are then added together to obtain an overall heuristic value for the current state. It is safe to add the values because of the fact that during generating each table only the movement of tiles involved in that pattern is counted, and the patterns are disjoint, so the double counting problem is not an issue in this case. In addition, by using DPDBs, the size of the pattern databases can be significantly reduced, while maintaining the same level of performance.

In summary, pattern databases and disjoint pattern databases are powerful techniques for solving combinatorial optimization problems such as the sliding tile puzzle. The use of DPDBs can significantly reduce the size of the PDBs required to solve large problems and also generate a better estimation of the cost. One of the papers that has investigated this approach is (Korf and Felner 2002) which applies this method on 15 and 24-puzzles.

In this report, we re-implement the work of (Korf and Felner 2002) on the use of DPDBs in solving the 24-puzzle and replicate some of the experiments they conduct on 24-puzzle by solving some of the instances mentioned in the paper.

The rest of this report is organized as follows. In the next section, we provide an overview of related work in the area of PDBs and DPDBs. Then, we describe the search algorithm and the process of PDB creation. After that we go through some implementation details, and finally, we present our experimental results and analysis.

Related Work

The sliding tile puzzle is an combinatorial puzzle that fascinated many researchers in this field. There are several approaches implemented to solve different variations of the puzzle. For example the eight puzzle can be easily solved using breadth-first search, because of its small state space (Korf and Felner 2002). However, coping with other variations like fifteen and 24 puzzle is not as trivial.

The use of pattern databases (PDBs) for solving combinatorial optimization problems such as the sliding tile puzzle has been extensively studied in the literature. One of the earliest works on PDBs was proposed by (Culberson and Schaeffer 1998), where they introduced the idea of PDBs for the fifteen puzzle. They selected a subset of tiles with the size

of 8 and the blank to generate a pattern database that stores the move needed to set the pattern tiles into their goal position. They also take into consideration the intractions of blank with other tiles to reach the goal; in other words, the values that are stored in the pattern database also contain the number of movements of the blank tile even when the pattern tiles are not moved. To find this value, they apply a backward breadth-first search from the goal setting to find the value of each entry in the PDB, and then run IDA* with that PDB to solve the puzzle. This way, in the fifteen puzzle, they reduced the number of generated nodes by a factor of 1038 (Culberson and Schaeffer 1998). They have also apllied PDB in the game of checkers to show the generality of this approach.

The use of heuristic search algorithms, such as A* and IDA*, is necessary to solve the Fifteen Puzzle optimally. A* is a best-first search algorithm that guarantees finding the shortest solution if the heuristic function is admissible. However, A* may not solve random instances of the sliding tile Puzzle as the size of puzzle increases due to its memory requirements. IDA* (Korf 1985) is a linear-space version of A* that guarantees an optimal solution if the heuristic function is admissible. it performs a series of depth-first search and pruns the tree when the nodes' cost cross a threshold that is set up in each iteration, similar to DFID. Since it only stores the actions in the path that it is currently searching, it does not exhaust the memory like A*. IDA* with the Manhattan distance heuristic was the first algorithm to find optimal solutions to random instances of the Fifteen Puzzle, generating an average of about 400 million nodes per problem (Korf 1985).

Korf has also contributed significantly to the area of parallel search, including the use of parallel breadth-first search in large-scale problems. In that work, Korf suggests a linear implementation of finding the rank of a state to use as index in the pattern database (Korf and Schultze 2005).

The first paper that have investigated solving 24-puzzle instances is (Korf and Taylor 1996). It uses a complex heuristic and a pruning approach based on finite-state machines to prune the duplicate nodes. After that, the work that this report is focusing on (Korf and Felner 2002) is focusing on the same problem and utilizes the idea of disjoint pattern databases to solve random instances of 24-puzzle.

Overall, the literature on PDBs and DPDBs is extensive and spans several decades. The works of Culberson and Schaeffer, Korf, and others have contributed significantly to the development and application of these techniques in solving combinatorial optimization problems, including the sliding tile puzzle.

Methodology

In this section, we will explain the design decisions we have made to implement the problem, and the search algorithm. First, state representation and the pattern database creation are discussed, and then heuristic calculation is explained, and finally the data structures used in PDB creation is briefly mentioned.

We represent the state of the puzzle by two arrays, one representing the grid of the puzzle and the position of tiles in that grid, and the other is the dual of the state. The dual is an array that for each tile, it stores the position at which the tile is located in the state array. The state of the puzzle is represented as a one-dimensional array in our implementation, and the tiles are located in that array from top to bottom and from left to right. Figure 1 shows an example of the state representation for a simple 2×2 puzzle. The dual is important as it is used to generate the index for the pattern databases, as well as keep track of the blank.

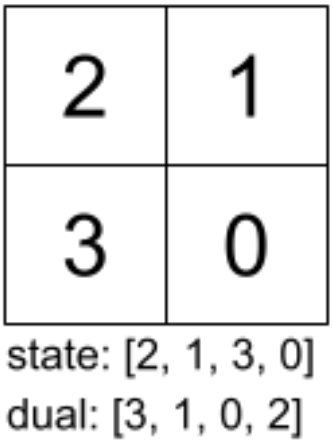

Figure 1: An example of representation of the problem's state and its dual as arrays

As we are replicating the work of (Korf and Felner 2002), we use the same pattern database structure that is used in that paper. For 24 puzzle, it implements four disjoint pattern databases, each of them containing six elements. Also as it is a disjoint table, it does not consider the location of blank in any of the patterns, and the patterns cover all the other tiles.

Figure 2 show how the paper defines the patterns for 24 puzzle. For this puzzle, they use four non-overlapping patterns, three of them having a 2×3 rectangular shape, and an irregularly shaped pattern that surrounds the blank tile. As each of these patterns cover 6 tiles and there are 25 positions in the puzzle in total, the total number of permutations of the tiles in each pattern would be ${}_{25}P_6 = 127,512,000$. So for each pattern we should keep track of 127,512,000 entries.

Although we have four different patterns in this puzzle, it is not required to generate the PDB for all of those four patterns, as we can reuse the PDB of one of the rectangular patterns for the other rectangular ones as well, due to the symmetry of the puzzle. Considering the symmetry, we only generate PDB for the irregular pattern and the rectangular pattern in the top right of the puzzle, and then map all the other rectangular ones into that pattern. As a result, by considering one byte per entry, we only need around 255 megabytes ($2 \times 127,512,000$ bytes) to store the PDBs for 24 puzzle.

To efficiently implement the PDBs, we keep them in plain array and use an indexing mechanism that maps each permutation of states into a unique integer ranging from 0 to

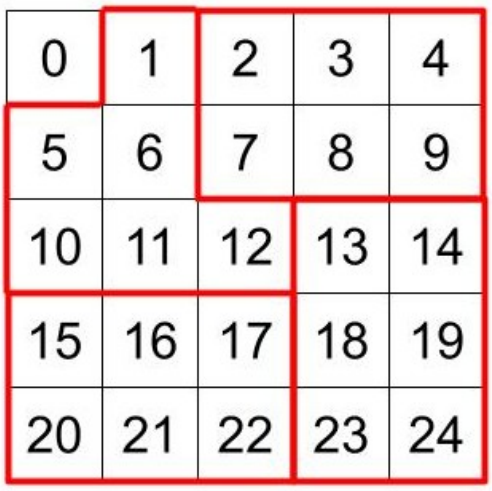

Figure 2: patterns in the 24 puzzle. Each shape with red boundaries determine a pattern containing the tiles inside the boundaries

$_{25}P_6$. To achieve the index, we use a mapping from the permutation to its index in a lexicographic ordering of all such permutations. We use the algorithm that is proposed in (Korf and Schultze 2005) and it is shown in algorithm 1. It basically calculates the factorial base equivalent of the permutation index and then converts it into its decimal value. To perform the approach in linear time, we should pre-compute a table that contains the necessary factorials for the process of converting the factorial base representation into a decimal number and also another table that stores the number of ones in binary representation of each number ranging from 0 to $2^{25}-1$. The latter table is used for when we want to find how many numbers before a given number in the permutation were smaller than that given number, which is needed to find the correct index in lexicographic ordering. In the algorithm, this table is called *numOnes*. To keep track of the numbers that we have seen while traversing the permutation, the algorithm keeps a binary representation called *seen*. In each iteration, it is right-shifted enough to extract a portion that refers to the numbers smaller than the current element of permutation, and is used as an index to the *numOnes* table. This way we count the number of elements that we have seen and were smaller than the current element.

Algorithm 1 Ranking calculation algorithm

procedure RANK(d, n, k) ▷ d is the permutation
  $seen \leftarrow 0$
  $rank \leftarrow 0$
  for $i = 0$ **to** k **do**
    $seen \leftarrow seen \vee 1 << (n - d_i - 1)$
    $temp \leftarrow d_i - numOnes[seen >> (n - d_i)]$
    $rank \leftarrow rank + temp \times_{(n-1-i)} P_{(k-1-i)}$
  end for
  return $rank$
end procedure

To generate the PDBs, we consider the tiles in each pattern and the position of the blank. We only keep track of the blank position to find the true heuristic value of each permutation, and we don't encode blank into the PDB; in other words, we use the blank to generate states in the correct order, but we don't count the interactions of blank with non-pattern tiles in our heuristic, and we calculate the cost of each permutation by only considering the movement of pattern tiles. if we don't consider the blank position in the node generation, we would generate the nodes in a wrong order, as we are practically considering all the non-pattern tiles to be blank by the movement of pattern tiles, and without regarding the blank position. This approach would generate a PDB that underestimate the cost of some permutations. Figure 3 demonstrates this issue in a real setting. The grid (a) shows the case in which we do not consider the blank position and (b) shows the case in which PDB is generated regarding the blank position. As we can see, the heuristic estimation of the (a) would be one as we have moved only a tile from the goal permutation, but (b) shows that we should do a considerable amount of work to take the blank out of the top left corner and generate the similar permutation as the goal and finally swap tile 10 with blank, which is the correct behaviour in the real case. So considering the blank position would capture movements that would be ignored if we do not consider the blank position. This happens whenever the grid is partitioned by the tiles of the pattern, like figure 3.

0	1	0	0	0
5	6	0	0	0
0	11	12	0	0
10	0	0	0	0
0	0	0	0	0

(a)

*	1	*	*	*
5	6	*	*	*
0	11	12	*	*
10	*	*	*	*
*	*	*	*	*

(b)

Figure 3: An example of a state whose cost would be different of we do not keep track of blank position

Generating the PDB without considering the blank would be faster, as during node generation we can move any tile in the pattern that is blocked by the other tiles of the pattern, but if we consider the blank, we should also generate the nodes in which the blank swaps with the tiles that are not part of the pattern. However, the latter PDB is more precise according to the reasons discussed above. According to our experiments, without considering the blank, the PDB is generated in about 1 minute with the values of PDB ranging from 0 to 32 for the irregular case, which is smaller than the results mentioned in the paper - the maximum value of irregular PDB in the paper is 34.

To create each PDB, we run a breadth-first search on the abstract state space of each pattern starting from the goal state, to generate all the different entries. An important matter in this case is the states in which the blank swaps the position with non-pattern tiles. In these case we do not add the value of heuristic.

Algorithm 2 Iterative deepening search algorithm

```
procedure IDA*(state)
    expandedNodes ← 0;
    generatedNodes ← 0;
    limit ← heuristic(state);
    blank ← state.getBlank()
    path ← [blank];
    while true do
        result ← ITERATE(state, limit, blank, path);
        if result == 0 then
            return path;
        end if
        limit ← result;
    end while
end procedure

procedure ITERATE(state, limit, previousBlank, path)
    expandedNodes ← expandedNodes + 1;
    if state.isGoal() then
        return 0                    ▷ 0 means the goal is found
    end if
    minCost ← ∞
    currentBlank ← state.getBlank()
    for neighbor in neighbors(currentBlank) do
        if neighbor = previousBlank then
            continue
        end if
        generatedNodes ← generatedNodes + 1
        path.append(neighbor)
        state.moveBlankTo(neighbor)
        gCost ← path.size() − 1
                ▷ g is actually the depth of current path
        childFCost ← heuristic(state) + gCost
        if childFCost > limit then
            minCost ← min(minCost, childFCost)
            state.moveBlankTo(currentBlank)
            path.removeLast()
            continue
        end if
        result ← ITERATE(state, limit, currentBlank, path)
        if result == 0 then
            return 0
        end if
        minCost ← min(minCost, result)
        state.moveBlankTo(currentBlank)
        path.removeLast()
    end for
    return minCost;
end procedure
```

We also add them in front of the queue to expand them right after their generation, to capture all the cases between two states in which blank is moving around without swapping with one of the pattern tiles. If we treat normally and add these states - which we call equivalent states from now on - at the end of the queue, we would postpone generation of some of the permutations in the PDB, and between the time when the equivalent states are generated and their expansions, some other states may generate some of their children which are unexplored permutations. In such cases, those children would be assigned to an overestimating cost. So it is necessary to add the equivalent states in front of the queue instead of insert them to the end of queue.

Another important issue that equivalent states may cause is that they would be re-generated in different stages of the search, and since there are too many equivalent states in the search space - basically for each state in which the pattern tiles do not partition the grid, there are $25 - 7 = 18$ equivalent states - the search for generating all the cases would take a considerable amount of time. To avoid that, a set of visited states is stored during the search that keep track of the index of permutations that are generated during the search. We use this set to filter the equivalent states that we generated once before. The size of this set would be ${}_{25}P_7 = 2,422,728,000$ (as we also encode the position of blank, we should pick 7 numbers out of 25) which is a very huge number. However, we keep an array of boolean values, in which each index is the index of the permutation in a lexicographic ordering of all such permutations (similar to how we keep the PDB entries). This way, we keep only one bit per entry, and total amount of memory needed for the visited set would be around 300 megabytes.

The last notable point about the PDB creation is how we keep states in the queue. To save memory, we keep a representation of state in form of a 64 bit integer value, which is obtained by concatinating the binary representation of the dual of the blank position along side the pattern tiles. This way we store whatever we need to regenerate the state from the integer representation. The reason for using this representation instead of the factorial based one is the simplicity of extracting the original state from the integer in this representation. We also reduce the amount of memory to keep each state in queue from $25 + 25 + 1 = 51$ bytes to 8 bytes. Although using this representation is still not an efficient use of memory (as it stores each dual value in 5 bits, and also it only uses the first 35 bits of the 64 bit number), we still choose to work with this representation due to simplicity of transferring between the state and the representation, which is also linear in terms of time complexity.

After knowing how we build two pattern databases, we can explain how we query the PDB to get the heuristic values. We use the values stored in the PDBs to find the heuristic value of each state and guide the search. Regarding the symmetry of the puzzle about the main diagonal, we can also use another heuristic that is the reflection of the original one about the main diagonal. This heuristic and the PDBs it contains is shown in figure 4. We sum the value of each of the disjoint PDBs in the original and reflected heuristics and then take the maximum of these two heuristics.

Algorithm 2 shows the IDA* implemented to search. It applies a series of action/undo on a state to generate the children and search each path in the search tree. Here, to find the neighbors of the blank position in the *Iterate* procedure, we use a cache that stores the neighbors of all possible position in the puzzle to avoid repetitive computation. It stores

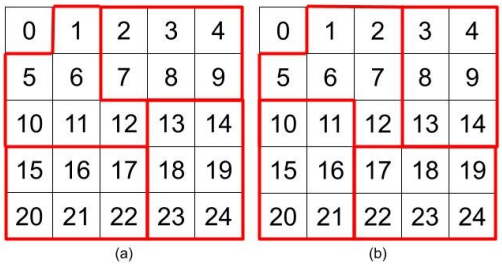

Figure 4: heuristics used in the search, (a) is the showing the patterns of the original heuristic, generated in the PDB creation and (b) is its reflection about the main diagonal

the neighbors of each tile in counter-clockwise order starting from the upper neighbor. Also, before expanding a child node we check if the cost of that child exceeds the limit or not; this would yield to fewer node expansions.

Implementation

In this section we briefly explain the implementation details. The code is written in C++ 17, and Clang++ with LLVM15 is used as the compiler. The code is compiled by -O3 flag.

The code takes the input problems in a form of a text file and for each of them whenever it finds the solution, it prints the solution length, the path to the goal starting from the initial state, and the execution time. During the search, in each iteration it prints the statistics about the current limit, by showing the number of expanded and generated nodes. It also reads the PDBs from files or create it, based on the configuration of the problem that is located in Constants header file.

Experiments and Results

In this section, the results of the experiments are discussed. The hardware specification of the system used for the experiment is an x86-64 architecture with 20 CPUs, 14 cores per socket, and 2 physical threads per core. It is a 12th generation Intel Core i9-12900H processor with a maximum frequency of 5000 MHz, 24 MB L3 cache, and approximately 16 GB memory.

We first check how our implementation works by running some selected problems of the paper, then we compare the results with the results reported in the original paper.

Before going through the experiments, we should mention that the process of PDB creation takes almost 850 seconds, for generating both irregular and rectangular PDBs. A log file containing logs of PDB creation is attached in the appendix.

As solving all the 50 instances would take a considerable amount of time (the average running time of the problems is 2 days according to (Korf and Felner 2002)), here we solved the first 16 problems considering the problems in order of the number of generated nodes in the original paper. A complete log of the problem, the number of node expansions, the solution length and the execution time is attached in the appendix.

Figure 5 shows the execution time in seconds as a function of the problem number. A summary of the statistics about the execution time and the number of node expansions is shown in table 1. As can be observed, There is a huge gap between the maximum and the upper quartile in both execution time and the number of node expansions, and this is also visible from the figure 5 that there is considerable difference between the execution time of the last problem and the others. The last problem is probably the reason behind the considerable difference between the median and the mean value in both execution time and the number of node expansions.

	execution time (s)	total node expansions
mean	402.84	3,494,790,743.12
std	365.15	3,167,232,819.88
min	7.06	61,777,118.00
25%	124.40	1,059,160,984.00
50%	366.53	3,177,915,113.50
75%	547.37	4,783,900,700.00
max	1408.73	12,196,264,959.00

Table 1: The solution length for each approach

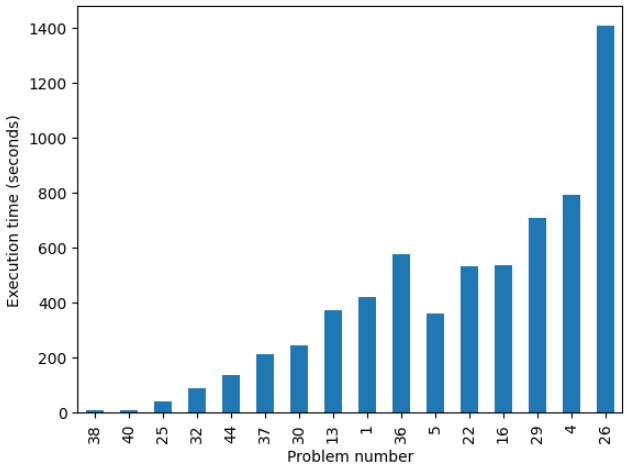

Figure 5: Execution time of selected problems

There is also a linear relation between the number of node expansions and the execution time that is depicted in figure 6. We can see the distribution of each of these two features in the charts parallel to them. According to the figure, by increasing the number of nodes that are needed to be expanded, the execution time also increases almost in a linear manner, which confirms the high correlation between these two features.

On average, the our solution generates 8,682,815 nodes and expands 3,964,075 nodes per second.

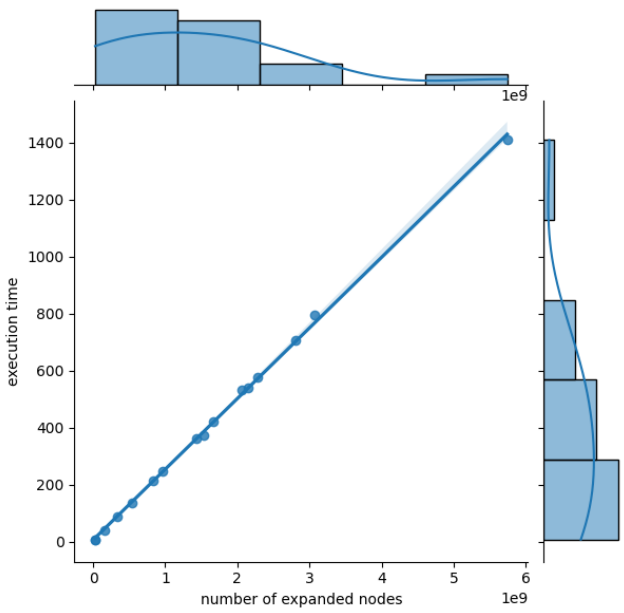

Figure 6: the execution time vs. the number of node expansions in selected problems.

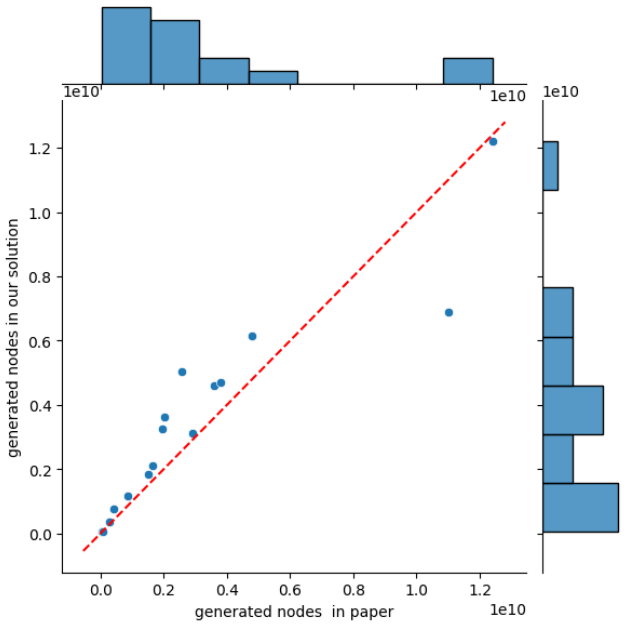

Figure 7: the number of generated nodes in our solution vs. the original paper

The only feature that the original paper reports is the number of generated nodes in each solution, which we use to compare the results. Figure 7 shows the number of generated nodes in our solution vs. the number of generated nodes reported in the table. The dashed red line in the middle of the chart is the identity line, and the histograms parallel to each axis show the distribution of that axis using 8 bins. according to the chart, there is noticeable gap between the number of generated nodes of the last two problems with the others in the paper, while this gap appears between only the last problem and all the others in our solution. Moreover, in most cases there is a small difference between the number of generated nodes, but generally the number of generated nodes reported in the paper is smaller than our solution. There is also a noticeable exception to this trend, which is problem 4, in which the number of generated nodes reported in the paper is 10,991,471,966 while the number of generated nodes in our case is 6,879,803,775. Figure 8 shows a more precise comparison per problem.

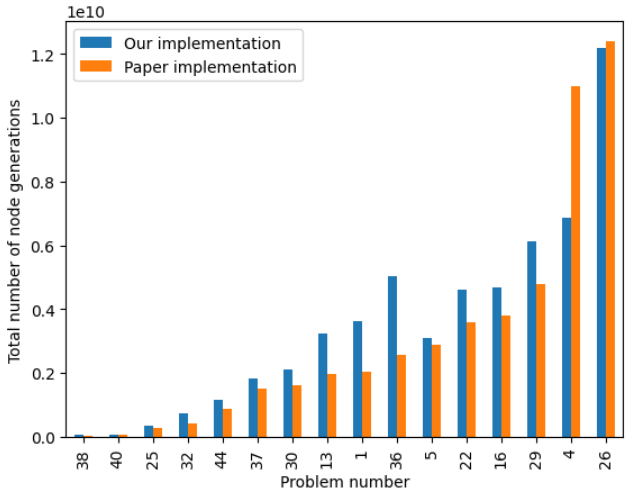

Figure 8: the number of generated nodes in our solution vs. the original paper in each of selected problems

One reason for these differences could be different ordering of the node generation; in other words, the number of generated node in that last layer of search, which is where the solution is found, depends on in what order we are generating the neighbors of each state. Furthermore, considering the exponential growth of the search tree in this problem, the number generated nodes in the last layer of the search is almost the same as the entire number of internal nodes. As a result, this noticeable differences in the number of generated nodes may only be caused by different order of generating the children of each state.

To check if this is a reasonable explanation, we have run an experiment on the first problem to see the difference of the number of generated nodes when the order of generating children is different. In this experiment *changed implementation* refers to an implementation in which the top and the bottom neighbors of blank are generated first and then the left and right neighbors are investigated. Figure 9 shows the difference in node generation of our implementation versus the changed implementation - in this chart the y axis is log scale. As expected, they both generate exactly same number of nodes in all the layers before the last limit, but in the last limit is so significant that it is apparent in the log-scale chart.

The total number of generated nodes in changed implementation is 349,485,807 and the problem is solved in 40.022 seconds, while in the original implementation we only generate 61,777,118 nodes and the problem is solved in 7.056 seconds.

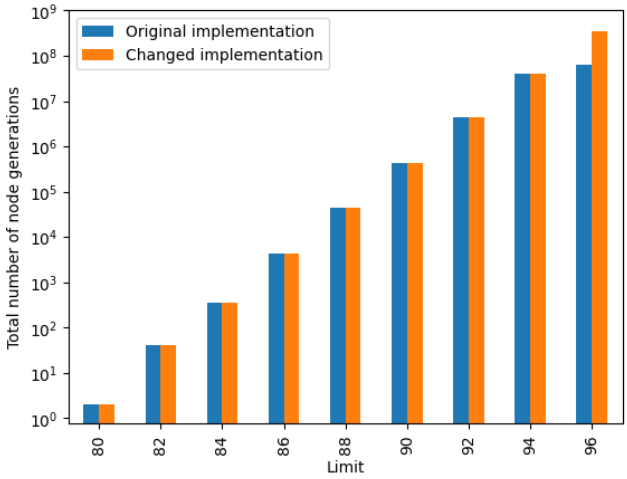

Figure 9: the number of generated nodes in our solution vs. the original paper in each of selected problems

Given this experiment, we could argue that the difference between the number of generated nodes in the original paper and our implementation might be because of the order of node generation.

In summary, we have re-implemented the work of (Korf and Felner 2002) on 24-puzzles by implementing disjoint pattern databases and using them to find optimal solutions of some of the problems mentioned in the paper.

References

Culberson, J. C., and Schaeffer, J. 1998. Pattern databases. *Computational Intelligence* 14(3):318–334.

Korf, R. E., and Felner, A. 2002. Disjoint pattern database heuristics. *Artificial Intelligence* 134(1):9–22.

Korf, R. E., and Schultze, P. 2005. Large-scale parallel breadth-first search. In *Proceedings of the 20th National Conference on Artificial Intelligence - Volume 3*, AAAI'05, 1380–1385. AAAI Press.

Korf, R. E., and Taylor, L. A. 1996. Finding optimal solutions to the twenty-four puzzle. In *AAAI/IAAI, Vol. 2*.

Korf, R. E. 1985. Depth-first iterative-deepening. *Artificial Intelligence* 27(1):97–109.

Appendix

A sample of execution of the our implementation is attached to the uploaded solution. As the content of the results in fairly large, the original files are attached to the uploaded solutions. They are 3 file, final-solution.txt contains the log file of the program, final-solution.csv is the data extracted from the file, such as the solution length, the number of node expansion and generation and the execution time of each problem, and finally a PDB-creation-log.txt that contains the log file of generating the PDBs. The last file does not include any important data except the exact execution time of generating each PDB.