CMPT310 PROJECT REPORT

15-puzzle solver using additive Pattern database

GROUP MEMBERS:

Delong Li : 301297103

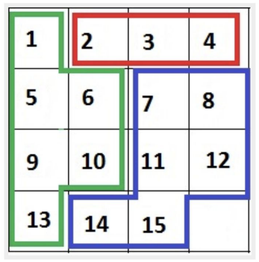
YiFan LIU : 301297560

Introduction:

Based on project requirements, we implemented Iterative deepening A* algorithm and created pattern database as admissible heuristics to solve random generated solvable 15-Puzzles. We randomly generated 100 15-Puzzle instances, each instance we applied the program to solve it 10 times, each time we collected the running time and computed the average running time for each instance.

Design and Implement Pattern database:

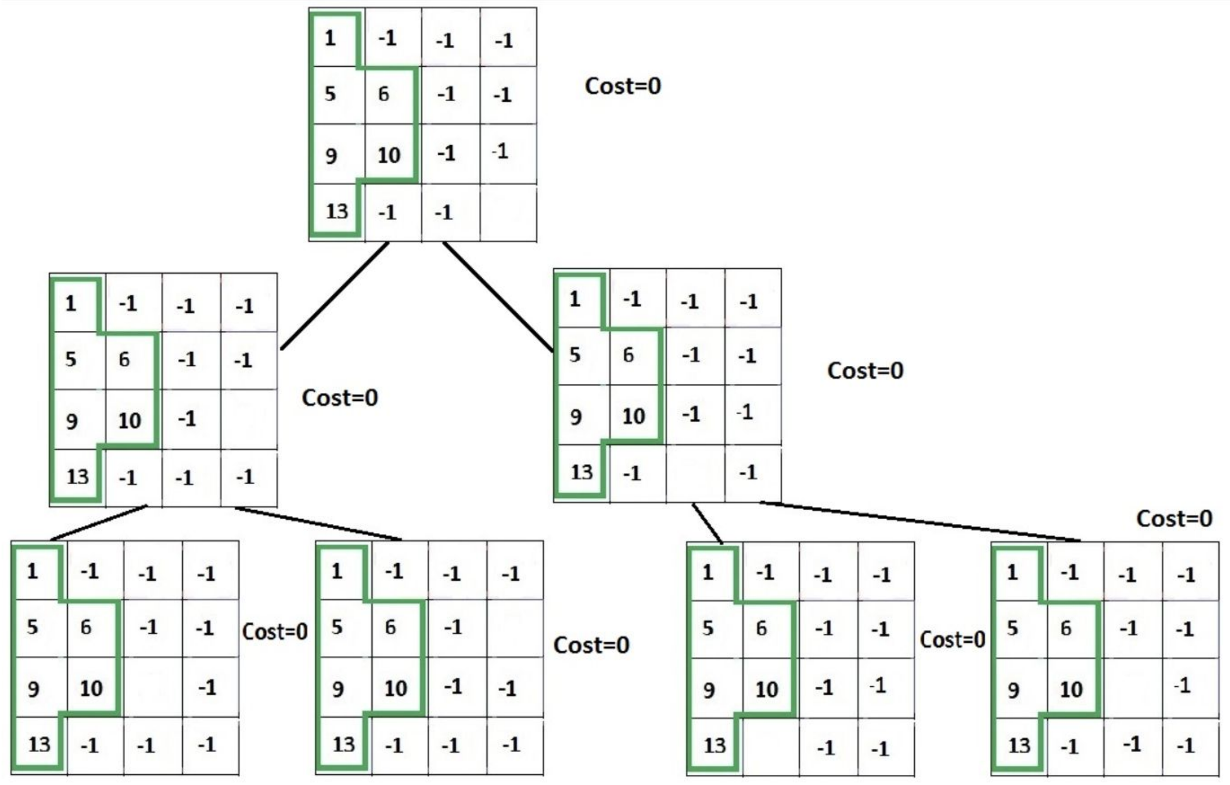
Based on the research result, We decided to build **Dynamically partitioned Additive Pattern Database** as admissible heuristics to solve 15-Puzzle. Pattern database is a database of all the possible permutations of a particular group (partitions) of tiles and the minimum moves required from that pattern to reach the goal state for that partition.



The pattern database is created by using breadth first search (BFS) to the final (goal) state of the select partition till all patterns are generated. We used 15-Puzzle with a 6-6-3 partitioning, the exact partitioning is shown on the left.
As we can see the 15-puzzle is partitioned into two blocks consisting of 6 tiles each and 1 block consisting of 3 tiles, so we created 3 txt files consisting the cost values of the 3 partitions respectively. We used three vectors to store three partitions of the goal state. Each vector contains specific location(ij) of all tiles, including blank tile. Then we applied map data structure, using vector(partition goal state) as key and a pair as value. Each pair contains the cost to reach the goal state of the target partition and whether the current instance visited.

Implementation details

As we know that the cost is calculated without considering the interference of other groups. We implemented three functions to generate pattern database. For the first function **bfs_pattern_database(map<vecint, pair<int, bool>>& database, vecint goal),** it passed two parameters, the first one is a variable called database, using map data structure and the second one is an integer vector which represent one of the partitions of the goal state. Initially, we passed the goal(integer vector) into map as a key and set cost = 0 and the boolean variable = False. Then, we constructed an empty integer vector queue and push the goal into queue. Each time Popped the vector in the front of the queue, check all possible moves of that vector using function move(**map<vecint, pair<int, bool>>& database, queue<vecient>& myqueue, int position0, int new_position0, vecint parent**) , to know if any of them can be pushed into queue. We use a while loop to repeat the procedure, until the queue is empty. If queue is empty, which means that the variable map contains all possible states for the passed partition goal state. Finally, we wrote the variable database into a txt file, each row contains a possible position for the partition goal state and the cost.



For the second function, move(**map<vecint, pair<int, bool>>& database, queue<vecient>& myqueue, int position0, int new_position0, vecint parent**), it will check whether the current state satisfied following two conditions, if so, add or update the corresponding key and value into map database. The first condition is If the current pattern does not exist in the map and the program will first check whether any tile in the goal partition changed with blank tile, if so, use the summation of the parent state cost + 1 as the cost of the current state and set the boolean variable equal to true, which represent the state is already visited. The second condition is if the current pattern has already exist in the map and the goal partition changed with blank tile, comparing the parent state cost + 1 with the cost of the appeared state. if the new cost is less than the other cost, replace the previous cost with the new cost. Otherwise, keep the previous cost and continue the first function.

Finally, the program will generate 3 txt file, each text file contains all permutations for each partition. When running Iterative deepening A* algorithm, each 15-puzzle will be partitioned into 3 parts and each partition will return a heuristic value. The summation of these three values will be treated as admissible heuristics(h-value) to speed up the IDA* to solve the 15-Puzzle instances.

Iterative deepening A*

Iterative deepening A* (IDA*) is a graph traversal and path search algorithm that can find the shortest path between a designated start node and any member of a set of goal nodes in a weighted graph. It is a variant of iterative deepening depth-first search that borrows the idea to use a heuristic function to evaluate the remaining cost to get to the goal from the A* search algorithm. Since it is a depth-first search algorithm, its memory usage is lower than in A*, but unlike ordinary iterative deepening search, it concentrates on exploring the most promising nodes and thus doesn't go to the same depth everywhere in the search tree.

IDA* is a memory constrained version of A*. It does everything that the A* does, it has the optimal characteristics of A* to find the shortest path but it uses less memory than A*.

Data Collection and Running time analyze

Solves normal instances of the 15 puzzle IDA* with the Manhattan distance heuristic and Dynamically partitioned Additive Pattern Database. We wrote a text file as an input file, containing 10 random generated 15-Puzzle instances. We wrote two cpp files, one called pattern_database_generator.cpp, we used the file to generate pattern database with 6-6-3 partition. The other cpp file called 15_Puzzle solver.cpp. Inside the program, we implemented IDA* with Manhattan distance heuristics and IDA* with Dynamically partitioned Additive Pattern Database heuristic.

- Experiment 1 : Read Input text file with multiple randomly generated 15-Puzzle instances and use IDA* with Manhattan distance heuristics to solve them and collecting the running time and length of the solution
- Experiment 2 : Read Input text file with multiple randomly generated 15-Puzzle instances and use IDA* with pattern database heuristics to solve them and collecting the running time and length of the solution

Running time of generating loading pattern database:

1. Generate pattern database with 3 partitions took 7721.8 seconds
    a. 6 tiles : 3844s and 3867s
    b. 3 tiles : 0.76s
2. Loading pattern database to cpp file took within 10s.

Based on the data, using IDE* with Dynamically partitioned Additive Pattern Database heuristic took less time than using IDA* with Manhattan distance heuristic. However, the length of the solution is the same.