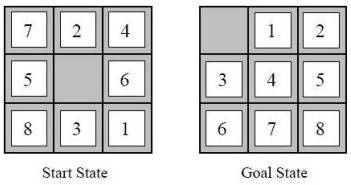
Parallel N Puzzle Solver

Zhonglin Yang, zy2496 Yuxuan Luo, yl4524

1 Introduction

For our group project, we would like to implement a parallel N puzzle solver in Haskell. The N puzzle problem is a classical search problem. The start state of the search problem is that there are N tiles numbered from 1 to N inside a side length $\sqrt{N+1}$ square frame and one position inside the square frame is left unoccupied. Each step, one tile that are adjacent to the unoccupied position can slides to the unoccupied position. The final goal is to have N tiles placed in a desired order. The following picture illustrates one possible pair of start state and goal state for an 8 puzzle problem.



2 Search Algorithms

To solve a N puzzle problem, search algorithms can be used. We can define a N puzzle problem as a classical search problem with a size $(N+1)!$ state space. It is hard to find (NP-hard) a optimal solution for a N puzzle problem, however, finding a feasible solution is not difficult. Uniform-cost search algorithm and A* algorithm will be used to solve N puzzle problems in our project.

2.1 Uniform-cost search

Uniform-cost search algorithm is a variant of Dijikstra's algorithm, which is usually used for large graph search problems. A priority queue is also used to store nodes. In the priority queue, the minimum cumulative cost node get the maximum priority. The algorithm is the following.

```
queue <— [root node]
while queue is not empty:
        choose and remove a node x from queue
        if state[x] is a goal state then
                return the corresponding solution
        else
                expand x, insert new nodes into queue
return failure
```

2.2 A* search

A* search is an informed search algorithm. It is almost identical to uniform-cost search but takes the state of a node and the search state space into consideration. It use heuristic to reduce the amount of search. A heuristic function $h(x)$ is used to estimate the smallest possible cost from state x to the goal state. $g(x)$ is the minimum cost from the start state to state x. A* search chooses a node from the queue with lowest $g(x) + h(x)$ while Uniform-cost search chooses a node from the queue with lowest $g(x)$.

3 Objective

Our base implementation can be found on [1]. For each iteration, the program generates all possible moves and adds them to a priority queue where they are placed based on a combination of the heuristic and the search algorithm provided by the user. The program then applies the best move from the priority queue, makes a recursive call, and then applies the second-best move from the priority queue, makes a recursive call... until the solved state of the board is reached, or until all moves have been exhausted.

This naive implementation does a linear search through all the possible moves with recursion. While it is technically correct and efficiently optimized, it still leaves most of the computing resources on the table if the CPU has more than one core. We will explore different parallel evaluation strategies discussed in class and experiment with parameters such as thread and parallelization depth to gain a more granular control over the performance. According to [2], the 15-Puzzle can reach upwards a depth of 80 and billions of intermediate states. As a result, we believe there are lots of potential to improve the efficiency of the solver if we are able to utilize the CPU fully. In the meantime, it is very import for us to find the sweet of of when to stop parallel evaluation so the system is not overwhelmed with wasted Sparks.

References

[1] Keuhdall. Keuhdall/n-puzzle: Implementation of algorithms to solve n-puzzles in haskell. https://github.com/keuhdall/N-Puzzle/.

[2] 15-puzzle optimal solver. http://kociemba.org/themen/fifteen/fifteensolver.html.