

Documentation

1. Problem Statement, Solution

15 Puzzle Problem.

Proposed solution: use **IDA*** (Iterative deepening A*) to find the solution which requires the minimum number of steps to reach the complete puzzle. (I.e., shortest one)
While the standard iterative deepening depth-first search uses search depth as the cutoff for each iteration, the IDA* uses the more informative $f(n) = g(n) + h(n)$, where $g(n)$ is the cost to travel from the root to node n and $h(n)$ is a problem-specific heuristic estimate of the cost to travel from n to the goal.

In our case, the chosen $h(n)$ is the **Manhattan Distance** of the current board state with respect to the solution one, $g(n)$ is the **number of steps** made so far, meaning, the number of times the empty tile has been moved.

2. Details of implementation

The search has been parallelized using a Java ExecutorService with a thread pool of fixed size, and the idea is that Java Futures are being submitted recursively to the service, and at each submission the current number of steps is increased by one (I.e. we “advance” with one move).

The workload is assigned evenly to each thread, in the sense that each submitted task receives $\text{nrThreads}/\text{nrOfUnexploredNextMoves}$ threads, and when nrThreads reaches 0, the search becomes sequential, similarly to what happens in parallelizing algorithms such as Merge Sort/ Karatsuba’s Multiplication.

When the futures finish, the smallest length of the path is returned and, if we’ve reached a solution, the algorithm finishes. If not, a new minimum bound is set the parallel search begins again.

The distributed algorithm uses the MPJ Express Java library which implements the MPI Interface. The number of nodes is passed as a command line argument, with one master node and several worker nodes.

Work is assigned to each worker by performing one or more `generateMoves()` from the root node recursively, until the total workload gets as close as possible to the number of nodes.

When each worker receives its state, it begins searching from that point on using the iterative A* sequential implementation, and sends each solution along with its number of steps to the master function. The master function checks whether a solution has been found, and if it has, all the workers are signaled to stop searching. If not, the min bound is reset and the search continues.

Our matrix class represents one puzzle state, i.e. board.

3. Times

Size	Number of threads	Threads time	MPI time
40 moves	2	248ms	285ms
40 moves	4	234ms	324ms
51 moves	2	5463ms	6982ms
51 moves	4	3683ms	8484ms
56 moves	2	12965ms	19330ms
56 moves	4	12131ms	14034ms
61 moves	2	61834ms	88141ms
61 moves	4	52951ms	72446ms

4. Conclusions

The problem can be solved relatively easy using the IDA* algorithm, and parallelizing the solution helps with the execution time when larger search spaces occur.

Distributing the search across several nodes yields approximately the same time as with one node, but a little bit more efficient when the sizes are bigger, although an overhead which represents the time spent creating the nodes and sending/receiving data needs to be considered, as with any distribution environments.

5. Bibliography/useful resources

<http://kociemba.org/themen/fifteen/fifteensolver.html>

<https://computerpuzzle.net/english/15puzzle/index.html>