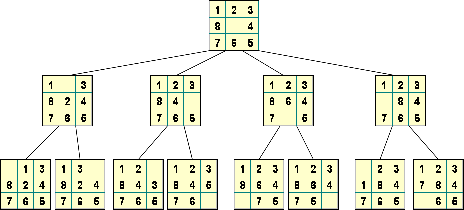
**PDP Semester Project – 15-Puzzle Problem**

****

**Iterative deepening A\*** (**IDA\***) is a graph traversal and [path search](https://en.wikipedia.org/wiki/Pathfinding) algorithm that can find the [shortest path](https://en.wikipedia.org/wiki/Shortest_path_problem) 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](https://en.wikipedia.org/wiki/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](https://en.wikipedia.org/wiki/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 does not go to the same depth everywhere in the search tree. Unlike A\*, IDA\* does not utilize [dynamic programming](https://en.wikipedia.org/wiki/Dynamic_programming) and therefore often ends up exploring the same nodes many times.

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){\displaystyle f(n)=g(n)+h(n)}, where g(n){\displaystyle g(n)} is the cost to travel from the root to node n {\displaystyle n}and h(n) {\displaystyle h(n)}is a problem-specific heuristic estimate of the cost to travel from {\displaystyle n}to the goal.

**Iterative-deepening-A\*** works as follows: at each iteration, perform a depth-first search, cutting off a branch when its total cost f(n)=g(n)+h(n){\displaystyle f(n)=g(n)+h(n)} exceeds a given *threshold*. This threshold starts at the estimate of the cost at the initial state, and increases for each iteration of the algorithm. At each iteration, the threshold used for the next iteration is the minimum cost of all values that exceeded the current threshold.

In this project, the heuristic h(n) will be computed as Manhattan distance (also called the Taxicab distance or City Block distance, calculates the distance between two real-valued vectors). In our case it will sum how far is each value from its correct position. There has also been some research done where there is an optimization called [The Manhattan Pair Distance](MPD.pdf) which is not used in our case. g(n) is the number of moves made so far.

The program uses an ExecutorService with a fixed thread pool, where Java Futures are submitted recursively to the service, and at each submission the current number of steps is incremented. The workload is assigned evenly to each thread, in the sense that each submitted task receives nrThreads/nrOfUnexploredNextMoves threads, and when nrThreads reaches 0, the search becomes sequential. 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 MPI. Work is assigned to each worker by performing generateMoves() from the root recursively until the total workload reaches (close to) the number of nodes. When each worker receives its state, it begins searching 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 signals all workers to stop searching in case is has. If not, the min bound is reset and the search continues.

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **Time** | **Steps** | **Work./Thread.** |
| P | 377ms | 40 | 1 |
| P | 248ms | 40 | 5 |
| P | 68373ms | 61 | 12 |
| P | 59364ms | 61 | 20 |
| D | 330ms | 40 | 1 |
| D | 364ms | 40 | 5 |
| D | 91943ms | 61 | 12 |
| D | 93321ms | 61 | 20 |