**PDP project – 15 Puzzle problem**

1. **Team members**
   1. Lupu Alexandra (934/1)
   2. Mardari Dayana-Raluca (934/1)
2. **Problem statement & solution**

Problem statement: <https://www.cs.ubbcluj.ro/~rlupsa/edu/pdp/projects.html>

Proposed solution: Employ IDA\* (Iterative Deeping A\*) to determine the solution that involves the fewest steps to complete the puzzle (i.e. the shortest path). Unlike the standard iterative deepening depth-first-search, which utilizes the search depth as the cutoff for each iteration, IDA\* utilizes the more informative heuristic function **f(n) = g(n) + h(n)**. Here, g(n) represents the cost of traveling 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 context, the selected h(n) is the **Manhattan Distance** of the current board state concerning the solution, while g(n) denotes the number of steps taken thus far. This implies the count of times the empty tile has been moved.

1. **Implementation details**

**Threads approach:**

The search process has been parallelized through the utilization of a Java **ExecutorService** with a fixed-size thread pool. The approach involves recursively submitting **Java Futures** to the service, with each submission leading to an incremental increase in the current number of steps (i.e. progressing by one move). The workload is evenly distributed among threads, ensuring that each submitted task receives a share proportional to **nrThreads/nrOfUnexploredNextMoves**. As **nrThreads** decreases to zero, the search transitions to a sequential mode, akin to the behavior observed in parallel algorithms like Merge Sort or Karatsuba's Multiplication.

Upon completion of the futures, the algorithm returns the smallest path length, and if a solution is reached, the algorithm concludes. Otherwise, a new minimum bound is established, and the parallel search recommences.

**Distributed approach:**

The distributed algorithm employs the **MPJ Express Java library**, implementing the MPI Interface. The number of nodes is specified as a command line argument, designating one master node and several worker nodes. Work distribution to each worker involves executing one or more **generateMoves()** operations from the root node recursively, striving to achieve a workload close to the number of nodes.

Once a worker receives its state, it initiates the search from that point using the iterative A\* sequential implementation. Subsequently, each solution, along with its corresponding number of steps, is transmitted to the master function. The master function assesses whether a solution has been found, signaling all workers to cease searching if so. If no solution is identified, the minimum bound is reset, and the search process persists.

1. **Software & hardware**
   1. Software: Java, IntelliJ IDEA
   2. Hardware: Macbook Pro, chip: Apple M2 Pro, memory: 16 GB
2. **Testing**

|  |  |  |  |
| --- | --- | --- | --- |
| Approach | Time Elapsed | Difficulty | Threads/Nodes |
| Parallelized | 113ms | 40 | 1 |
| Parallelized | 114ms | 40 | 5 |
| Parallelized | 2105ms | 51 | 5 |
| Parallelized | 2143ms | 51 | 10 |
| Parallelized | 7566ms | 56 | 10 |
| Parallelized | 29299ms | 61 | 12 |
| Distributed | 149ms | 40 | 2 |
| Distributed | 220ms | 40 | 5 |
| Distributed | 2012ms | 51 | 5 |
| Distributed | 1547ms | 51 | 11 |
| Distributed | 3713ms | 56 | 11 |
| Distributed | 25514ms | 61 | 12 |