Authors:
Robin Karlsson
Alexander Korpas

Date:
2021-10-19

A-Mazed report

For our solution we've utilized thread-safe lists, sets and maps, as well as an AtomicBoolean object.

- To keep track of our list of threads, we used CopyOnWriteArrayList so that as soon as it's mutated, a fresh copy is created.
- In order to be able to stop all thread's work when the goal is found, an AtomicBoolean is used for setting the flag "goalFound". Before a thread begins its work, it checks if this flag is set and if so, returns null.
- ConcurrentSkipListMap and ConcurrentSkipListSet are used to safely execute insertion and removal operations concurrently by multiple threads.

When a thread is spawned, we pass in the maze, the current node, the forkAfter value, a set of previously visited nodes as well as a map of predecessor node IDs.

The algorithm

To implement the DFS concurrently, we follow a few simple steps:

- If the goal has been found, return immediately
- Add the current note ID to our visited set
- If the current node ID is the goal, set the flag and return a path from the start node to the current node
- Get all neighbors of current node
- For each neighbor node, add it to an unvisited list unless we've already visited it
- Map the neighbor node to the current node in order to keep track of predecessors
- If we have one unvisited neighbor node, move the player to the next node in the unvisited list and run the algorithm again, starting from that node
- If we have more than one, for each unvisited node:
  - To the thread safe ArrayList ("threads"):
    - Fork a child thread to run ForkJoinSolver, starting from each unvisited node, respectively, passing along the current data (visited set, predecessor map etc.)
- Go through the list of threads doing work and join their respective partial results
- As long as the result isn't null, set the path to the partial result
- Finally, return the path (null if not found)