Victoria University of Wellington

School of Engineering and Computer Science

NWEN303 Concurrent Programming (2016)
Project One: Concurrent Search in a Maze

This project involves implementing a program that uses multiple threads to search for a path in an undirected graph. The problem is presented as a search for a path in a maze, but the structure is similar to many other problems where a task may be broken down into subtasks and/or alternatives that can be solved concurrently.

The problem

A group of friends are trying to find their way through a maze. The maze is large and complex (there can be more than one path between any two positions in the maze), and they have limited time (they are all going to the pub afterwards and don't want to lose too much drinking time), so they devise a plan to allow them to work together to find their way to the other end of the maze. Their basic idea is to split up when they come to a junction in the maze, so long as there are still enough of them, and use a system of markings on the wall at the beginning of each path to record whether that path has been explored and whether it has been found to lead to the exit or a dead-end.

If a wall has no marking, that path has not yet been explored. When a group first takes that path, they mark it as being "live", and record which path they arrived at that junction on (we will assume that the maze is square and that all paths run North-South or East-West, so each path at a junction can be identified as coming from/leading to the North, South, East or West).

If a group reaches a dead-end (a position with no paths leading out of it other than the one they arrived on), they return to the last junction on the path they arrived on, mark that path as "dead", and look for another path. A junction where all other paths have been marked as "dead" is also regarded as a dead-end.

If a group reaches the exit and no others are already at the exit, one member of the group follows the path they took from the start to the exit (ignoring side tracks that proved to be dead-ends) back to the start, marking the walls as "gold" to indicate that they are part of a path to the exit, and then returns to the exit. The rest of the group waits at the exit for the others to arrive. If a group reaches the exit and there are others already at the exit, they join the group that is there, and wait for the rest to arrive. When all of the initial group have reached the exit, they have completed the maze and they can all go to the pub.

If a group reaches a junction where one of the paths is marked as "gold", they all follow that path.

If a group reaches a junction where two or more paths are unlabelled, they split into smaller groups of roughly equal size (i.e. group sizes differ by at most one), and each group takes one of the unlabelled paths. If the number in the group is less than the number of unlabelled paths, this will leave some paths still unexplored and thus unlabelled.

If a group reaches a junction where no paths are unlabelled, but some are labelled as "live", they again split into smaller groups of roughly equal size, and each group takes one of the "live" paths.

If a group meets another group which is returning from a dead-end, the groups merge into one and continue returning from the dead-end.

If two groups meet at a junction where there are either unlabelled or "live" paths (other than they ones they arrived on), they again merge into one and then decide how to split up and what paths to take as though they had arrived there together.

Towards a solution

You are to simulate this search process in Java, using threads to represent groups of friends, using suitable synchronisation between threads to ensure that the process is implemented correctly and that all of the friends reach the exit as quickly as possible. Your program should start by reading a description of the maze from a file with the format described below. The name of the file, along with the number of friends, should be given as command line arguments.

Since we are assuming that the maze is square, it can be represented as a two dimensional grid, and there can never be more than four paths leading to/from a junction. We will also assume that there is exactly on entrance and one exit, which are non-wall points on the edge of the maze.

The first line of the file should contain three integers, M, U and V, where M is the number of rows and columns in the grid, and U and V are the coordinates (row and column) of the tunnel entrance. These coordinates should be 0-based indexes, counting from the top-left hand corner. The entrance must be at the edge of the maze but cannot be in a corner, so we must have either $U = 0$ and $0 < V < M - 1$, or $V = 0$ and $0 < U < M - 1$; this also means that M must be greater than 2.

The rest of the file contains M lines of M characters each, where `X` represents a wall and space represents a path. Paths should never be more than one space wide (i.e. there should never be a square of four spaces). See `grid1.txt` for a sample file.

Your program will need to store a representation of the maze, e.g. as a two dimensional array, in which each cell represents either a wall or a space (part of a path). At each junction, you will need to store the marks needed for each path out of the junction, as described above.

The main point of this exercise is to synchronise the behaviour of the threads so as to ensure that all of the friends reach the exit while allowing the maximum amount of concurrency (thus minimising the time required). You should think carefully about how to control access to shared data structures and how to coordinate the actions of the threads. Where appropriate, you should use techniques discussed in lectures (can you do it without locks?); you may also use Java concurrency classes that we have not discussed.

You should check that the procedure outlined above does in fact guarantee that all friends will eventually reach the exit, and if not amend it so that it does. Once you have the program working correctly, you should then consider whether the procedure can be modified in a way that will allow the friends to all reach the exit more quickly, perhaps by changing the rules for deciding how/when to split the group and which path(s) to explore, or by changing the information recorded on the walls. Think about what should happen if there are multiple paths to the exit.

You should run your program on a least two machines (including Lighthouse) with different numbers or cores, varying the size and layout of the maze and the number of friends, to see how this affects the performance of the program. You should also think about how to produce output illustrating the behaviour of the program, and try adding delays to simulate the time taken to move along a path from one junction to the next.

Reporting your results

You should submit your Java source code, along with a report explaining what you have done. Your report should explain the design of your program, including the classes used and their purpose. In particular, you should explain carefully the way in which threads are created, how they access shared data and communicate with each other, and any modifications you made to the search process decsribe above. You should present sample output illustrating the behaviour of your program, explaining how varying the factors mentioned above affects its perforance, along with the associated data files and instructions on how to run the program.

Note that a significant proportion (quite likely around 50%) of the marks will be based on your report, so it is important that you allow enough time to write a report that adequately presents what you have done.