Concurrent Programming
Practical 3: Concurrent Depth-First Search

In this practical we will develop a concurrent program for depth-first search. The course web site includes a program that makes use of this depth-first search in order to solve sudoku problems.

I recommended that you study the material in the lecture notes related to breadth-first search before starting this practical.

Deadline: practical sessions in Week 7.

As in lectures, we will deal with graphs with the following interface.

```
/** A trait representing an unlabelled graph with nodes of type N. */
trait Graph[N]{
  /** The successors of node n. */
  def succs(n: N): List[N]
}
```

We will implement a depth-first search matching the following interface.

```
/** Abstract class representing graph search problems. */
abstract class GraphSearch[N](g: Graph[N]){
  /** Perform a depth−first search in g, starting from start, for a node that
    * satisfies isTarget. */
  def apply(start: N, isTarget: N => Boolean): Option[N]
}
```

We will implement a tree search (as opposed to a graph search): we will not keep track of nodes of the graph previously seen. In particular, the sudoku search graph is, in fact, a tree, so no node will be encountered twice via different routes. Figure 1 contains code for a sequential depth-first tree search.

Your task is to implement a concurrent version of this search. In particular, you will need to replace the stack with a suitable concurrent stack.

You will need to think carefully about termination. You may, if you want, assume that the graph contains at least one reachable node that satisfies isTarget. **Optional:** do not make this assumption, so your code will need to deal with the case where no solution is found.

You should run your search on some sudoku problems.

The course website contains a zip file containing the following.

- `GraphSearch.scala`, which contains all the above code.

```
/** Sequential depth−first search of graph g. */
class SeqGraphSearch[N](g: Graph[N]) extends GraphSearch[N](g){
  /** Perform a depth−first search in g, starting from start, for a node that
    * satisfies isTarget. This performs a tree search, not storing the set of
    * nodes seen previously. */
  def apply(start: N, isTarget: N => Boolean): Option[N] = {
    // Stack storing nodes
    val stack = new Stack[N](); stack.push(start)

    while(stack.nonEmpty){
      val n = stack.pop
      for(n1 <− g.succs(n)){
        if(isTarget(n1)) return Some(n1) else stack.push(n1)
      }
    }
    None
  }
}
```

Figure 1: The sequential depth-first tree search.

- `Partial.scala` and `Sudoku.scala` which represents sudoku problems as graphs, and solves them; you do not need to study this code in much detail.
- `ConcGraphSearch.scala` which provides the outline of the search you have to implement.
- Various files with suffix `.sud`, which represent sudoku problems; in particular, `impossible.sud` represents a problem with no solution.

Your report should be in the form of well commented code, including a brief description of any design decisions.