CS4811 Chapter 3 Handout and In-class Exercise
Structures and Strategies for State Space Search

Initial algorithm from: Stuart Russell and Peter Norvig
Artificial Intelligence a Modern Approach, Prentice Hall Series in Artificial Intelligence, 2003.

function TREE-SEARCH (*problem*)
returns a solution, or failure
inputs:
    problem, a search problem
variables:
    fringe, states that will be explored

// Initially, the search graph contains only the start node.
fringe ← INSERT(MAKE-NODE(INITIAL-STATE [*problem*]), *fringe*)
loop do
    // Stop if there are no nodes to be explored.
    if EMPTY?(*fringe*) **then return** failure
    // Pick the node to be explored.
    node ← REMOVE-FIRST(*fringe*)
    // Check if the goal has been reached.
    // SOLUTION traverses the graph to return the solution.
    if GOAL-TEST[*problem*] applied to STATE[*node*] succeeds
        then return SOLUTION(*node*)
    // Find the children of the node explored.
    // (See the next algorithm for EXPAND.)
    fringe ← INSERT-ALL(EXPAND(*node, problem*), *fringe*)

function EXPAND (*node, problem*)
returns a set of nodes

successors ← the empty set
for each < *action, result*> **in** SUCCESSOR-FN [*problem*(STATE[*node*]) **do**
    s ← a new NODE
    STATE[*s*] ← *result*
    PARENT-NODE[*s*] ← *node*
    ACTION[*s*] ← *action*
    PATH-COST[*s*] ← PATH-COST[*node*] + STEP-COST(*node,action,s*)
    DEPTH[*s*] ← DEPTH[*node*] +1
    add *s* to *successors*
return *successors*

Now, we will change the tree search method into a graph search by renaming *fringe* to *open*, and adding a new list called *closed*.

function GRAPH-SEARCH (*problem*)
returns a solution, or failure

inputs:
    problem, a search problem
variables:
    open, states that will be explored
    closed, states that have already been explored

// Initially, the search graph contains only the start node.
open ← INSERT(MAKE-NODE(INITIAL-STATE [*problem*]),*open*)
loop do
    // Stop if there are no nodes to be explored.
    if EMPTY?(*open*) **then return** failure
    // Pick the node to be explored.
    node ← REMOVE-FIRST(*open*)
    // Put the picked node in the closed list.
    closed ← INSERT-ALL(node, *closed*)
    // Check if the goal has been reached.
    // SOLUTION traverses the graph to return the solution.
    if GOAL-TEST[*problem*] applied to STATE[*node*] succeeds
        then return SOLUTION(*node*)
    // Find the children of the node explored.
    // Discard children that are already in *open* or *closed*.
    // (INSERT-NOTSEEN should take care of this.)
    // The EXPAND algorithm is the same.
    open ← INSERT-NOTSEEN(EXPAND(*node, problem*), *open*)

Note that the GRAPH-SEARCH method outlined above is a generic search algorithm. The last line where the *open* list is updated controls the type of the search. For instance, if the new nodes are inserted to the beginning, then it is Depth First Search (DFS). If the new nodes are inserted to to the end, then it is Breadth First Search (BFS). If the open list is ordered with respect to a heuristic function, then it is heuristic search.

In class exercise

A farmer wants to get his goat, wolf and cabbage to the other side of the river. His boat isn't very big and can only carry him and either his goat, his wolf or his cabbage. Now . . . if he leaves the goat alone with the cabbage, the goat will gobble up the cabbage. If he leaves the wolf alone with the goat, the wolf will gobble up the goat. When the farmer is present, the goat and cabbage are safe from being gobbled up by their predators.

How does the farmer manage to get everything safely to the other side of the river?

Formulate this as a search problem. In other words,

- Define what a state is.
- Define the initial state and the goal state
- Describe how to obtain the children of a state, i.e., define the SUCCESSOR-FN method.
- Draw the full search tree with repeated nodes omitted.