Generic search algorithms

Tree search can be used is the state space is a tree, otherwise **graph search** must be used. All search algorithms (BFS, DFS, uniform-cost, A*, etc) are variations of one of these (usually graph search). The only difference between tree search and graph search is that tree search does not need to store the explored set, because we are guaranteed never to attempt to visit the same state twice.

```
TREE-SEARCH(problem):
    Initialize the frontier using the initial state of problem
    Loop forever:
        If frontier is empty, return failure
        Choose a leaf node from the frontier and remove it (from the frontier)
        If the node contains a goal state, return the corresponding solution
        If the node did not contain a goal state, expand the chosen node, adding the resulting nodes to the frontier
```

```
GRAPH-SEARCH(problem):
    Initialize the frontier using the initial state of the problem
    Initialize the explored set to empty
    Loop forever:
        If frontier is empty, return failure
        Choose a leaf node from the frontier and remove it (from the frontier)
        If the node contains a goal state, return the corresponding solution
        Add the node to the explored set
        If the node did not contain a goal state, expand the chosen node, adding the resulting nodes to the frontier,
            but only if the resulting node is not already in the frontier or the explored set
```

Uninformed search algorithms

Breadth-first search: Run the generic graph search algorithm with the frontier stored as a (LIFO) queue.
Depth-first search: Run the generic graph search algorithm with the frontier stored as a (FIFO) stack.

Uniform cost search
We have a node data structure that contains a state, a path-cost (also known as g), a pointer to the parent node, and the action that generated this state from the parent state.

```
UNIFORM-COST-SEARCH(problem)  // aka Dijkstra's algorithm
    node ← a new node corresponding to the initial state, with path-cost (g) = 0
    frontier ← a priority queue of nodes sorted by path-cost (g), initialized to contain only node
    explored ← an empty set of states
    loop forever:
        if frontier is empty, return failure
        node ← pop(frontier)  // remove lowest path-cost node from frontier (smallest g)
        if IS-GOAL(node.state), then return the corresponding solution
        add node.state to the explored set
        for each action in ACTIONS(node.state):
            child_node ← new node with   child_node.state = RESULT(node.state, action)
                                         child_node.g = node.g + COST(node.state, action, child_node.state)
                                         child_node.action = action
                                         child_node.parent = node
            if child_node.state is not in explored or frontier:
                add child_node to frontier
            else if child_node.state is already in frontier, but child_node.g is smaller than the frontier's:
                replace that frontier node with child_node
```