COMP250 WEEK 3 WORKSHOP ACTIVITY

Version 1.0
Computing
COMP250

Ed Powley

Introduction

In this workshop, you will implement a solver for the classic puzzle game Sokoban.

Begin by forking and cloning the base repository:

https://gamesgit.falmouth.ac.uk/projects/COMP250/repos/comp250-workshop-sokoban/browse

Open the sample project in Unity. The bulk of your work will take place in the Solver class, which is called from the main game to find a solution to the puzzle.

Basic breadth first search

The following pseudocode illustrates a basic breadth first search (BFS) based planner. The planner uses a node class containing three pieces of information: the parent node, the action that led to this node, and the game state at this node. The BFS itself is based around a **queue** of nodes, i.e. a last-in, first-out data structure. Note that C# provides a `Queue` class in `System.Collections.Generic`. Also note the importance of taking a copy of the state before descending the tree; otherwise you run the risk of changing states that are stored in existing nodes and hence breaking the algorithm. The provided `GameState` class contains a `Copy()` method as well as a copy constructor.

```
procedure SOLVE(state)
    root ← new node (null, null, state)
    q ← new queue
    add root to q
    while q is not empty do
        node ← dequeue from q
        for each available action from node.state do
            nextState ← copy of node.state
            apply action to nextState
            child ← new node (node, action, nextState)
            if nextState is solved then
                return RECONSTRUCTPLAN(child)
            else
                add child to q
            end if
        end for
    end while
end procedure
procedure RECONSTRUCTPLAN(node)
    plan ← empty list
    while node is not null do
        insert node.action at the beginning of plan
        node ← node.parent
    end while
```

```
    return plan
end procedure
```

Notice that RECONSTRUCTPLAN iterates from the child node up to the root, hence it reconstructs the path in reverse. This is why it inserts at the beginning, rather than the end, of the list.

Improving the algorithm

If you test your algorithm now, you will find it easily solves the first level as well as some other simple levels, but fails on moderately complex levels. One of the problems is that there is nothing to stop the algorithm revisiting a state it already visited, which makes it extremely inefficient. Luckily, it is relatively simple to modify the algorithm to keep track of states it has seen before, and avoid adding them to the tree again:

```
procedure SOLVE(state)
    ...
    if nextState is solved then
        return RECONSTRUCTPLAN(child)
    else if nextState has not already been seen then
        record nextState as seen
        add child to q
    end if
    ...
end procedure
```

Make this modification to your algorithm. The `HashSet` is an appropriate data structure for keeping track of seen states.

Discuss: Why is `HashSet` an appropriate choice? C# imposes some requirements on what can be stored in a `HashSet` — find out what they are, and convince yourself that the provided `GameState` class satisfies them.

Further improvements

With this improvement properly implemented, your solver should manage to solve most levels. However there is still room for improvement. Here are some suggestions:

- Detect states which are unsolvable (e.g. where a box has been pushed into a corner) and exclude them from the search
- Move ordering — when searching the available actions from a state, prioritise e.g. actions which push a box over actions which do not
- Pathfinding — combine BFS with A* search, for example to create a single "action" which walks the shortest path to a given box

Discuss potential improvements to the search with your peers, and if you have time, try to implement one.