Project 1: Search Algorithms

Deadline: 2018-10-21

Abstract

The goal of this project is to help you better understand search algorithms and also to learn how to apply them to solve a "real" problem that can be represented as a state-space search problem.

Introduction

This project has two parts. In the first part, you will first code the main search algorithms covered in class. Then, you will apply them on the n-puzzle and plot their computation times for increasing n.

In the second part, you will use your knowledge of search algorithms to solve the Clickomania puzzle (see presentation below).

Part 1: Search Algorithms

In order to implement the search algorithms, we assume that a non-valued state-space graph is represented by a class with the following methods:

- `successors(state)` returns the list of successor states of `state`
- `isGoal(state)` returns `True` if `state` is an end state

When the state-space graph is valued (i.e., actions have costs), the first method is defined as follows:

- `successors(state)` returns for a given `state` a list of 2-tuples consisting of costs and a corresponding successor states (see example below)

These two methods define the interface of a state-space graph. Note that as we assume that actions are deterministic, they do not need to be stated specifically in the interface, because given the current state and the next state, the information of the action is redundant.

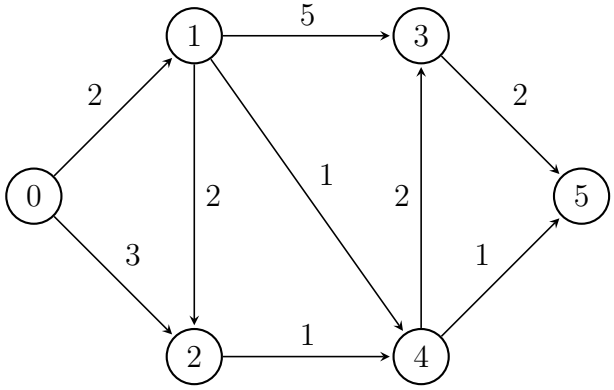

Figure 1: Small State-Space Graph.

We provide below some examples of state-space graphs created with such interface. The code for those examples is available in the file `testgraphs.py` and can be imported in your Python code.

Example 1. If the costs are not taken into account, the non-valued state-space graph depicted on Figure 1 can be coded as follows:

```
class SimpleGraph:
   """Simple graph for testing."""

   def successors(self, state):
      if state==0: return [1, 2]
      if state==1: return [2, 3, 4]
      if state==2: return [4]
      if state==3: return [5]
      if state==4: return [3, 5]
      if state==5: return []

   def isGoal(self, state):
      return state == 5
```

Recall that any method of a class always includes a first parameter (here, called `self`) that provides an access to the instance on which the method is invoked.

The corresponding valued state-space graph of Figure 1 can be implemented as follows:

```
class SimpleValuedGraph:
   """Simple graph for testing."""
```

```
   def successors(self, state):
      if state==0: return [(2, 1), (3, 2)]
      if state==1: return [(2, 2), (5, 3), (1, 4)]
      if state==2: return [(1, 4)]
      if state==3: return [(2, 5)]
      if state==4: return [(2, 3), (1, 5)]
      if state==5: return []

   def isGoal(self, state):
      return state == 5
```

Example 2. The n-puzzle can be implemented as follows:

```
class State:
   """State for nPuzzle graph that stores the position of the blank."""

   def __init__(self, v):
      self.value = v
      self.blankPosition = v.index(0)

   def clone(self):
      return State(list(self.value))

   def __repr__(self):
      """Converts an instance in string.

         Useful for debug or with print(state)."""
      return str(self.value)

   def __eq__(self, other):
      """Overrides the default implementation of the equality operator."""
      if isinstance(other, State):
         return self.value == other.value
      elif other==None:
         return False
      return NotImplemented

   def __hash__(self):
      """Returns a hash of an instance.

         Useful when storing in some data structures."""
```

```
      return hash(str(self.value))

class Action:
   """Feasible actions for nPuzzle graph and their effects."""

   LEFT = 0
   UP = 1
   RIGHT = 2
   DOWN = 3
   shift = [-1, -3, 1, 3] # Initilalized for n=3, can be overwritten for other n

   @classmethod # specifies that update is a class (not instance) method
   def update(cls, n):
      cls.shift[Action.UP] = -n
      cls.shift[Action.DOWN] = n

class nPuzzleGraph:
   """nPuzzle graph based on State and Action."""

   def __init__(self, n):
      Action.update(n)

   def setInitialState(self, initialState):
      self.initialState = initialState

   def succ(self, state, action):
      nextState = state.clone()
      shift = Action.shift[action]
      blankPosition = nextState.blankPosition
      nextState.value[blankPosition] = nextState.value[blankPosition+shift]
      blankPosition += shift
      nextState.value[blankPosition] = 0
      nextState.blankPosition = blankPosition

      return nextState

   def successors(self, state):
      succs = []
      if (state.blankPosition % self.n != 0):
         succs.append(self.succ(state, Action.LEFT))
      if (state.blankPosition >= self.n):
```

```
            succs.append(self.succ(state, Action.UP))
        if (state.blankPosition % self.n != self.n-1):
            succs.append(self.succ(state, Action.RIGHT))
        if (state.blankPosition < self.n*(self.n-1)):
            succs.append(self.succ(state, Action.DOWN))
        return succs

    def isGoal(self, state):
        return state.value == [1, 2, 3, 4, 5, 6, 7, 8, 0]
```

For this example, we defined a class `State` and a class `Action` to represents states and actions and other information related to them. To create an instance of this class, we need to pass a parameter n which represents the number of rows (or columns) of the $n \times n$-grid. Therefore, for the 8-puzzle, n is equal to 3.

A heuristic function is defined as a function that takes a state (i.e., `heuristic(state)`) and returns a non-negative value. Note that in Python, a function can be passed as an argument of another function.

Coding Tasks

You need to provide the code for the following search algorithms:

- Breadth-First Search (BFS)
- Uniform-Cost Search (UCS)
- Depth-First Search (DFS)
- Depth-Limited Search (DLS)
- Iterative Deepening Search (IDS)
- A* Search (A*)
- Monte Carlo Tree Search (MCTS)

They should respect the following names and signatures:

- BFS: `BFS(Graph, initialState)`
- UCS: `BFS(ValuedGraph, initialState)`
- DFS: `BFS(Graph, initialState)`

- DLS: `BFS(Graph, initialState, depthLimit)`
- IDS: `BFS(Graph, initialState)`
- A*: `Astar(ValuedGraph, initialState, heuristic)`

where `Graph` (resp. `ValuedGraph`) follows the interface defined above for non-valued (resp. valued) graphs, `initialState` is the initial state, `depthLimit` is the depth limit for DLS, and `heuristic` is a heuristic function. All those functions should return a 2-tuple whose first component is a path represented as a list of states from the initial state to the end state and whose second component is the cost of that path. In the non-valued case, this cost is the number of edges in the path and in the valued case, it is the sum of the costs of the edges of that path.

For MCTS, the function should be defined as follows:

- `MCTS(ValuedGraph, state, budget)`

where `Graph` (resp. `ValuedGraph`) follows the interface defined above for non-valued (resp. valued) graphs, `state` represents the current state, and `budget` is a positive integer specifying how many simulations should be performed. The function should return the next state to be chosen in `state`.

Important: In your implementation, you will access the state-space graph **only** via the interface defined by the two methods listed above.

All those search functions should be placed in a file called `search.py`. Your code that uses and tests those implementations should be in another file, which does not need to be part of your final submission. You can use the previous simple graph (or create your own) for your tests.

To evaluate the computational times of those algorithms, write a script saved in a file named `testtime.py` that runs all your implementations on the n-puzzle with increasing **odd** n starting from 3 and print the average running times for each algorithm (in the order listed above). The average is computed over 5 runs with different random solvable initial states. The format of the printed message is not important, but it should be clear which average corresponds to which algorithm. The max value of the odd n can be fixed when the computational times become greater than 30 minutes. For MCTS, you will of course call your function iteratively until you reach a goal state. You can terminate the running of a search if it lasts for more than one hour. In that case, the average running time of that algorithm for the corresponding n is NA.

Note that some initial states correspond to unsolvable states. To make sure that the initial state is solvable, the number of inversions should be

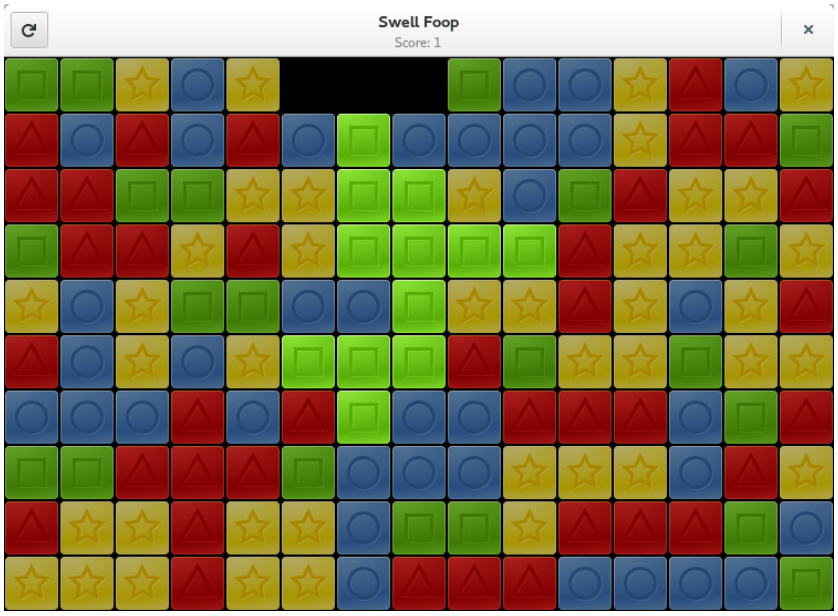

Figure 2: A version of Clickomania called Swell-Foop available in Gnome.

even. Assuming we are in state $[x_1, x_2, \ldots, x_n]$, an inversion happens when $x_i > x_j$ and $i < j$. For instance, the end state has an even number (i.e., 0) of inversion. State $[7, 2, 4, 5, 0, 6, 8, 3, 1]$ has 16 inversions and is thus a solvable initial state.

Part 2: Clickomania

Clickomania[1] is a tile-matching puzzle game known under many names, e.g., ChainShot! or Swell-Foop. See Figure 2 for a screenshot. The game mechanics is described as follows on Wikipedia:

> SameGame is played on a rectangular field, typically initially filled with four or five kinds of blocks placed at random. By selecting a group of adjoining blocks of the same color, a player may remove them from the screen. Blocks that are no longer supported will fall down, and a column without any blocks will be trimmed away by other columns always sliding to one side (often the left). The goal of the game is to remove as many blocks from the playing field as possible.

[1]If you want to try the game, http://www.8games8.com/puzzle/clickomania. Adobe Flash needed.

We will make the following assumptions:

- the game is played on a $N \times M$ grid
- there are K different colors
- during one move, the minimum tiles that can be removed is two

In this game, the goal is to maximize a score, which is obtained as follows. The initial score is zero. After one move, if $n \geq 2$ tiles are removed, the score is increased by $(n-1)^2$ points. The end game is reached when no tiles can be removed anymore. At that stage, the player receives a penalty: for each color C, the score is reduced by $(m-1)^2$ where m is the number of remaining tiles (regardless of their positions) of that color C.

Coding Tasks

You will code this game as a valued state-space graph as described in Part 1. The obtained class should be called `Clickomania`. An instance of it will also store as attributes the three parameters: N, M and K. A state is identified to a grid which is simply represented as a list of length $N \times M$ where each component is a nonnegative integer value smaller than or equal to K. The values 1 to K are used to represent the different colors and 0 encodes an empty cell. This class should be saved in a file `clickomania.py`.

Beside that class, you need to provide a function called `clickomaniaPlayer` that takes as arguments an instance of `Clickomania` and an initial state. It returns a 2-tuple whose first component is a sequence of states from the initial state to the end game and whose second component is the final score. This function should be saved in a file named `clickomaniaplayer.py`.

For the definition of this solving function, you are free to use any search algorithms (or variant of them). Notably you can reuse the work you did in Part 1. For instance, if you use the A* algorithm, you can define any heuristic function you want. In that case, the heuristic function should also naturally be saved in `clickomaniaplayer.py`. Your goal is to provide the best solver you can.

Submission and Due Date

You need to submit a zipped file named `P1-Firstname-Lastname.zip` where you replace `Firstname` and `Lastname` by your first name and last name respectively. This compressed file should contain all the source code files mentioned above and a **short** report in pdf format containing:

- for Part 1, a plot of the computational times for part 1 and some comments about the results you obtained,
- for Part 2, an explanation of the approach you adopted to solve the puzzle.

The submission will be by email to the TAs and the instructor. The due date is 11:59 pm on Oct 21st, 2018, as verified by the email reception time. There will be a penalty of 20% per day for late submission. Note that submissions will be checked for plagiarism.

Grading

Your program will be graded along three criteria:

1. Functional Correctness
2. Implementation Constraints
3. Report

An example of Functional Correctness is whether or not your algorithms produce the correct output. You should naturally test your program extensively, as they may be tested on a different problem that the two domains presented in this project. Implementation Constraints checks whether you stick to the implementation requirements. The length of the report has not much importance: we prefer clear and concise comments/explanations.