sheet04-programming

May 20, 2024

Exercises for the course Artifical Intelligence Summer Semester 2024

G. Montavon Institute of Computer Science Department of Mathematics and Computer Science Freie Universität Berlin

Exercise Sheet 4 (programming part)

[22]:
```
%matplotlib inline
import numpy
import copy
import utils
```

0.1 Exercise 2: Maximin (40 P)

In this exercise, we would like to implement the minimax procedure studied in the theoretical exercise where the first agent (Agent A) needs to collect as many coins possible in presence of another possibly adversarial agent (Agent B). As a reminder, the game is illustrated below:

The following variables (stored in `utils.py`) encode the game and its rules

- `utils.turns` encodes the sequence by which the different agents (A and B) set or update their positions on the grid-world,
- `utils.neighbors` encodes the connectivity between the tiles of the grid-world, and
- `utils.coins_tiles` encodes the coins initially on each tile before the agents have been placed and moved in the grid-world.

[23]:
```
utils.turns
```

[23]:
```
'ABABBA'
```

[24]:
```
utils.neighbors
```

[24]:
```
{'1': {'2', '3'},
 '2': {'1', '4'},
 '3': {'1', '4'},
 '4': {'2', '3', '5'},
 '5': {'4', '6'},
 '6': {'5'}}
```

[25]:
```
utils.coins_tiles
```

[25]:
```
{'1': 2, '2': 0, '3': 10, '4': 0, '5': 0, '6': 0}
```

We note as in the theoretical part of the exercise that the state of the game and the utility that results from it is uniquely determined by the sequence of positional updates of the two players. In particular, one defines the variables $X_1, \ldots, X_6$ where X_1 and X_2 are the initial positions of agents A and B, X_3 is the position reached by agent A after its first move, X_4 and X_5 is the position reached by agent B in its first and second move, and X_6 is the position reached by agent A in its second move. Hence, the variable X can be represented by a numerical string of length 6, e.g. `X = '251423'`, and we use smaller strings for denoting partial assignments (we will use for this the variable name `partialX`).

[26]:
```
utils.nbvars
```

[26]:
```
6
```

0.1.1 Searching for legal plays

The folloging code implements a stantard tree-search (similar to those studied in Lecture 3), where variables of the CSP are instantiated in the original order $(X_1, X_2, \ldots, X_6)$ with backtracking (i.e. only setting variables to values that satisfy the local constraints). The following function detemines the values of X_{i+1} given the partial assignment $(X_1, \ldots, X_i)$.

[27]:
```
def legal(partialX):

    assert(len(partialX)>=2)

    pos = {
        'A': None,
        'B': None,
    }

    nextturn = utils.turns[len(partialX)]

    for x,turn in zip(partialX,utils.turns):
        pos[turn] = x

    if nextturn == 'A':
        return utils.neighbors[pos['A']] - set(pos['B'])

    if nextturn == 'B':
        return utils.neighbors[pos['B']] - set(pos['A'])
```

The tree can now be searched by applying depth-first-search (DFS) and using the function above for backtracking. When the algorithm reaches a leaf (depth `nbvars`) it returns the full variable assignment. When reaching another node, it prints the partial assignment.

[28]:
```
def explore(partialX):
    if len(partialX) == utils.nbvars:
        X = partialX
```

```
        print(X+" *")
    else:
        print(partialX)
        utilities = []
        for position in legal(partialX):
            explore(partialX+position)

explore('25')
```

```
25
251
2514
25142
251423 *
25145
251452 *
251453 *
25143
251432 *
2516
25165
251652 *
251653 *
254
2546
25465
254652 *
254653 *
```

We would like now to find a strategy, i.e. a mapping from a given game's state (or partial assignment) to the next move for Agent A, that yields maximum utility. The strategy should assume that Agent B plays rationally to minimize Agent A's utility. In other words, we would like to optimize Agent A's worst-case utility. For this, we will first implement the utility function that maps (full) variable assignment to the corresponding utility for Agent A.

0.1.2 Implementing the Utility Function (20 P)

Implement a function that takes as input a full assignment X, and returns the corresponding utility for Agent A.

[29]:
```
def calculate_utility(X):

    if len(X) != utils.nbvars:
        raise Exception('Not a full assignment')

    A_turns = [X[index] for index, item in enumerate(utils.turns) if item ==␣
↪'A']
    coins = [utils.coins_tiles[tile] for tile in A_turns]
```

```
    return numpy.sum(coins)
```

Your utility function can be tested on some legal sequence of moves obtained with the code above:

[30]:
```
print(calculate_utility('254653'))
print(calculate_utility('251423'))
print(calculate_utility('251652'))
```

```
10
12
2
```

Different sequences of moves yield different utility scores for Agent A.

0.1.3 Implementing the Maximin Procedure (20 P)

Now, we would like to implement the maximin procedure. It can be seen as an extension of the `explore` function above. Specifically, it should additionally compute for each leaf of the tree the corresponding utility function, and then propagate the scores upwards in the tree following the maximin rule, i.e. computing the maximum utility if the corresponding move is decided by agent A and the minimum utility if the move is decided by agent B. Additionally, your function should keep track of the moves associated to these computations.

[32]:
```
def maximin(partialX):
    depth = len(partialX)
    if depth == utils.nbvars:
        utility = calculate_utility(partialX)
        print(f'X = {partialX}; U = {utility}')
        return utility, ''

    print(f'X = {partialX}')

    nextturn = utils.turns[len(partialX)]
    legal_moves = legal(partialX)
    best_move = None

    if nextturn == 'A':
        max_util = float('-inf')
        for position in legal_moves:
            util, _ = maximin(partialX + position)
            if util > max_util:
                max_util = util
                best_move = position
        print(f'X = {partialX}; U = {max_util}; next = {best_move}')
        return max_util, best_move
    else:
        min_util = float('inf')
        for position in legal_moves:
            util, _ = maximin(partialX + position)
```

```
            if util < min_util:
                min_util = util
                best_move = position
        print(f'X = {partialX} ; U = {min_util}; next = {best_move}')
        return min_util, best_move
```

Your implementation can now be tested with the given initial state (where agents A and B start on tiles 2 and 5 respectively).

[33]:
```
maximin('25')
```

```
X = 25
X = 251
X = 2514
X = 25142
X = 251423; U = 12
X = 25142; U = 12; next = 3
X = 25145
X = 251452; U = 2
X = 251453; U = 12
X = 25145; U = 12; next = 3
X = 25143
X = 251432; U = 2
X = 25143; U = 2; next = 2
X = 2514 ; U = 2; next = 3
X = 2516
X = 25165
X = 251652; U = 2
X = 251653; U = 12
X = 25165; U = 12; next = 3
X = 2516 ; U = 12; next = 5
X = 251 ; U = 2; next = 4
X = 254
X = 2546
X = 25465
X = 254652; U = 0
X = 254653; U = 10
X = 25465; U = 10; next = 3
X = 2546 ; U = 10; next = 5
X = 254 ; U = 10; next = 6
X = 25; U = 10; next = 4
```

[33]:
```
(10, '4')
```

We observe that the worst-case utility Agent A gets by playing rationally is 10, and it is guaranteed to reach this utility by moving first to tile 4 (i.e. $X_3 = 4$).

Suppose now that Agent A indeed plays the move $X_3 = 4$, and then the agent B moves successively to tiles 6 and 5 (i.e. $X_4 = 6$ and $X_5 = 5$). Agent A can find the best next move to play next by

running the code.

[34]:
```
maximin('25465')
```

```
X = 25465
X = 254652; U = 0
X = 254653; U = 10
X = 25465; U = 10; next = 3
```

[34]:
```
(10, '3')
```

The output of this program tells us that the best worst-case outcome for Agent A is still 10 and this outcome can be obtained if Agent A now moves to tile 3 (i.e. $X_6 = 3$). After this move, the game terminates, and one can observe that Agent A has indeed reached an utility of 10.

[35]:
```
calculate_utility('254653')
```

[35]:
```
10
```