Intelligent Systems 2020: First practical assignment

Uninformed Search

Your name: Seeun Park

Your VUNetID: spk760

If you do not provide your name and VUNetID we will not accept your submission.

Learning objectives

At the end of this exercise you should be able to understand the implementations of the basic algorithms for uninformed search, BFS, and DFS. You should be able to:

1. Understand the algorithms (be able to explain in your own words)
2. Follow the individual steps of the algorithms
3. Make small modifications of the code to see the effect on the search algorithms
4. Make small adaptations to the algorithm to study the computational properties

Practicalities

Follow this Notebook step-by-step.

Of course, you can do the exercises in any Programming Editor of your liking. But you do not have to. Feel free to simply write code in the Notebook. Please use your studentID+Assignment1.ipynb as the name of the Notebook.

Note: unlike the courses dedicated to programming we will not evaluate the style of the programs. But we will, however, test your programs on other data that we provide, and your program should give the correct output to the test-data as well.

As was mentioned, the assignment is graded as pass/fail. To pass you need to have either a full working code or an explanation of what you tried and what didn't work for the tasks that you were unable to complete (you can use multi-line comments or a text cell).

Initialising

First, let us make sure the necessary packages are installed, and imported. Run the following code:

In [1]:
```
import sys
!{sys.executable} -m pip install numpy
import datetime
import numpy as np
from numpy import random
from collections import deque
```

```
from utils import *

# This might produce a warning that numpy is already installed.
```

```
Requirement already satisfied: numpy in c:\users\seeun\appdata\local\programs\python\pyt
hon39\lib\site-packages (1.19.3)
```

Starting your first game

In [2]:
```
HAND_SIZE = 7      #TODO: replace with your desired hand size (should not be higher tha
STUDENT_NUMBER = 2701501 #TODO: replace with your own student number
```

With the constant HAND_SIZE we decide how many cards we want in our hand. By default it is set to 5, you can change it to any hand size, do keep in mind that the higher the number of cards in your hand, the more branches in the search tree there are, and the longer it will take to run.

Your student number is used to set a random seed. There are situations imaginable where you want a pseudo random selection (for example when debugging it's nice to always work with the same values) In short, the seed ensures that you get a pseudo random distribution that will always be the same when you re-run the code. It is a random distribution because you don't have to hard code them in yourself, but it is not random in the sense that the next time you run it you get different cards! For more information on pseudo random number generators, check out https://www.geeksforgeeks.org/pseudo-random-number-generator-prng/.

Task 1

You may wonder how the cards for this game are represented. Go to utils and find out in which variable this information is found, print this variable below:

In [3]:
```
print(representation)
```

```
        h  d  s  c
ace  [[ 0  1  2  3]  h = hearts
ten   [ 4  5  6  7]  d = diamonds
king  [ 8  9 10 11]  s = spades
queen [12 13 14 15]  c = clubs
jack  [16 17 18 19]]

For example: '10' is 'king of spades'
```

Some support functions

Next, there are some functions we need for the implementation. Try to get the gist of what they do, but if you do not understand fully, don't worry. The first one tests whether a move is valid (so whether a card follows suit, or has the same value). The second is a helper function that checks whether two cards have the same suit, and the third function checks whether two cards have the same value. The last one makes a random choice of cards in the hands.

We don't expect you to fully understand the coding behind these functions, however, if you are interested you might find the following link useful: https://www.programiz.com/python-

programming/matrix. Take a look at how Python Matrices are created and how to access rows and columns.

In [4]:
```
def valid_move(cardA, cardB):
    #print("validMove: comparing " + str(cardA) + " to " + str(cardB)) UNCOMMENT THIS T
    g = np.arange(20).reshape(5, 4) #this produces the same grid as the representation,
    if check_value(cardA, cardB, g):
        return True
    elif check_suit(cardA, cardB, g):
        return True
    else:
        #print("validMove: No move found")
        return False

def check_suit(cardA, cardB, grid):
    r, c = grid.shape
    for i in range(c):
        if np.any(grid[:, i] == cardA) and np.any(grid[:, i] == cardB):
            return True

def check_value(cardA, cardB, grid):
    r, c = grid.shape
    for i in range(r):
        if np.any(grid[i] == cardA) and np.any(grid[i] == cardB):
            return True

def pick_cards(seed, size):
    random.seed(seed)
    cards = np.random.choice(20, (size*2), replace = False)
    leftHand = cards[:size]
    rightHand = cards[size:]
    return (leftHand, rightHand)
```

Main Search algorithms

Now it is time to define BFS. Try to understand the implementation. It is as close to the pseudocode presented in class as possible.

In [5]:
```
def breadth_first_tree_search(problem):
    """
    Search the shallowest nodes in the search tree first.
    Search through the successors of a problem to find a goal.
    The argument fringe should be an empty queue.
    Repeats infinitely in case of loops.
    """

    fringe = deque([Node(problem.initial)])  # FIFO queue
    while fringe:
        node = fringe.popleft()
        if problem.goal_test(node.state):
            print("############")
            print("success!")
            print("The solution is: {}".format(node.solution()))
            return node
        fringe.extend(node.expand(problem))
    print("############")
    print("unfortunately no solution has been found!")
    return None
```

And here comes DFS. Again, check out the code, and look at the difference between the two implementations.

In [6]:
```
def depth_first_tree_search(problem):
    """
    Search the deepest nodes in the search tree first.
    Search through the successors of a problem to find a goal.
    The argument fringe should be an empty queue.
    Repeats infinitely in case of loops.
    """

    fringe = [Node(problem.initial)]  # Stack
    while fringe:
        node = fringe.pop()
        if problem.goal_test(node.state):
            print("############")
            print("success!")
            print("solution: {}".format(node.solution()))
            return node
        fringe.extend(node.expand(problem))
    print("############")
    print("unfortunately no solution has been found!")
    return None
```

Task 2

Explain in your own words the difference between the implementations. Write your explanation into the following cell:

The function 'popleft' removes and returns the element placed in the most left, while 'pop' uses the last added node only allowing the player to pick the uppermost card and return it. This is the difference in between the two implementations. DFS uses LIFO where as BFS uses FIFO allowing the two different searches to search either from the first node in and out or the last node in and first node out.

The cell below generates the cards for your left and right hand.

In [7]:
```
leftHand, rightHand = pick_cards(STUDENT_NUMBER, HAND_SIZE)


print("Right hand: {}".format(rightHand))
print("Left hand: {}".format(leftHand))
```

```
Right hand: [ 6 18  4  8  0  9 10]
Left hand: [13 11  2  7  5 14  3]
```

Constructing your own game

Of course you don't have to (always) use random cards, you can also choose your own set of cards. Just make sure the numbers (integers) in the arrays are unique and between 0 and 19. Uncomment and define the variable values. Also make sure that the array is of correct size (according to the previous tasks).

In [8]:
```
#####
customLeftHand = [4, 11, 13, 3, 9, 6, 15]
```

```
customRightHand = [14, 18, 5, 1, 2, 7, 10]
#####
```

Here we specify our initial state of the problem. Again, uncomment and fill in/correct the code.

In [9]:
```
initialState = GameState(customLeftHand, customRightHand, True, True)
```

Task 3

Look in the utils file how you can print the state, and print the initial state below (make sure you've specified HAND_SIZE and STUDENT_NUMBER or the custom hands).

Hint: to call a function from a class use class.function(paramaters) or if the class was previously called use variable.function(). For more information check https://www.geeksforgeeks.org/python-call-function-from-another-function/.

In [10]:
```
initialState.printState()
```

```
------
GameState: Printing state:
Left hand: [4, 11, 13, 3, 9, 6, 15]
Right hand: [14, 18, 5, 1, 2, 7, 10]
Do we play from left hand to get to next state? True
------
```

This cell constructs the problem for your game.

In [11]:
```
p = Problem(initialState)
```

We can now run experiments on how long the search took by looking at the time difference in microseconds between the start and finish of the search.

In case your game does not have a solution, you might want to try to run a custom-made set of hands as presented above.

In [12]:
```
startTime = datetime.datetime.now()

#search = breadth_first_tree_search(p)
search = depth_first_tree_search(p)

endTime = datetime.datetime.now()

duration = endTime-startTime
print("The search took {} microseconds".format(duration.microseconds))
```

```
############
success!
solution: [15, 14, 13, 5, 4, 7, 11, 10, 9, 1, 3, 2, 6, 18]
The search took 960571 microseconds
```

Task 4

In the next section a text file will be generated. To ensure all answers are saved into it, it is important to follow the instructions carefully, especially when naming the variables!

Using the representation matrix of the cards, think of a combination of two hands for which no solution would be found (minimal hand size = 3).

To save your solution, declare two lists, named “noLeft” and “noRight” and populate them with suitable cards.

In [13]:
```
noLeft = [6, 11, 13, 3, 9]
noRight = [14, 18, 5, 1, 2]
```

Task 5

Implement a counter that counts how many nodes had to be generated before reaching a goal state.

Run the Depth First Search algorithm based on your student number (set as seed) and hand size of 5. Have the algorithm return or print the counter. Save your solution in the variable below.

Hint: take a look at the code of the algorithm and find where exactly the new node is generated.

In [14]:
```
def depth_first_tree_search(problem):
    """
    Search the deepest nodes in the search tree first.
    Search through the successors of a problem to find a goal.
    The argument fringe should be an empty queue.
    Repeats infinitely in case of loops.
    """

    nodeCount = 0

    fringe = [Node(problem.initial)]  # Stack
    while fringe:
        node = fringe.pop()
        nodeCount += 1
        if problem.goal_test(node.state):
            print("############")

            print("Node Count: %d" % nodeCount)

            print("success!")
            print("solution: {}".format(node.solution()))
            return node
        fringe.extend(node.expand(problem))
    print("############")
    print("unfortunately no solution has been found!")
    return None

nodeCount = depth_first_tree_search(p)
```

```
############
Node Count: 894
success!
solution: [15, 14, 13, 5, 4, 7, 11, 10, 9, 1, 3, 2, 6, 18]
```

Task 6

Report briefly on if (and if so, how):

```
    * a) changing the order of the cards in your hand
    * b) choosing depth first or breadth first search
```

...individually and combined have an influence on the running time and nodes generated. Save this (brief!) report in a multi line string variable named “myReport”.

Note: If your student number does not generate hands for which a solution can be found, pick a custom hand (be sure to implement it above, and run all the cells again to make sure it gets executed properly).

In [15]:
```
myReport = """Changing the order of the cards has an affect on both algorithms because
starts with the most left node, if this were to happen for DFS and the goalstate was wi
depth of the first node then DFS would have the possibility of having a shorter time co
general DFS is appropriate to use for situations when playing schnapsen because the sea
depth of the search. If there were a infinite path or loop that the algorithm would get
DFS, it would have been inefficient to use BFS despite the long running time. However,
infinite paths for this game DFS will take a shorter running time and display the solut
```

Task 7

Using the representation of the cards and your knowledge of the search algorithms, define a pair of hands that will be faster with depth first search (hand size = 3).

Save your solution by declaring 2 lists named "DFSL" (left hand) and "DFSR" (right hand).

In [16]:
```
# HAND_SIZE = 3
DFSL = [0, 4, 8]
DFSR = [1, 5, 9]
```

Task 8

Explain why it is not possible to define a pair of hands such that Breadth First Search is faster than Depth First Search. Save your (brief) explanation as a multi-line string in the variable below

In [17]:
```
MyReport2 = """Due to the fact that the game schnapsen has a minimum requirement must
game in terms of the amount of tricks or actions that need to be used. In addition, BF
tree has a high branching factor, and as since there is no possibility of an infinite
the DFS the solution will always be found under the circumstances that there is a solu
proximated that DFS will be able to provide a solution in a shorter amount of time in
it will search into the depths of the tree."""
```

Final Task: Collect all the results

Uncomment and run this cell (and all the cells above) to generate the text file that you have to hand in together with the notebook on canvas!

In [19]:
```
exportToText(STUDENT_NUMBER, noLeft, noRight, nodeCount, myReport, DFSL, DFSR, MyReport
```