Intelligent Systems 2020: Second practical assignment

Informed 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 informed search. 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+Assignment2.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 functools
import heapq
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 = 3      #TODO: replace with your desired hand size (3, 5 or 10)
STUDENT_NUMBER = 2579950 #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/.

Updated representation

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]  = 11pts     h = hearts
ten   [ 4  5  6  7]  = 10pts     d = diamonds
king  [ 8  9 10 11]  = 4pts      s = spades
queen [12 13 14 15]  = 3pts      c = clubs
jack  [16 17 18 19]] = 2pts
For example: '10' is 'king of spades'
```

Note that cards have values now w.r.t. the game from Assignment 1

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 returns the points value of a card, the second tests whether a move is valid (so whether a card follows suit, or has the same value). The third is a helper function that checks whether two cards have the same suit, and the fourth 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 get_points(cardA):
    g = np.arange(20).reshape(5, 4) #this produces the same grid as the representation
    if (np.any(g[0] == cardA)): #Ace (11)
        return 11
    elif (np.any(g[1] == cardA)): #Ten (10)
        return 10
    elif (np.any(g[2] == cardA)): #King (4)
        return 4
    elif (np.any(g[3] == cardA)): #Queen (3)
        return 3
    elif (np.any(g[4] == cardA)): #Jack (2)
        return 2

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)
```

Breadth, and Depth First Search

From assignment 1 we are already familiar with the Breadth First and Depth First search algorithms. See the code below:

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
    counter = 0
    while fringe:
        node = fringe.popleft()
        counter += 1
        if problem.goal_test(node.state):
            print("############")
            print("success!")
            print("solution: {}".format(node.solution()))
            print("{} Nodes generated".format(counter))
            return node, counter
        fringe.extend(node.expand(problem))
    print("############")
    print("unfortunately no solution has been found!")
    print("{} Nodes generated".format(counter))
    return None
```

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
    counter = 0
    while fringe:
        node = fringe.pop()
        counter += 1
        if problem.goal_test(node.state):
            print("############")
            print("succes!")
            print("solution: {}".format(node.solution()))
            print("{} Nodes generated".format(counter))
            return node, counter
        fringe.extend(node.expand(problem))
    print("############")
    print("unfortunately no solution has been found!")
    print("{} Nodes generated".format(counter))
    return None
```

Preparing the game

To run a search, we need to define an initial state. Run the cells below to generate hands automatically, or define a custom set of hands. Please note that if you use a custom set of hands, you need to replace 'leftHand' and 'rightHand' with 'customLeftHand' and 'customRightHand' at gamestate initialisation.

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

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

print("Left hand: {}".format(leftHand)) #or customLeftHand
print("Right hand: {}".format(rightHand)) #or customRightHand
```

```
Left hand: [12  7 19]
Right hand: [ 0  4 16]
```

Now we define an initial state, and the problem.

In [8]:
```
initialState = GameState(leftHand, rightHand, True, True)
initialState.printState()

p = Problem(initialState)
print("There are {} points needed to win(reach goal state)".format(p.winPoints))
print("There are {} points in the game".format(sum(p.initial.allCardPoints)))
```

```
------
GameState: Printing state:
Left hand: [12  7 19]
Right hand: [ 0  4 16]
Do we play from left hand to get to next state? True
Points scored: 0
All cards: [12, 7, 19, 0, 4, 16]
All card points: [3, 10, 2, 11, 10, 2]
------
There are 20 points needed to win(reach goal state)
There are 38 points in the game
```

Comparing the points needed to win versus the total amount of points in the game, what do you notice? Return you (brief) findings to the string variable below:

In [9]:
```
myFindings = """The amount of points needed in order to win is lower than the total poi
consists of. Therefore, allowing the player to lose points but still have the opportuni
```

Task 1:

Run Breadth and Depth first search and implement a counter in the algorithm above that prints the number of nodes generated (as you did in the previous assignment, although now to print you must uncomment the print statements instead of formulating your own).

In [10]:
```
startTime = datetime.datetime.now()
breadth_first_tree_search(p)
endTime = datetime.datetime.now()
duration = endTime - startTime
print("The Breadth First Search Took {} Microseconds".format(duration.microseconds))

startTime = datetime.datetime.now()
depth_first_tree_search(p)
endTime = datetime.datetime.now()
duration = endTime - startTime
print("The Depth First Search Took {} Microseconds".format(duration.microseconds))
```

```
############
success!
solution: [7, 4]
8 Nodes generated
The Breadth First Search Took 20005 Microseconds
############
succes!
solution: [19, 16, 12, 4, 7]
6 Nodes generated
The Depth First Search Took 12019 Microseconds
```

Greedy Best First Search

See the code below for the Greedy Best First Search Algorithm

In [11]:
```
def greedy_best_first_search(problem, f, display=True):
    """Search the nodes with the lowest f scores first.
    You specify the function f(node) that you want to minimize; for example,
    if f is a heuristic estimate to the goal, then we have greedy best
    first search; """
    f = memoize(f, 'f')
    node = Node(problem.initial)
    frontier = PriorityQueue('min', f)
    frontier.append(node)
    explored = set()
    counter = 0
    while frontier:
        node = frontier.pop()
        counter += 1
        if problem.goal_test(node.state):
            if display:
                print("Search succesful!")
                print(len(explored), "paths have been expanded and", len(frontier), "pa
                print("solution: {}".format(node.solution()))
                print("{} Nodes generated".format(counter))
            return node
        explored.add(node.state)
        for child in node.expand(problem):
            if child.state not in explored and child not in frontier:
                frontier.append(child)
            elif child in frontier:
                if f(child) < frontier[child]:
                    del frontier[child]
                    frontier.append(child)
    print("Search failed")
    print("{} Nodes generated".format(counter))
    return None
```

Task 2

Run A Greedy Best First Search and implement a counter in the algorithm above that prints the number of nodes generated (as you did in the previous assignment, although now to print you must uncomment the print statements instead of formulating your own).

In [23]:
```
startTime = datetime.datetime.now()
greedy_best_first_search(p, p.h, True)
endTime = datetime.datetime.now()
duration = endTime - startTime
print("The Greedy Best First Search Took {} Microseconds".format(duration.microseconds)
```

```
Search succesful!
2 paths have been expanded and 2 paths remain in the frontier
solution: [7, 4]
3 Nodes generated
The Greedy Best First Search Took 5007 Microseconds
```

Task 3

In what way do Depth First Search, Breadth First Search and Greedy Best First Search (using the distance to the goal as heuristic) differ?

Return a (brief) report to the multi line string below, take into account the number of nodes counted from the counter you implemented:

In [13]:
```
myReport1 = """When comparing DFS to BFS, DFS costs less time and space, thus it expand
amount of nodes when reaching the goal state. On the other hand, BFS has a high branchi
a numerous amount of nodes therefore having a time complexity of b^(d+1) whereas DFS ha
these searches are an uninformed search, but GBFS is an informed search meaning that th
expanded is the least in comparison to the uninformed search algorithms. Moreover, GBFS
node closest to the goal state, thus resulting in a shorter shortest line distance than
inefficient in in comparison to GBFS, however when comparing DFS to GBFS the time compl
value, b^(m) but GBFS takes more space by b^(m)."""
```

Task 4

Run a Greedy Best First Search (again) but this time print the path cost (uncomment the print statement and fill in the right variable.)

(Hint: Go into utils.py, in class Node you can see a variable that should help you. Remember that a -successful- search returns a Node which is saved as 'search' here)

In [14]:
```
search = greedy_best_first_search(p, p.h)

print("Path cost: {}".format(search.path_cost))
```

```
Search succesful!
2 paths have been expanded and 2 paths remain in the frontier
solution: [7, 4]
3 Nodes generated
Path cost: 110
```

A* Search algorithm

See the code for the A star search below:

In [22]:
```
def astar_search(problem, h=None, display=True):
    """A* search is best-first graph search with f(n) = g(n)+h(n).
    You need to specify the h function when you call astar_search, or
    else in your Problem subclass."""
    h = memoize(problem.h, 'h')
    return greedy_best_first_search(problem, lambda n: n.path_cost + h(n), display)
```

Task 5

Consider the the solution of a greedy best first search and an a star search:

In [21]:
```
print("Greedy Best First Search: ")
gbf = greedy_best_first_search(p, p.h, True)
print("Path cost: {}".format(gbf.path_cost))

print("A Star Search: ")
```

```
ast = astar_search(p)
print("Path cost: {}".format(ast.path_cost))
```

```
Greedy Best First Search:
Search succesful!
2 paths have been expanded and 2 paths remain in the frontier
solution: [7, 4]
3 Nodes generated
Path cost: 110
A Star Search:
Search succesful!
11 paths have been expanded and 3 paths remain in the frontier
solution: [19, 16, 12, 4, 7]
12 Nodes generated
Path cost: 84
```

Explain in your own words what the difference between the two algorithms is, and why the solutions differ. Return your explanation to the multi line string below:

In [20]:
```
myExplanation = """The evaluation function for the greedy best first search(GBFS) is f(
for the A* search it is f(n) = h(n) + g(n). When f(n) estimates the total cost of the p
to get to the goal, h(n) is an estimate of the cost from the node to the goal. Finally
reach the node. The difference in the evaluation function is caused because the goal fo
the node which is closest to the goal state and A* search tries to avoid all expansions
lot in order to expand. Therefore, A* search never overestimates the cost to reach the
heuristic is admissible. In regards, to expanding and generating the nodes GBFS stores
memory because it does not need to keep the information of the nodes that have been exp
hand, A* search stores all the nodes which takes a lot of memory as can be seen by the
number of nodes generated. Despite the fact that A* uses up more memory than GBFS. Howe
complete, but GBFS is incomplete and non-optimal."""
```

Task 6

Implement the second heuristic as described on Canvas in the customHeuristic function. Remove the multi line string quotation marks when done to run.

In [19]:
```
def customHeuristic(n=lambda n:n):
#    print("using custom heuristic")

    total_add_up = sum(n.state.allCardPoints)

    number_of_cards = len(n.state.allCards)

    mincost = number_of_cards * (number_of_cards + 1)

    averagecardvalue = total_add_up / number_of_cards

    H2 = mincost - averagecardvalue

    #33 is points to win for hand size of 5, it depends on the hand size
    return(H2) #n is the node, n.state is its state, n.state.pointsScored = the points

p.H2 = customHeuristic

astar_search(p, p.H2)
```

```
Search succesful!
11 paths have been expanded and 3 paths remain in the frontier
solution: [19, 16, 12, 4, 7]
12 Nodes generated
```

Out[19]: <Node [][0]>

Task 7 (Optional)

Try to find a better heuristics than h1 and h2 and implement them as you did above.

In [ ]:

In [ ]: