CS550 Assignment 02

1. Explain the difference between a problem state and search state
   1. A problem state is the current state with available actions and information, such as cost and heuristics towards a goal state. On the other hand, search state are the states that are reached when doing the search algorithm from the initial state to the goal state.
2. 4.1; for 4.1c, ignore that the simulated annealing algo presented in figure 4.5 would terminate immediately when temperature is zero
   1. Hill-climbing
   2. Breadth First
   3. First Choice Hill-Climbing
   4. Depth-Fist
   5. Random Walk
3. Consider the problem of managing the reproduction of a critically endangered species in a captive environment. One fitness function could be the population size although more sophisticated ones might take into account indicators such as the number of births, mortalities, genetic diversity, and behavioral indicators of stress, fertility, etc. Assume that you are managing this population and that your facility has the following resources:

* Nm males
* Nf females
* Nd units of food/day
* Ne enrinchement items (toys to prevent boredom)

Devise a state representation that partitions the food, animals, and enrichment items amongst three habitat units. Propose a crossover function that is different from the one used on the N-queens problem that could be used in a genetic algo search if we could predict the fitness from your state

1. Consider the fairly simply problem of selecting 2 matching cards from a deck of cards with pictures on them. The pictures consist of moon, sun, and stars, and there are two instances of each card (6 cards in total). Legal moves are pick a card that has not been picked, and the goal state is matching the first card drawn. Sketch an and-or search tree for this problem. You do not need to draw the full tree, just enough to make it clear that you understand what the full tree would look like.
2. a

*` '''*

*Created on Feb 8, 2018*

*file: explored.py*

**@author:** *mroch*

*'''*

class **Explored**(object):

*"Maintain an explored set. Assumes that states are hashable"*

def **\_\_init\_\_**(*self*):

*"\_\_init\_\_() - Create an empty explored set"*

# uses python's dictionary to create dictionary

*self*.statedict = dict()

def **exists**(*self*, state):

*"""exists(state) - Has this state already been explored?*

*Returns True or False, state must be hashable*

*"""*

#initially assume state has not been explored

exist = False

# create a hash for current state

statehash = hash(state)

try:

#creates a list using hash as index

copydict = *self*.statedict

hashlist = copydict[statehash]

#search list

for states in hashlist:

if state == states: #if state exists in dictionary

exist = True

break

except KeyError:

exist = False

return exist

def **add**(*self*, state):

*"""add(state) - add given state to the explored set.*

*state must be hashable and we asssume that it is not already in set*

*"""*

# The hash function is a Python builtin that generates

# a hash value from its argument. Use this to create

# a dictionary key. Handle collisions by storing

# states that hash to the same key in a bucket list.

# Note that when you access a Python dictionary by a

# non existant key, it throws a KeyError

statehash = hash(state)

try:

#creates a list using hash as indexes

copydict = *self*.statedict

hashlist = copydict[statehash]

hashlist.append(state)

#if unable to add to list, add the state to dictionary

except KeyError:

copydict = *self*.statedict

copydict[statehash] = [state]

*self*.statedict = copydict

*'''@file: npuzzle.py'''*

from basicsearch\_lib02.tileboard import TileBoard

from basicsearch\_lib02.searchrep import Problem

class **NPuzzle**(Problem):

*"""*

*NPuzzle - Problem representation for an N-tile puzzle*

*Provides implementations for Problem actions specific to N tile puzzles.*

*"""*

def **\_\_init\_\_**(*self*, n, force\_state=None, \*\*kwargs):

*""""\_\_init\_\_(n, force\_state, \*\*kwargs)*

*NPuzzle constructor. Creates an initial TileBoard of size n.*

*If force\_state is not None, the puzzle is initialized to the*

*specified state instead of being generated randomly.*

*The parent's class constructor is then called with the TileBoard*

*instance any any remaining arguments captured in \*\*kwargs.*

*"""*

# Note on \*\*kwargs:

# \*\*kwargs is Python construct that captures any remaining arguments

# into a dictionary. The dictionary can be accessed like any other

# dictionary, e.g. kwargs["keyname"], or passed to another function

# as if each entry was a keyword argument:

# e.g. foobar(arg1, arg2, …, argn, \*\*kwargs).

*self*.puzzlesize = n

*self*.tileboard = TileBoard(n, force\_state = force\_state)

super(NPuzzle, *self*).\_\_init\_\_(*self*.tileboard, \*\*kwargs)

def **actions**(*self*, state):

*"actions(state) - find a set of actions applicable to specified state"*

#copies board state to return its actions

boardcopy = TileBoard(*self*.puzzlesize, force\_state=state.state\_tuple())

return boardcopy.get\_actions()

def **result**(*self*, state, action):

*"result(state, action)- apply action to state and return new state"*

#returns the applied action to the board with move command.

boardcopy = TileBoard(*self*.puzzlesize, force\_state=state.state\_tuple())

return boardcopy.move(action)

def **goal\_test**(*self*, state):

*"goal\_test(state) - Is state a goal?"*

#determines if it's solved with a copied board state

boardcopy = TileBoard(*self*.puzzlesize, force\_state=state.state\_tuple())

return boardcopy.solved()

*'''*

*Created on Feb 10, 2018*

*file: problemsearch.py*

**@author:** *mroch*

*'''*

from basicsearch\_lib02.searchrep import (Node, print\_nodes)

from basicsearch\_lib02.queues import PriorityQueue

from explored import Explored

from searchstrategies import (BreadthFirst, DepthFirst, Manhattan)

import time

def **graph\_search**(problem, verbose=False, debug=False):

*"""graph\_search(problem, verbose, debug) - Given a problem representation*

*(instance of basicsearch\_lib02.representation.Problem or derived class),*

*attempt to solve the problem.”””*

explored = Explored()

numnodes = 0

finished = False

#create initial state without parent or actions

currentnode = Node(problem, problem.initial)

#DepthFirst doesn't create the correct frontier with the default PriorityQueue()

if(problem.g == DepthFirst.g and problem.h == DepthFirst.h):

frontier = PriorityQueue(currentnode.get\_f())

else:

frontier = PriorityQueue()

frontier.append(currentnode)

while not finished:

currentnode = frontier.pop()

if debug: #will print all the steps of traversing thru search

print(currentnode.\_\_repr\_\_())

explored.add(currentnode.state)

if not currentnode.state.solved():

for nodes in currentnode.expand(problem):

numnodes += 1

if not explored.exists(nodes.state):

frontier.append(nodes)

finished = frontier.\_\_len\_\_() == 0

else:

finished = True

if verbose: #will print the solution steps per function description

if(currentnode.solution() == 0):

print(*"No solution found"*)

else:

print(*"Solution in "*, len(currentnode.solution()), *" moves"*)

print(*"Initial State"*)

print(problem.initial)

problemcopy = problem.initial

for moves in range(len(currentnode.solution())):

print(*"Move "*, moves + 1, *" - "*, currentnode.solution()[moves])

problemcopy = problemcopy.move(currentnode.solution()[moves])

print(problemcopy.\_\_repr\_\_())

return (numnodes, currentnode.solution())

*"""* *file: searchstrategies.py* *"""*

import math

from basicsearch\_lib02.board import \*

from basicsearch\_lib02.queues import \*

from basicsearch\_lib02.searchrep import \*

from basicsearch\_lib02.tileboard import \*

# For each of the following classes, create classmethods g and h

# with the following signatures

# @classmethod

# def g(cls, parentnode, action, childnode):

# return appropritate g value

# @classmethod

# def h(cls, state):

# return appropriate h value

class **BreadthFirst**(object):

*"BredthFirst - breadthfirst search"*

*@classmethod*

def **g**(cls, parentnode, action, childnode):

return parentnode.depth + 1

*@classmethod*

def **h**(cls, state):

return 0

class **DepthFirst**(object):

*"DepthFirst - depth first search"*

*@classmethod*

def **g**(cls, parentnode, action, childnode):

return (parentnode.depth + 1) \* -1

*@classmethod*

def **h**(cls, state):

return 0

class **Manhattan**(object):

*"Manhattan Block Distance heuristic"*

*@classmethod*

def **g**(cls, parentnode, action, childnode):

return parentnode.depth + 1

*@classmethod*

def **h**(cls, state):

goalstate = []

cost=0

#generically creates the goal state for N-puzzle

#ex: goalstate[1,2,3,4,None,5,6,7,8] for an 8-puzzle

for i in range(state.boardsize\*\*2):

goalstate.append(i + 1)

goalstate[(int)(state.boardsize\*\*2 / 2)] = None #add blank node

#shift the nodes after the None

for j in range((int)(state.boardsize\*\*2 / 2), (int)(state.boardsize\*\*2 - 1)):

goalstate[j + 1] = j + 1

goaltuple = tuple(goalstate) #force\_state only takes tuples

tempboard = TileBoard((int)(state.boardsize\*\*2 - 1), force\_state=goaltuple)

for row in range(state.boardsize):

for col in range(state.boardsize):

for row2 in range(state.boardsize):

for col2 in range(state.boardsize):

if(tempboard.board[row][col] == state.board[row2][col2]):

cost += abs(row-row2) + abs(col-col2)

return cost

*'''*

*driver for graph search problem*

*Created on Feb 10, 2018*

*file: driver02.py*

**@author:** *mroch*

*'''*

from npuzzle import NPuzzle

from basicsearch\_lib02.tileboard import TileBoard

from searchstrategies import (BreadthFirst, DepthFirst, Manhattan)

from problemsearch import graph\_search

import collections

import time

import searchstrategies

from statistics import (mean, stdev)

def **tic**():

*"Return current time representation"*

return time.time()

def **tock**(t):

*"Return time elapsed in sec since t where t is the output of tic()"*

return time.time() - t

def **driver**():

mannytime = []

mannysteps = []

mannynodesexp = []

breadytime = []

breadysteps = []

breadynodesexp = []

deppytime = []

deppysteps = []

deppynodesexp = []

for test in range(31):

print(*"Running Test "*, test + 1)

#Manhattan

mannyLoopTime = 2

while(mannyLoopTime > 1): #for the sake of shorter test times

tb = TileBoard(8)

mannypuzzle = NPuzzle(8, tb.state\_tuple(), g=Manhattan.g, h=Manhattan.h)

currenttime = tic()

mannygraph = graph\_search(mannypuzzle) # @@@@@TEST OTHER PARAMETERS@@@@@

mannyLoopTime = tock(currenttime)

mannytime.append(tock(mannyLoopTime))

#print(mannygraph) #to compare number of nodes expanded and solution moves

mannysteps.append(len(mannygraph[1]))

mannynodesexp.append(mannygraph[0])

#BreadthFirst

breadypuzzle = NPuzzle(8, tb.state\_tuple(), g=BreadthFirst.g, h=BreadthFirst.h)

currenttime = tic()

breadygraph = graph\_search(breadypuzzle) # @@@@@TEST OTHER PARAMETERS@@@@@

breadytime.append(tock(currenttime))

#print(breadygraph) #to compare number of nodes expanded and solution moves

breadysteps.append(len(breadygraph[1]))

breadynodesexp.append(breadygraph[0])

#DepthFirst

deppypuzzle = NPuzzle(8, tb.state\_tuple(), g=DepthFirst.g, h=DepthFirst.h)

currenttime = tic()

deppygraph = graph\_search(deppypuzzle) # @@@@@TEST OTHER PARAMETERS@@@@@

deppytime.append(tock(currenttime))

#print(deppygraph) #to compare number of nodes expanded and solution moves

deppysteps.append(len(deppygraph[1]))

deppynodesexp.append(deppygraph[0])

#to compare times for each loop

print(*"elapsed time: "*, tock(currenttime))

#Mean Values (time, steps, nodes expanded) of each search

mannytimemean = mean(mannytime)

mannystepsmean = mean(mannysteps)

mannynodesmean = mean(mannynodesexp)

breadytimemean = mean(breadytime)

breadystepsmean = mean(breadysteps)

breadynodesmean = mean(breadynodesexp)

deppytimemean = mean(deppytime)

deppystepsmean = mean(deppysteps)

deppynodesmean = mean(deppynodesexp)

print(*"Manhattan: "*)

print(*"Mean Steps: "*, mannystepsmean)

print(*"Mean Expansion: "*, mannynodesmean)

print(*"Mean Time: "*, mannytimemean)

print(*"StDev Steps: "*, stdev(mannysteps,mannystepsmean))

print(*"StDev Expansion: "*, stdev(mannynodesexp,mannynodesmean))

print(*"StDev Time: "*, stdev(mannynodesexp,mannynodesmean))

print(*"BreadthFirst: "*)

print(*"Mean Steps: "*, breadystepsmean)

print(*"Mean Expansion: "*, breadynodesmean)

print(*"Mean Time: "*, breadytimemean)

print(*"StDev Steps: "*, stdev(breadysteps))

print(*"StDev Expansion: "*, stdev(breadynodesexp))

print(*"StDev Time: "*, stdev(breadytime))

print(*"DepthFirst: "*)

print(*"Mean Steps: "*, deppystepsmean)

print(*"Mean Expansion: "*, deppynodesmean)

print(*"Mean Time: "*, deppytimemean)

print(*"StDev Steps: "*, stdev(deppysteps))

print(*"StDev Expansion: "*, stdev(deppynodesexp))

print(*"StDev Time: "*, stdev(deppytime))

if \_\_name\_\_ == *'\_\_main\_\_'*:

driver()

|  |  |
| --- | --- |
| **Statistics Output for 31 solved puzzles:**  Running Test 1  Running Test 2  Running Test 3  Running Test 4  Running Test 5  Running Test 6  Running Test 7  Running Test 8  Running Test 9  Running Test 10  Running Test 11  Running Test 12  Running Test 13  Running Test 14  Running Test 15  Running Test 16  Running Test 17  Running Test 18  Running Test 19  Running Test 20  Running Test 21  Running Test 22  Running Test 23  Running Test 24  Running Test 25  Running Test 26  Running Test 27  Running Test 28  Running Test 29  Running Test 30  Running Test 31  Manhattan:  Mean Steps: 20.741935483870968  Mean Expansion: 1373.3225806451612  Mean Time: 1519728875.861567  StDev Steps: 1.9658372590405997  StDev Expansion: 830.7399668206161  StDev Time: 830.7399668206161  BreadthFirst:  Mean Steps: 20.741935483870968  Mean Expansion: 311041.0967741936  Mean Time: 58.417898024282145  StDev Steps: 1.9658372590405997  StDev Expansion: 196449.73661734356  StDev Time: 38.55359098141843  DepthFirst:  Mean Steps: 20.741935483870968  Mean Expansion: 302791.1612903226  Mean Time: 59.18057694742757  StDev Steps: 1.9658372590405997  StDev Expansion: 189472.31089934957  StDev Time: 38.96559819669535 | **graphsearch(manhattan, debug=True) Output:**  Checking steps  f=16.0 (g=0.0 + h=16.0)  0 1 2  0 3 . 5  1 7 6 8  2 1 4 2  f=17.0 (g=1.0 + h=16.0)  0 1 2  0 3 6 5  1 7 . 8  2 1 4 2  f=17.0 (g=1.0 + h=16.0)  0 1 2  0 . 3 5  1 7 6 8  2 1 4 2  f=18.0 (g=2.0 + h=16.0)  0 1 2  0 3 6 5  1 7 4 8  2 1 . 2  f=18.0 (g=2.0 + h=16.0)  0 1 2  0 3 6 5  1 . 7 8  2 1 4 2  f=18.0 (g=2.0 + h=16.0)  0 1 2  0 7 3 5  1 . 6 8  2 1 4 2  f=17.0 (g=3.0 + h=14.0)  0 1 2  0 7 3 5  1 6 . 8  2 1 4 2  ... ... ... ...  ... ... ... ...  f=23.0 (g=15.0 + h=8.0)  0 1 2  0 3 2 5  1 1 . 7  2 6 4 8  f=23.0 (g=19.0 + h=4.0)  0 1 2  0 . 1 3  1 4 2 5  2 6 7 8  f=22.0 (g=20.0 + h=2.0)  0 1 2  0 1 . 3  1 4 2 5  2 6 7 8  f=21.0 (g=21.0 + h=0.0)  0 1 2  0 1 2 3  1 4 . 5  2 6 7 8 |

|  |  |
| --- | --- |
| **graph\_search(manhattan, verbose=True) Output:**  Running Test 1  solving Manhattan  Solution in 22 moves  Initial State  0 1 2  0 6 3 5  1 8 7 1  2 . 4 2  Move 1 - [-1, 0]  0 1 2  0 6 3 5  1 . 7 1  2 8 4 2  Move 2 - [0, 1]  0 1 2  0 6 3 5  1 7 . 1  2 8 4 2  Move 3 - [0, 1]  0 1 2  0 6 3 5  1 7 1 .  2 8 4 2  Move 4 - [-1, 0]  0 1 2  0 6 3 .  1 7 1 5  2 8 4 2  Move 5 - [0, -1]  0 1 2  0 6 . 3  1 7 1 5  2 8 4 2  Move 6 - [1, 0]  0 1 2  0 6 1 3  1 7 . 5  2 8 4 2  Move 7 - [1, 0]  0 1 2  0 6 1 3  1 7 4 5  2 8 . 2  Move 8 - [0, -1]  0 1 2  0 6 1 3  1 7 4 5  2 . 8 2  Move 9 - [-1, 0]  0 1 2  0 6 1 3  1 . 4 5  2 7 8 2  Move 10 - [-1, 0]  0 1 2  0 . 1 3  1 6 4 5  2 7 8 2 | Move 11 - [0, 1]  0 1 2  0 1 . 3  1 6 4 5  2 7 8 2  Move 12 - [0, 1]  0 1 2  0 1 3 .  1 6 4 5  2 7 8 2  Move 13 - [1, 0]  0 1 2  0 1 3 5  1 6 4 .  2 7 8 2  Move 14 - [1, 0]  0 1 2  0 1 3 5  1 6 4 2  2 7 8 .  Move 15 - [0, -1]  0 1 2  0 1 3 5  1 6 4 2  2 7 . 8  Move 16 - [0, -1]  0 1 2  0 1 3 5  1 6 4 2  2 . 7 8  Move 17 - [-1, 0]  0 1 2  0 1 3 5  1 . 4 2  2 6 7 8  Move 18 - [0, 1]  0 1 2  0 1 3 5  1 4 . 2  2 6 7 8  Move 19 - [0, 1]  0 1 2  0 1 3 5  1 4 2 .  2 6 7 8  Move 20 - [-1, 0]  0 1 2  0 1 3 .  1 4 2 5  2 6 7 8  Move 21 - [0, -1]  0 1 2  0 1 . 3  1 4 2 5  2 6 7 8  Move 22 - [1, 0]  0 1 2  0 1 2 3  1 4 . 5  2 6 7 8 |