Robert Werthman

CIS 579 Section 102

Professor Bruce Maxim

Table of Contents

[CIS 579 Assignment 2 1](#_Toc515789028)

[My approach to the problem 1](#_Toc515789029)

[Rationale for the expand (move generation) function 1](#_Toc515789030)

[Rationale for the heuristic for remaining distance to goal in A\* 1](#_Toc515789031)

[Comparison of A\* to Branch and Bound 2](#_Toc515789032)

# CIS 579 Assignment 2

## My approach to the problem

I decided to use Python for this assignment. I am more comfortable with it than Lisp. I also know how to use it in a debugger, and I know how to use its unit tests framework which allowed me to do test driven development with this assignment. I found out, though, that Python does not have the nice built-in functions for nested lists like Lisp does. Therefore, I had to figure out a different way to represent the states of problem state space without using a list.

Instead of using a list to represent the object “art---", I used strings. So instead of (art---) it becomes ‘art---‘. A partial path, then, is just a list of strings. Python provides a lot of useful functions to manipulate strings.

### Rationale for the expand (move generation) function

Given an initial state like ‘art---’ how do we get to a final state like ‘---tar’, and then how do we figure out what a next state looks like so that we can get to the final state? In the writeup, we were given operational constraints that told us how we could move from ‘ART---’ to ‘---ART’. I used these constraints to generate the possible moves a state could make. These constraints gave me the children of the state. For example, if we have the initial state ‘art---’ and the constraints from the write up, then the next states or children could be ‘a-tr--’ or ‘ar-t--‘. In terms a graph, these children would the neighbors of ‘art---’ and so the algorithm could then advance to these new states and generate even newer states from these. Eventually, one of the children will be the final state.

### Rationale for the heuristic for remaining distance to goal in A\*

After figuring out what the move generation function would look like, I used the pseudocode from the lecture notes for Branch and Bound to create Branch and Bound in Python. I wrote some helper functions and some tests and was able to solve the examples from the write-up with Branch and Bound.

A\* is different from Branch and Bound because it removes duplicates and sorts with the distance travelled from the initial state to the current state and the estimated distance from the current state to the final state. The code for removing duplicates wasn’t too difficult to find with a Google search, and the code to sort based on distance travelled was taken from Branch and Bound. But, figuring out what should be used as the estimated distance from the current state to the final state was difficult.

At first, I thought about using min edit distance. Min edit distance can be used to tell you how similar strings are. The fewer edits you need to make to a string to make it look like another string the more similar the string is. For example, if state1 is ‘art---’, state2 is ‘a-t--r’, and the final state is ‘---tar’, then min edit distance would say that state2 is more similar to the final state then state1 is because it takes fewer edits to the characters to get there. I thought this would be a good heuristic to tell me how close I was from the current state to the final state. The only problem is that if you had a state like ‘---atr’ and the final state was ‘---tar’, according to our constraints, we would never be able to reach the final state. But, min edit distance would say that ‘---atr’ is more similar to ‘---tar’ than ‘a-t--r’ even though we could never use ‘---atr’ to find a path to ‘---tar’. That gave me the idea to use the constraints in the writeup as an estimate of how close a state was to the final state. Given a string like ‘art---’, my idea would assign a number to it based on two things: if it was possible to eventually get from that state to the final state, and how many moves it would take to get there if just moving letters to the right. This fixed the problem with ‘---atr’ because my heuristic would assign a very large number to this state, making it so the algorithm would not choose it.

## Comparison of A\* to Branch and Bound

As you can see in Table 1, A\* performed significantly better than Branch and Bound when it came to the time to find a solution. A\* also did better keeping the size of the queue of partial paths smaller. The solutions found by the algorithms were the same length but the routes they took to get to the solutions were different.

Overall A\* was the better algorithm because it took less time to find a solution and used less space to find a solution. Although, I only used 3 strings for the testing. To get a better understanding of which algorithm is better for this state space problem, you would want to test with more strings.

Table - Results of comparing Branch and Bound to A\* with different strings

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Time to find path  A\* | Time to find path  Branch and Bound | Partial paths queue size  A\* | Partial paths queue size  Branch and Bound | Solution A\* | Solution  Branch and Bound |
| art---  to  ---tar | 0.054 seconds | 64.16 seconds | 10 paths | 172 paths | 'art---',  'a-tr--',  'a-t-r-',  'a-t--r',  'a--t-r',  '-a-t-r',  '--at-r',  '---tar' | 'art---',  'ar-t--',  'a-rt--',  'a--tr-',  '-a-tr-',  '-a-t-r',  '--at-r',  '---tar' |
| googleeyes---  to  google---eyes | 0.0049 seconds | 0.0087 seconds | 24 paths | 62127 paths | 'googleeyes---', 'googleey-se--', 'googleey--es-', 'google-ye-es-', 'google--eyes-', 'google--eye-s', 'google--ey-es', 'google--e-yes', 'google---eyes' | 'googleeyes---', 'googleey-se--', 'googleey-s-e-', 'google-yes-e-', 'google-ye-se-', 'google--eyse-', 'google--ey-es', 'google--e-yes', 'google---eyes' |
| abcd----  to  ----dcba | 0.01 seconds | 188.11 seconds | 23 paths | 60572 paths | 'abcd----',  'ab-dc---',  '-badc---',  '-bad-c--',  '-b-dac--',  '-b-d-ca-',  '-b-d-c-a',  '--bd-c-a',  '---dbc-a',  '---d-cba',  '----dcba' | 'abcd----',  'ab-dc---',  'ab-d-c--',  '-bad-c--',  '-b-dac--',  '--bdac--',  '--bd-ca-',  '---dbca-',  '---dbc-a',  '---d-cba',  '----dcba' |

### Code

### assignment2.py (source code)

import copy

def **expandElementsChildren**(queueElement):

*'''*

*Return the children of a state (queueElement)*

*'''*

queueElement = list(queueElement)

elementsChildren = []

for index, character in enumerate(queueElement):

# Create a copy of the queue element

copyQueueElement = queueElement[:]

if character is *'-'*:

# Don't do anything because this character means empty space

pass

else:

# Check if we are at the end of the list

if (index + 1) is len(queueElement):

# Do nothing

pass

# if there is an empty adjacent space move there

elif queueElement[index + 1] is *'-'*:

# Swap the letter with the adjacent space

copyQueueElement[index + 1] = copyQueueElement[index]

copyQueueElement[index] = *'-'*

# Add the child to the list that will be returned

elementsChildren.append(*""*.join(copyQueueElement))

# if we can jump a letter to the right like checkers

elif ((index + 2) is not len(queueElement) and

queueElement[index + 2] is *'-'*):

# Jump the letter to the right of the current letter

copyQueueElement[index + 2] = copyQueueElement[index]

copyQueueElement[index] = *'-'*

# Add the child to the list that will be returned

elementsChildren.append(*""*.join(copyQueueElement))

return elementsChildren

def **removeDuplicates**(queueOfPartialPaths):

*'''*

*Removes duplicate lists that have elements the same order*

*'''*

queueOfPartialPathsWithoutDuplicates = []

for path in queueOfPartialPaths:

if path not in queueOfPartialPathsWithoutDuplicates:

queueOfPartialPathsWithoutDuplicates.append(path)

return queueOfPartialPathsWithoutDuplicates

def **heuristicEstimateofDistanceToGoal**(currentState, finalState):

*'''*

*Estimates the distance from the current state to the final state*

*'''*

totalDistanceToGoal = 0

# Iterate backwards from the end of the current state

for i, currentStateCharacter in reversed(

list(enumerate(currentState))):

if currentStateCharacter == *'-'*:

# Do nothing since we won't move - characters

pass

else:

for j, finalStateCharacter in reversed(

list(enumerate(finalState))):

# if the characters are the same and the current

# state characters are still in valid positions

# according to the constraints

if currentStateCharacter == finalStateCharacter:

if i <= j:

# If the character in the currentState is

# already in the right position

if currentState[j] == finalState[j]:

# Remove the character from the final state

# so we can keep track of duplicates

tempListFinalState = list(finalState)

tempListFinalState[j] = *'-'*

finalState = *''*.join(tempListFinalState)

break

else:

# Track the number of moves we have to

# make to move the character to the right

# position

totalDistanceToGoal += (j - i)

# Remove the character from the final state

# so we can keep track of duplicates

tempListFinalState = list(finalState)

tempListFinalState[j] = *'-'*

finalState = *''*.join(tempListFinalState)

break

else:

# We know the current state will never lead to

# the final state because the characters can't

# move to the left

# For example, '-r-t-a' and '---art'

# Return a large number indicating this is

# a bad state

return (len(finalState) \* len(finalState))

return totalDistanceToGoal

def **solveWithAStar**(initialState, finalState):

*'''*

*A\**

*Add root to queue of partial paths*

*Until queue is empty or goal is attained*

*If first queue element equals the goal then do nothing*

*Else*

*remove the first queue element*

*add its children to the front of the queue of the partial paths*

*sort the queue of partial paths by distance traveled plus the estimate of distance to goal*

*remove redundant paths*

*If goal is attained then announce success*

*Function returns true if there is a path to the final state*

*from the initial state, false if there isn't a path*

*'''*

if len(initialState) < 3:

return False

# Add root to queue of partial paths

queueOfPartialPaths = [[initialState]]

# Until the queue is empty or goal is attained

while len(queueOfPartialPaths) > 0:

# If first queue element equals the goal

if queueOfPartialPaths[0][0] == finalState:

# Return the path from the initial state to the final state

result = queueOfPartialPaths[0]

# Reverse the path before returning it so that it goes

# initial state -> final state

# return result[::-1]

print *'A\*: Path solution'*, result[::-1]

print *'A\*: Size of queue of partial paths'*, len(queueOfPartialPaths)

return True

else:

# Remove the first queue element

firstList = queueOfPartialPaths.pop(0)

firstElement = firstList[0]

# if debugAStar: print 'First element', firstElement

# if debugAStar: print 'Heuristic cost of first element', heuristicEstimateofDistanceToGoal(firstElement, finalState)

# if debugAStar: print 'Length of first path', len(firstList)

# Adds its children to the front of the queue of the partial paths

firstElementsChildren = expandElementsChildren(firstElement)

# if debugAStar: print 'First elements children', firstElementsChildren

for child in firstElementsChildren:

copyOfFirstList = copy.deepcopy(firstList)

copyOfFirstList.insert(0, child)

queueOfPartialPaths.insert(0, copyOfFirstList)

# sort the queue of partial paths by distance traveled plus the estimate of distance to goal

# queueOfPartialPaths.sort(key=len)

queueOfPartialPaths = sortWithDistanceTravelledAndHeuristic(queueOfPartialPaths, finalState)

# remove redundant paths

queueOfPartialPaths = removeDuplicates(queueOfPartialPaths)

# return None

return False

def **sortWithDistanceTravelledAndHeuristic**(queueOfPartialPaths, finalState):

*'''*

*Sort the queue of partial paths by the length of the paths*

*and the hueristic of the most recent state*

*'''*

queueOfPartialPaths.sort(key = lambda x: len(x) + heuristicEstimateofDistanceToGoal(x[0], finalState))

return queueOfPartialPaths

def **timingWrapper**(func, initialState, finalState):

*'''*

*Create a wrapper function that we can send to timeit*

*because timeit only excepts functions*

*with no arguments but our functions we want to time*

*have arguments*

*'''*

def **timingWrapped**():

return func(initialState, finalState)

return timingWrapped

def **solveWithBranchAndBound**(initialState, finalState):

*'''*

*Branch and Bound*

*Add root to queue of partial paths*

*Until queue is empty or goal is attained*

*If first queue element equals the goal then do nothing*

*Else*

*remove the first queue element*

*add its children to the front of the queue of the partial paths*

*sort the queue of partial paths by distance traveled*

*If goal is attained then announce success*

*Function returns true if there is a path to the final state*

*from the initial state, false if there isn't a path*

*'''*

if len(initialState) < 3:

return False

# Add root to queue of partial paths

queueOfPartialPaths = [[initialState]]

# Until the queue is empty or goal is attained

while len(queueOfPartialPaths) > 0:

# If first queue element equals the goal

if queueOfPartialPaths[0][0] == finalState:

# Return the path from the initial state to the final state

result = queueOfPartialPaths[0]

# Reverse the path before returning it so that it goes

# initial state -> final state

print *'Branch and Bound: Path solution'*, result[::-1]

print *'Branch and Bound: Size of queue of partial paths'*, len(queueOfPartialPaths)

return True

else:

# Remove the first queue element

firstList = queueOfPartialPaths.pop(0)

firstElement = firstList[0]

# Adds its children to the front of the queue of the partial paths

firstElementsChildren = expandElementsChildren(firstElement)

for child in firstElementsChildren:

copyOfFirstList = copy.deepcopy(firstList)

copyOfFirstList.insert(0, child)

queueOfPartialPaths.insert(0, copyOfFirstList)

#Sort the queue of partial paths by distance traveled

queueOfPartialPaths.sort(key=len)

return False

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

pass

### assignment2\_test.py (test code)

import unittest

import timeit

from assignment2 import solveWithAStar

from assignment2 import solveWithBranchAndBound

from assignment2 import expandElementsChildren

from assignment2 import removeDuplicates

from assignment2 import heuristicEstimateofDistanceToGoal

from assignment2 import timingWrapper

from assignment2 import sortWithDistanceTravelledAndHeuristic

class **Test**(unittest.TestCase):

def **testAStarNO**(*self*):

result = solveWithAStar(*'no--'*, *'on--'*)

*self*.assertFalse(result)

def **testAStarTooSmallLength**(*self*):

result = solveWithAStar(*'no'*, *'on'*)

*self*.assertFalse(result)

def **testAStarART**(*self*):

result = solveWithAStar(*'art---'*, *'---tar'*)

*self*.assertTrue(result)

def **testBranchAndBoundART**(*self*):

result = solveWithBranchAndBound(*'art---'*, *'---tar'*)

*self*.assertTrue(result)

def **testBranchAndBoundNO**(*self*):

result = solveWithBranchAndBound(*'no--'*, *'on--'*)

*self*.assertFalse(result)

def **testBranchAndBoundTooSmallLength**(*self*):

result = solveWithBranchAndBound(*'no'*, *'on'*)

*self*.assertFalse(result)

def **testexpandElementsChildren**(*self*):

result = expandElementsChildren(*'art---'*)

*self*.assertListEqual(result, [*'a-tr--'*, *'ar-t--'* ])

def **testRemoveDuplicates**(*self*):

result = removeDuplicates([[*'art--'*, *'a-tr--'*], [*'art--'*, *'a-tr-t'*], [*'art--'*, *'a-tr--'*]])

*self*.assertListEqual(result, [[*'art--'*, *'a-tr--'*], [*'art--'*, *'a-tr-t'*]])

def **testHeuristicEstimateofDistanceToGoalStillPossible**(*self*):

result = heuristicEstimateofDistanceToGoal(*'art---'*, *'---tar'*)

*self*.assertEqual(result, 9)

def **testHeuristicEstimateofDistanceToGoalNotPossible**(*self*):

result = heuristicEstimateofDistanceToGoal(*'ar---t'*, *'---tar'*)

*self*.assertEqual(result, 36)

def **testHeuristicEstimateofDistanceToGoalString1**(*self*):

result = heuristicEstimateofDistanceToGoal(*'--t-ar'*, *'---tar'*)

*self*.assertEqual(result, 1)

def **testHeuristicEstimateofDistanceToGoalString2**(*self*):

result = heuristicEstimateofDistanceToGoal(*'a-tr--'*, *'---tar'*)

*self*.assertEqual(result, 7)

def **testsortWithDistanceTravelledAndHeuristic**(*self*):

result = sortWithDistanceTravelledAndHeuristic([[*'a-tr--'*, *'art---'*], [*'--t-ar'*, *'art--'*]], *'---tar'*)

*self*.assertListEqual(result, [ [*'--t-ar'*, *'art--'*], [*'a-tr--'*, *'art---'*]])

def **testTimingBranchAndBound**(*self*):

timingWrapped = timingWrapper(solveWithBranchAndBound, *'art---'*, *'---tar'*)

print *'Time Branch and Bound takes to find path from art--- to ---tar in seconds'*, timeit.timeit(timingWrapped, number=1)

timingWrapped = timingWrapper(solveWithBranchAndBound, *'googleeyes---'*, *'google---eyes'*)

print *'Time Branch and Bound takes to find path from googleyes--- to google---eyes in seconds'*, timeit.timeit(timingWrapped, number=1)

timingWrapped = timingWrapper(solveWithBranchAndBound, *'abcd----'*, *'----dcba'*)

print *'Time Branch and Bound takes to find path from abcd---- to ----dcba in seconds'*, timeit.timeit(timingWrapped, number=1)

def **testTimingAStar**(*self*):

timingWrapped = timingWrapper(solveWithAStar, *'art---'*, *'---tar'*)

print *'Time A\* takes to find path from art--- to ---tar to in seconds'*, timeit.timeit(timingWrapped, number=1)

timingWrapped = timingWrapper(solveWithAStar, *'googleeyes---'*, *'google---eyes'*)

print *'Time A\* takes to find path from googleeyes--- to google---eyes in seconds'*, timeit.timeit(timingWrapped, number=1)

timingWrapped = timingWrapper(solveWithAStar, *'abcd----'*, *'----dcba'*)

print *'Time A\* takes to find path from abcd---- to ----dcba in seconds'*, timeit.timeit(timingWrapped, number=1)

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

#import sys;sys.argv = ['', 'Test.testName']

unittest.main()