*AI for Games* – Chapter 4 (Part I)

Summary

This chapter focuses on pathfinding, particularly the best way to navigate.

Terms

1. Characteristic Point

Outline

Introduction:

1. The Pathfinding Graph
   1. Graphs
      1. Graphs are made up of nodes and edges.
      2. Each node represents an area of a game (e.g. level, platform)
   2. Weighted Graphs
      1. Each edge costs a certain amount.
      2. We measure distances as relative points.
      3. Mathematically, graphs could have negative weights, but not in actual programming.
   3. Directed Weighted Graphs
      1. A subcategory of weighted graphs where some paths are one-way.
   4. Terminology
      1. Terms in graphs will vary based on context.
   5. Representation
      1. Graphs need to be represented in code in order to affect them.
2. Dijkstra
   1. The Problem
      1. The graphs are named after mathematician, Edsger Dijkstra.
      2. This sorting pattern is used to find the shortest distance with a weighted graph.
   2. The Algorithm
      1. The starting point will find the shortest path to its neighbors. Once all neighbors are visited, the current point is transferred to its neighbors in shortest order
      2. This cycle repeats until the target is found.
      3. Used up nodes are placed in a list of what to follow in which order and which to stop checking for.
      4. When the list is empty, the algorithm will terminate.
   3. Pseudo Code
      1. Code should contain a smallestElement, a contains, and a find method.
   4. Data Structures and Interfaces
      1. A simple list is used at the end to return.
      2. A pathfinding list will add and remove entry based on weight and having visited.
   5. Performance of Dijkstra
      1. Worst case is O(nm)
      2. Most common is O(n2)
   6. Weaknesses
      1. Almost all options will have been checked, which can cost memory.
3. A\*
   1. The Problem
      1. Just like Dijkstra, the problem is finding the shortest route.
   2. The Algorithm
      1. We now try to find the most likely to arrive at the location first.
      2. We store the cost-so-far
      3. If cost-so-far is lower than existing value, we update the cost-so-far.
   3. Pseudo-Code
      1. A NodeRecord is added to keep track of nodes.
   4. Data Structures and Interfaces
      1. Pathfinding lists will regularly modify them
      2. Priority Queues sort the list of open nodes
      3. Priority Queues that are unlikely to be shortest will be “bucketed”
   5. Implementation Notes
      1. Cost values and assumptions will vary based on structure.
   6. Algorithm Performance
      1. The general performance is O(lm)
   7. Node Array A\*
      1. If nodes are numbered, we won’t have to search for them in two lists.
      2. Each node should have a record showing its cost so far, estimated cost, and current status.
   8. Choosing a Heuristic
      1. A\* behaves differently depending on amount of heuristic
      2. A low heuristic can be underestimated
      3. A high heuristic can be overestimated
4. World Representations
   1. Tile Graphs
      1. Like Chess or *Fire Emblem*, a grid of tiles that have a shortest distance.
      2. Tiles can be blocked out for more interesting maps
      3. Can be simple as generally each edge is the same cost
   2. Dirichlet Domains
      1. Characteristic points are possible locations in a 3D world for the AI to visit.
      2. Some nodes have larger pulls
      3. The shapes can make it tough to guarantee travelling wouldn’t cause errors from walking into other obstacles.
   3. Points of Visibility
      1. These points are chosen by characteristics that change over distance.
   4. Navigation Meshes
      1. 3D ares use special meshes to find paths.
      2. Floor polygons are used as regions and nodes\
   5. Non-Translational Problems
      1. Different orientations can have different connections
   6. Cost Functions
      1. We need to add different weights for different conditions
   7. Path Smoothing
      1. We need to use the path and predict actions to make sure turns are smooth
5. Improving on A\*
   * 1. It will take a lot of memory to use A\* on large environments
     2. It is a good idea to use hierarchical pathfinding

Questions

* Using the following graph and Dijkstra’s algorithm, determine the shortest path from node A to node G
  + A picture containing sky

    Description automatically generated

*Python –* Chapter 7

Summary

This chapter focuses on User Input and While Loops.

Outline

1. How the input() Function Works

**This program asks the user to type a message and repeats said message**

message = input("Tell me something, and I will repeat it back to you: ")

print(message)

This program asks the user to type a name, and states a message based on input.

name = input("Please enter your name: ")

print(f"\nHello, {name}!")

**This program uses multiple lines to ask the user a question, and receives info**

prompt = "If you tell us who you are, we can personalize the messages you see."

prompt += "\nWhat is your first name? "

name = input(prompt)

**This program takes the input and converts it into an int so it can be used mathmatically**

height = input("How tall are you, in inches? ")

height = int(height)

if height >= 48:

print("\nYou're tall enough to ride!")

else:

print("\nYou'll be able to ride when you're a little older.")

**This program uses a modulo operator to give a message based on the remainder**

number = input("Enter a number, and I'll tell you if it's even or odd: ")

number = int(number)

if number % 2 == 0:

print(f"\nThe number {number} is even.")

else:

print(f"\nThe number {number} is odd.")

1. Introducing while Loops

**This program will keep printing and adding to a number until the number is greater than 5:**

current\_number = 1

while current\_number <= 5:

print(current\_number)

current\_number += 1

**This program will keep repeating inputed messages until an inputed message is quit**

prompt = "\nTell me something, and I will repeat it back to you:"

prompt += "\nEnter 'quit' to end the program. "

message = ""

while message != 'quit':

message = input(prompt)

print(message)

**This program determines whether to keep going based on a bool that is turned to false by the statement break**

prompt = "\nPlease enter the name of a city you have visited:"

   prompt += "\n(Enter 'quit' when you are finished.) "

➊ while True:

city = input(prompt)

if city == 'quit':

break

else:

print(f"I'd love to go to {city.title()}!")

**This program will keep printing numbers until it gets to 10, but will avoid even numbers via a continue**

current\_number = 0

while current\_number < 10:

➊ current\_number += 1

if current\_number % 2 == 0:

continue

print(current\_number)

**This loop will start…and never….ends**

x = 1

while x <= 5:

print(x)

1. Using a While Loop with lists and Dictionaries

**This loop will move each user from the unconfirmed list into the confirmed list until the unconfirmed list is empty**

# Start with users that need to be verified,

# and an empty list to hold confirmed users.

➊ unconfirmed\_users = ['alice', 'brian', 'candace']

confirmed\_users = []

# Verify each user until there are no more unconfirmed users.

# Move each verified user into the list of confirmed users.

➋ while unconfirmed\_users:

➌ current\_user = unconfirmed\_users.pop()

print(f"Verifying user: {current\_user.title()}")

➍ confirmed\_users.append(current\_user)

# Display all confirmed users.

print("\nThe following users have been confirmed:")

for confirmed\_user in confirmed\_users:

print(confirmed\_user.title())

**This program will go through a list and remove all instances of cat until the list is clear of cat**

pets = ['dog', 'cat', 'dog', 'goldfish', 'cat', 'rabbit', 'cat']

print(pets)

while 'cat' in pets:

pets.remove('cat')

print(pets)

**This program will keep storing mountains in a list until told to stop repeating**

responses = {}

# Set a flag to indicate that polling is active.

polling\_active = True

while polling\_active:

# Prompt for the person's name and response.

➊ name = input("\nWhat is your name? ")

response = input("Which mountain would you like to climb someday? ")

# Store the response in the dictionary.

➋ responses[name] = response

# Find out if anyone else is going to take the poll.

➌ repeat = input("Would you like to let another person respond? (yes/ no) ")

if repeat == 'no':

polling\_active = False

# Polling is complete. Show the results.

print("\n--- Poll Results ---")

➍ for name, response in responses.items():

print(f"{name} would like to climb {response}.")

*Python –* Chapter 8

Summary

This chapter focuses on Functions

Outline

1. Defining a Function

**This program defines a greeting function and then uses said function**

def greet\_user():

➋ """Display a simple greeting."""

➌ print("Hello!")

➍ greet\_user()

**This program is similar, but will vary based on inputed information**

def greet\_user(username):

"""Display a simple greeting."""

print(f"Hello, {username.title()}!")

greet\_user('jesse')

1. Passing Arguments

**This program will use a function with outcomes based on where the information is place**

def describe\_pet(animal\_type, pet\_name):

"""Display information about a pet."""

print(f"\nI have a {animal\_type}.")

print(f"My {animal\_type}'s name is {pet\_name.title()}.")

➋ describe\_pet('hamster', 'harry')

1. Return Values

**This program will ask for two strings can return a single string involving both information**

➊ def get\_formatted\_name(first\_name, last\_name):

"""Return a full name, neatly formatted."""

➋ full\_name = f"{first\_name} {last\_name}"

➌ return full\_name.title()

➍ musician = get\_formatted\_name('jimi', 'hendrix')

print(musician)

**This program is similar, but has an optional middle name**

def get\_formatted\_name(first\_name, last\_name, middle\_name=''):

"""Return a full name, neatly formatted."""

➋ if middle\_name:

full\_name = f"{first\_name} {middle\_name} {last\_name}"

➌ else:

full\_name = f"{first\_name} {last\_name}"

return full\_name.title()

musician = get\_formatted\_name('jimi', 'hendrix')

print(musician)

➍ musician = get\_formatted\_name('john', 'hooker', 'lee')

print(musician)

**This program will create a dictionary in a function and return it**

def build\_person(first\_name, last\_name):

"""Return a dictionary of information about a person."""

➊ person = {'first': first\_name, 'last': last\_name}

➋ return person

musician = build\_person('jimi', 'hendrix')

➌ print(musician)

**This program will return a formatted name and keep doing so until the user inputs quit**

def get\_formatted\_name(first\_name, last\_name):

"""Return a full name, neatly formatted."""

full\_name = f"{first\_name} {last\_name}"

return full\_name.title()

while True:

print("\nPlease tell me your name:")

print("(enter 'q' at any time to quit)")

f\_name = input("First name: ")

if f\_name == 'q':

break

l\_name = input("Last name: ")

if l\_name == 'q':

break

formatted\_name = get\_formatted\_name(f\_name, l\_name)

print(f"\nHello, {formatted\_name}!")

1. Passing a List

**This program will print all members of a list**

def greet\_users(names):

"""Print a simple greeting to each user in the list."""

for name in names:

msg = f"Hello, {name.title()}!"

print(msg)

➊ usernames = ['hannah', 'ty', 'margot']

greet\_users(usernames)

**This function will add items from one list into another until a list is empty, then print each member individually**

# Start with some designs that need to be printed.

unprinted\_designs = ['phone case', 'robot pendant', 'dodecahedron']

completed\_models = []

# Simulate printing each design, until none are left.

# Move each design to completed\_models after printing.

while unprinted\_designs:

current\_design = unprinted\_designs.pop()

print(f"Printing model: {current\_design}")

completed\_models.append(current\_design)

# Display all completed models.

print("\nThe following models have been printed:")

for completed\_model in completed\_models:

    print(completed\_model)

1. Passing an Arbitrary Number of Arguments

**This program will accept any amount of arguments and create a list to represent a pizza**

def make\_pizza(\*toppings):

"""Print the list of toppings that have been requested."""

print(toppings)

make\_pizza('pepperoni')

make\_pizza('mushrooms', 'green peppers', 'extra cheese')

**This program is similar, but will also ask for a size of pizza**

def make\_pizza(size, \*toppings):

"""Summarize the pizza we are about to make."""

print(f"\nMaking a {size}-inch pizza with the following toppings:")

for topping in toppings:

print(f"- {topping}")

make\_pizza(16, 'pepperoni')

make\_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')

**This function will build a dictionary based on any amount of information passed in**

def build\_profile(first, last, \*\*user\_info):

"""Build a dictionary containing everything we know about a user."""

➊ user\_info['first\_name'] = first

user\_info['last\_name'] = last

return user\_info

user\_profile = build\_profile('albert', 'einstein',

location='princeton',

field='physics')

print(user\_profile)

1. Storing Your Functions in Modules

**This will store the function as a module**

module\_name.function\_name()

**This will import functions from a module**

from module\_name import function\_name

1. Styling Functions