Analysis of Search Algorithms in a Pacman World

Pablo Sainz Albanez
EGIA
Tecnologico de Monterrey
Zapopan, Jalisco 45201
Email: pablo.sainz@gmail.com

Jorge Xicotencatl del Valle Valenzuela
EGIA
Tecnologico de Monterrey
Zapopan, Jalisco 45201
Email: xico.delvalle@gmail.com

***Abstract*—Search algorithms are a medular part of the AI discipline. The way we choose and apply such search techniques to a specific problem and context will determine the effectiveness and the efficency of finding the solution.This paper illustrates an analysis being made to several serch algorithms, heuristics applied to a series of problems in the context of a pac-man game. This will lead us to develop our, and your, conclusions about the search techniques used and the impact on getting the solution to a specific problem.**

I. Introduction

As stated by Russel and Norvig[1], the solution to a problem comes in two main phases: One is to formulate the objective we want to acheive, and the other one is to formulate the problem itself, meaning which pieces of information are the relevant ones and which actions need to be taken to really solve the problem at hand.Speaking in general terms, the more clear we have the first formulation and, even more important, the way we formulate the problem, will determine the success of our search agent.

A. Taxonomy of the Pacman Environment

Given the context of this small project, the Pacman world, a portion of formulating the problem is already done. Meaining that, we will be working with an entire framework that abstracts and encapsulate a lot of functionality to interact and respresent a given pacman game scenario. Such functionality includes environment representation, display and basic operations and data structures for environment navigation.

As for the formulation of the objectives, these will vary from problem to problem that is being analyzed.

The Pacman environment posses a large collection of classes and utility methods to offer such functionality. For the purpose of these experiments, we will focus on two modules: The search agents and the search utilities. The search utilities file will contain the base generic search algorithms and search tree navigation. This algorithms will serve as the core functionality in order to get the solution for a given problem of a particular experiment. On the otherhand, a search agent is a defined entity that solves a specific search problem(equivalent to a specific Pacman game scenario). These agents will leverage on search algorithms and defined heuristics (some predefined and some others proposed) to extract the useful information from the environment and come up with the solution of a given problem.

B. Methodology

This small project consist on a series of small problems or challenges that need to be solved. On each problem, a specific functionality is required to be implemented on the modules described on the previous subsection and then, a series of experiments are proposed with a base expected result in order to evaluate the effectiveness and eficency of the solution developed.

During the next section we will elaborate on each one of these problems, what their purpose is, what the proposed implementation was, and which were the obtained results with its corresponding comparison to the base results.

In a second moment,after elaborating all the experiments and based on the results obtained,we will elaborate on the conclusions regarding the search algorithms and the heuristics being proposed.

II. Search Algorithm Experiments

A. Implementing Depth First Search

Problem statement: Implement the depth-first search (DFS) algorithm in the depthFirstSearch function in search.py. To make your algorithm complete, write the graph search version of DFS, which avoids expanding any already visited states (textbook section 3.5).
In this problem Pac-Man should find a path to a fixed dot located in the labyrinth. First step in order to solve this problem is to formulate the problem:

- States: The Pac-man is in specific position composed by a ”x,y” coordinates.
- Initial State: Any state can be assigned as initial state with the exception of wall coordinates.
- Succesor Fuction: This function should return the next coordinates.
- Test Goal: The Pac-man finds the dot located in the labyrinth.
- Total Cost: Number of movements from the initial state to the goal.

The DFS algorithm expands the deepest node of the tree. In order to develop an algorithm that work for this first problem there are many ways to implement it but all of them work with the same principle: usage of a data structure with a LIFO behavior. The expression ”many ways to implement it”

means that implementation can vary, examples for these are the following:

- Implement a PriorityQueue can be achived by using a decrementing counter by each new item inserted into the queue.
- Implement a Stack using the utilities provided into the "util" file.

The code implemented reflects both points by just changing the structure used (priorityQueue vs Stack) and removing/adding the priority parameter in the "push" call.
The following table reflects the results by using the DFS on this problem:

Scenario	Time spent	Total cost	Nodes expanded
TinyMaze	0.0	11	15
MediumMaze	0.0	131	146
BigMaze	0.1	211	390

TABLE I
DFS RESULTS

B. Implementing Breadth First Search

This specific problem was to add a generic breadth first search (BFS) functionality into the search library. As of speaking from the implementation in comparison with the deep first search, the only variant was a simplification on the criterion for search node insertion on the fringe, meaining that the search nodes are inserted in to the fringe as a standard queue (FIFO). The results obtained are shown in the table below:

Maze Size	Solution Cost	Time	Nodes Expanded	Score gained
Tiny	9	0.0 sec	15	502
Medium	69	0.1 sec	273	442
Big	211	0.1 sec	617	300

TABLE II
BREADTH FIRST SEARCH PERFORMANCE

C. Implementing Uniform Cost Search

Problem statement:Implement the uniform-cost graph search algorithm in the uniformCostSearch function in search.py. We encourage you to look through util.py for some data structures that may be useful in your implementation. You should now observe successful behavior in all three of the following layouts, where the agents below are all UCS agents that differ only in the cost function they use (the agents and cost functions are written for you):
python pacman.py -l mediumMaze -p SearchAgent -a fn=ucspython
pacman.py -l mediumDottedMaze -p StayEastSearchAgent-python
pacman.py -l mediumScaryMaze -p StayWestSearchAgent The BFS is potimal when all costs are the same because always expands the cost of the most superficial node. Instead of expanding the most superficial node the uniform cost search expands the node with the lowest cost. Implementing this algorithm in the same problem does not require many changes in the source code. The approach to follow is the same, however instead of using the Stack, the PriorityQueue is more useful since the cost can be used as parameter for each node is inserted into the Queue. The state definition provide us the cost of each state, this means that we have already the information (cost) needed for each expanded state. Notice when all the costs are identycal, the uniform cost search is the same as breadth-first-search.
The following table reflects the results by using the DFS in the different scenarios: The note from the excercise: *"You should get very low and very high path costs for the StayEast-SearchAgent and StayWestSearchAgent respectively, due to their exponential cost functions (see searchAgents.py for details)"* can be observed in the described table. The exponential increment between the StayEastSearch vs the StayWestSearch is explained by looking at the searchAgents.py file. The StayEastSearch penalizes being in positions on the West side of the board while the StayWestSearch penalizes being in positions on the East side of the board.

Scenario	Time spent	Total cost	Nodes expanded
MediumMaze	0.1	69	213
StayEastSearch	0.0	1	207
StayWestSearch	0.0	68719479866	98

TABLE III
UCS RESULTS

D. Implementing A* Search

This specific problem was to add a generic A* search (BFS) functionality into the search library.The A* search implementation contained the additional variation of receiving the heuristic function as an additional parameter, which in this case was a predefined Manhattan heuristic. Considering the A* approach on which heuristic and cumulative cost are also consider for node prioritization on the fringe, additional functionality was added on both the search node creation and structure to consider the cumulative search cost of a given point on the search tree.Considering this implementation, the results obtained are shown in the table below:

Maze Size	Solution Cost	Time	Nodes Expanded	Score gained
Tiny	9	0.0 sec	14	502
Medium	69	0.1 sec	224	442
Big	211	0.1 sec	549	300

TABLE IV
A* SEARCH PERFORMANCE

E. The Four Corners Problem with BFS

Problem statement: Implement the CornersProblem search problem in searchAgents.py. You will need to choose a state representation that encodes all the information necessary to detect whether all four corners have been reached. Formulating

the problem can be a good methodology to understand better adopt a good approach:

- States: The Pac-man is in specific position composed by a "x,y" coordinates and a tuple of 4 components containing the corners that have been found. This last sentence is really important because it made me realize that I was stating incorrectly the problem and helped me to find a good approach.
- Initial State: Any state can be assigned as initial state and all corners in "False" state (unless the Pac-Man's initial coordinates match with a corner coordinates).
- Succesor Fuction: This function should return the next coordinates.
- Test Goal: The Pac-man finds the four corners of the labyrinth.
- Total Cost: Number of movements from the initial state to the goal.

As mentioned in the "States" description an initial approach was used with incorrect results. In the initial representation of the state it was not considered the 4 corners as part of the state and inside the condition to determine if the Goal state has been reached it was only comparing if wether a state matched with each of the coordinates. This was causing an incorrect behavior because wether the 4 coordinates matched with any state (even when that state corresponded to different branches) it was moving only to one of the corners.
With this learning and after formulating correctly the problem it was decided to add a tuple with 4 corners in each state. This means that each state encapsulate the following components:the coordinates and a tuple with 4 elements of each corner reached.
The results of the implementation are shown in the following table:

Scenario	Time spent	Total cost	Nodes expanded
TinyCorners	0.3	29	1502
MediumCorners	–	–	–

TABLE V
CORNER PROBLEM RESULTS

F. The Heuristic for the Four Corners Problem

Problem statement: Implement a heuristic for the CornersProblem in cornersHeuristic. Grading: inadmissible heuristics will get no credit. 1 point for any admissible heuristic. 1 point for expanding fewer than 1600 nodes. 1 point for expanding fewer than 1200 nodes. Expand fewer than 800, and you're doing great! The first step is to decide admissible heuristic for this problem. One initial heuristic to try can be the manhattanDistance from each current state to the 4 corners and deciding to the closest corner to the current state.

G. The Eat-All-Dots Problem

Problem statement: Now we'll solve a hard search problem, eating all the Pac-Man food in as few steps as possible.

Heuristic	Time spent	Total cost	Nodes expanded
ManhattanDistance	2.8	107	2499
???	–	–	–

TABLE VI
CORNER PROBLEM WITH HEURISTICRESULTS

For this, we'll need a new search problem definition which formalizes the food-clearing problem: FoodSearchProblem in searchAgents.py (implemented for you). A solution is defined to be a path that collects all of the food in the Pac-Man world. For the present project, solutions do not take into account any ghosts or power pellets; solutions only depend on the placement of walls, regular food and Pac-Man.

Now for the purpose of this experiment we will need to fill the functionality in foodHeuristic in searchAgents.py with a consistent heuristic for the FoodSearchProblem.Part of the problem stated that the sample implementation using a Uniform Cost Search method found the optimal way within 13 seconds and after explanding up to 16,000 nodes in the search three. Considering this, the problem offered different scores depending on the benchmark of your heuristic against theirs based on the following criteria:

Fewer nodes than:	Score
15,000	1
12,000	2
9,000	3 (medium)
7,000	4 (hard)

TABLE VII
HEURISTIC SCORING CRITERIA

For testing this heuristic, two experiments were designed, both testing scenarios varies in on both maze size and food dots distribution:

- python pacman.py -l trickySearch -p AStarFoodSearchAgent
- python pacman.py -l mediumSearch -p AStarFoodSearchAgent

Considering the context of an informed search scheme. The perceptions and available data to be provided as an input to the heuristic were the following:

- Pacman possible next position in the cartesian plane
- The food grid on the maze, which could be also manipulated as a list of 2D cartesian coordinates.
- The wall grid on the maze, which could be also manipulated as a list of 2D cartesian coordinates.

For the purposes of this problem we proposed the following heuristic:

$$H(n) = p(n) + q(n)$$

$$p(n) = max\{d(n)\}$$

$$d(n) = |Fx1 - Px1| + |Fx2 - Px2|$$

$$q(n) = |F|^2$$

$$F = ((Fx1, Fx2), (Fx1, Fx2)...)$$

Briefly explaining the heuristic proposed above, the core parts of the calculation consist on:

- Calculating the maximum Manhattan distance to a given food dot(Fx1,Fx2) from the current position (Px1,Px2).
- Consider also the number of remaining food dots on the environment given the movement applied. Penalizing those nodes that didnt takes us closer to the ultimate goal of the game which was to consume all the food dots in the maze.

The formulation of this heuristic is based on the simple idea of giving preference to those nodes which besides taking us closer to the food dots, will cause that we reduce the number of remaining food on the environment.

The results obtained are shown in the table below:

Experiment	Time spent	Total cost	Nodes expanded
trickySearch	0.1 sec	63	172
mediumSearch	0.4 sec	205	373

TABLE VIII
ALL FOOD DOTS HEURISTIC PERFORMANCE

H. The Suboptimal Search Problem

Problem statement:Sometimes, even with A* and a good heuristic, finding the optimal path through all the dots is hard. In these cases, we'd still like to find a reasonably good path, quickly. In this section, we'll write an agent that always eats the closest dot. ClosestDotSearchAgent is implemented for you in searchAgents.py, but it's missing a key function that finds a path to the closest dot.

For the purpose of this experiment we need to Implement the function findPathToClosestDot in searchAgents.py. The problem states that the sample implementation finds a solution within a second and having a total cost of 350. The proposed experiment is the following one:

- python pacman.py -l bigSearch -p ClosestDotSearchAgent -z .5

To address this specific problem we leveraged from the existing functionality already implemented for the A* searches. But before even start searching for a solution there were two main steps to be completed:

- Find an quick way to choose the local goal to start the search
- Assign the appropiate heuristic for the A* search

For the first part,the approach followed to locally select the local goal to look for before launching the search for the solution was to establish a search radius. The value of this radius always started at one. If no food dot was found then this radius would be incremented by a factor of 2 and look again for food dots wihin this search radius. The goal with the minimum euclidean distance to the current Pacman position will be the local goal for that search iteration.

As for the second part the heuristic of choice was a simple Manhattan function to establish the priority of the nodes within the fringe during the search process.

After finishing the impplementation described above, the result we found was that actually the solution found has a total cost of 466 which is less optimal that the proposed base implementation, but it has the benefit of being able to find the solution in 0.20 seconds, which is a lot quicker than the example implementation.

resultados obtenidos

III. CONCLUSIONS

Agregale algo de rollo si quieres

IV. OBSERVATIONS ABOUT THIS ARTICLE

Que piensas del paper en si

REFERENCES

[1] H. Kopka and P. W. Daly, *A Guide to LaTeX*, 3rd ed. Harlow, England: Addison-Wesley, 1999.