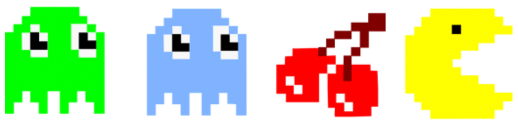
Project 1: Search

Pacman

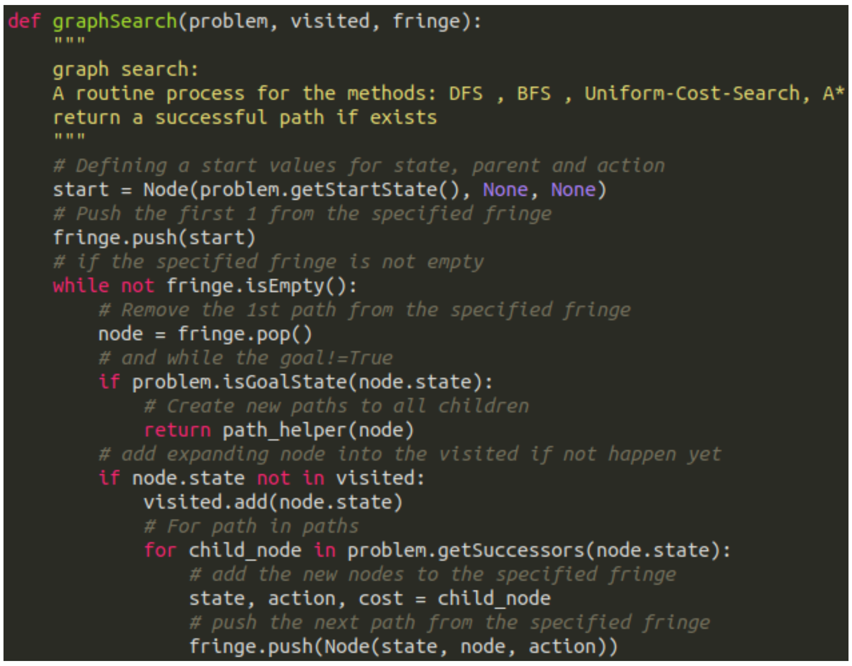


Autonomous Systems

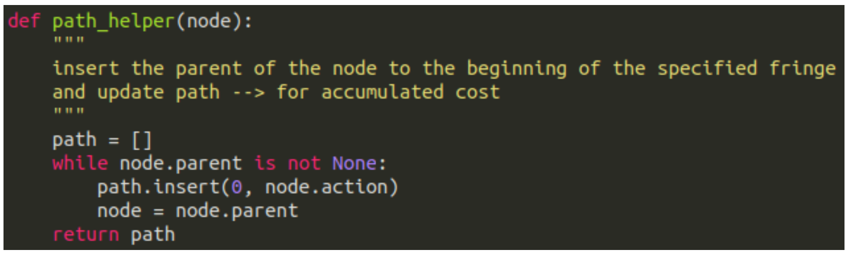
February 2020

Hamit Kavas
Elif Hangül
Jonatan Koren

- To deal with DFS, BFS, and Uniform-Cost-Search, A* (with additional function), the **graphSearch function** below Perform a similar process except for the fringe and its corresponding visited order, and the type of the desired problem - the search. When creating new paths to expanded nodes the function calls the **path_helper function** that inserts the action to the parent of the node to the beginning of the specified fringe and updates the path for accumulated cost and or heuristic

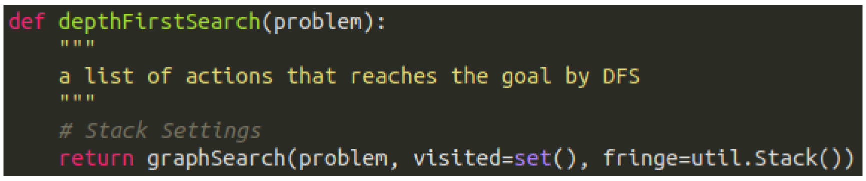


```python
def graphSearch(problem, visited, fringe):
    """
    graph search:
    A routine process for the methods: DFS , BFS , Uniform-Cost-Search, A*
    return a successful path if exists
    """
    # Defining a start values for state, parent and action
    start = Node(problem.getStartState(), None, None)
    # Push the first 1 from the specified fringe
    fringe.push(start)
    # if the specified fringe is not empty
    while not fringe.isEmpty():
        # Remove the 1st path from the specified fringe
        node = fringe.pop()
        # and while the goal!=True
        if problem.isGoalState(node.state):
            # Create new paths to all children
            return path_helper(node)
        # add expanding node into the visited if not happen yet
        if node.state not in visited:
            visited.add(node.state)
            # For path in paths
            for child_node in problem.getSuccessors(node.state):
                # add the new nodes to the specified fringe
                state, action, cost = child_node
                # push the next path from the specified fringe
                fringe.push(Node(state, node, action))
```



```python
def path_helper(node):
    """
    insert the parent of the node to the beginning of the specified fringe
    and update path --> for accumulated cost
    """
    path = []
    while node.parent is not None:
        path.insert(0, node.action)
        node = node.parent
    return path
```

1) Depth-First Search



```python
def depthFirstSearch(problem):
    """
    a list of actions that reaches the goal by DFS
    """
    # Stack Settings
    return graphSearch(problem, visited=set(), fringe=util.Stack())
```

Implementation:

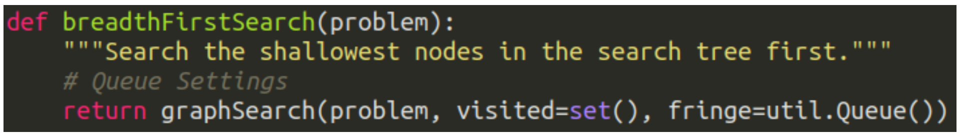
- Data Structure - DFS uses *util.Stack* as a fringe.

Observations:

DFS				
Maze Layout	**Search Path**	**Cost**	**Nodes expanded**	**Time Spent (s)**
Tiny Maze	Search Agent	10	15	<0.1
Medium Maze	Search Agent	130	146	<0.1
Big Maze	Search Agent	210	390	0.1
Open Maze	Search Agent	298	576	0.1

- The order of the exploration was the way that we expected as the search goes down along the depth before checking other paths on the same depth.
- Pacman explorers all the squares, it should rather find a path without dead ends.
- DFS is not optimal due to the fact that it expands all the children of a node first and thus it does not go through a path with lesser cost. Tiny Maze also follows the same, but it has been affected less than Medium and Open Maze.
- DFS could be optimal if the search tree is finite, all action costs are the same and all the solutions have the same length.

2) Breadth-First Search



```
def breadthFirstSearch(problem):
    """Search the shallowest nodes in the search tree first."""
    # Queue Settings
    return graphSearch(problem, visited=set(), fringe=util.Queue())
```

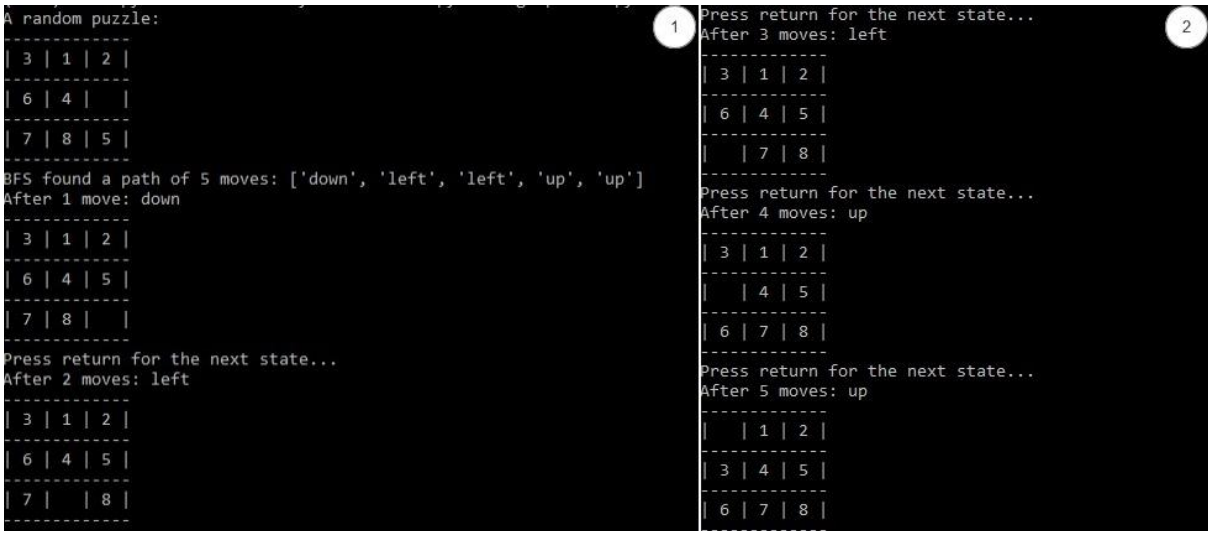
Implementation:

- Data Structure - BFS uses util Queue as a fringe.

Observations:

BFS				
Maze Layout	**Search Path**	**Cost**	**Nodes expanded**	**Time Spent (s)**
Tiny Maze	Search Agent	8	15	<0.1
Medium Maze	Search Agent	68	269	0.1
Big Maze	Search Agent	210	620	0.1
Open Maze	Search Agent	54	682	0.1

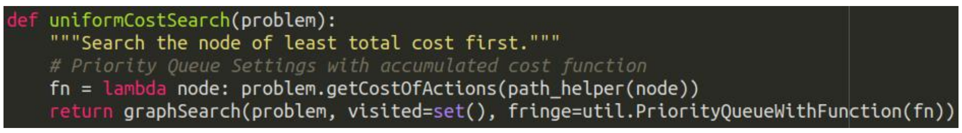
● Breadth First Search found path costs for Tiny, Medium and Open Maze less than Depth First Search.
● Since Breadth First Search checks all the nodes present at a particular depth and then increasingly goes to the next depth, the solution found is optimal.
● The picture below shows the execution of the Eight Puzzle problem:



```
A random puzzle:
-------------
| 3 | 1 | 2 |
-------------
| 6 | 4 |   |
-------------
| 7 | 8 | 5 |
-------------
BFS found a path of 5 moves: ['down', 'left', 'left', 'up', 'up']
After 1 move: down
-------------
| 3 | 1 | 2 |
-------------
| 6 | 4 | 5 |
-------------
| 7 | 8 |   |
-------------
Press return for the next state...
After 2 moves: left
-------------
| 3 | 1 | 2 |
-------------
| 6 | 4 | 5 |
-------------
| 7 |   | 8 |
-------------
```

```
Press return for the next state...
After 3 moves: left
-------------
| 3 | 1 | 2 |
-------------
| 6 | 4 | 5 |
-------------
|   | 7 | 8 |
-------------
Press return for the next state...
After 4 moves: up
-------------
| 3 | 1 | 2 |
-------------
|   | 4 | 5 |
-------------
| 6 | 7 | 8 |
-------------
Press return for the next state...
After 5 moves: up
-------------
|   | 1 | 2 |
-------------
| 3 | 4 | 5 |
-------------
| 6 | 7 | 8 |
-------------
```

3) Uniform-Cost Search



```python
def uniformCostSearch(problem):
    """Search the node of least total cost first."""
    # Priority Queue Settings with accumulated cost function
    fn = lambda node: problem.getCostOfActions(path_helper(node))
    return graphSearch(problem, visited=set(), fringe=util.PriorityQueueWithFunction(fn))
```

Implementation:

● Data Structure - UCS uses util priority Queue as a fringe.

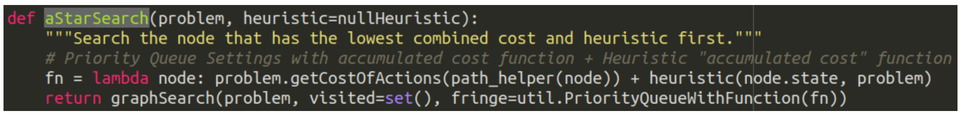
Observations:

UCS				
Maze Layout	**Search Path**	**Cost**	**Nodes expanded**	**Time Spent (s)**
Tiny Maze	Search Agent	8	15	<0.1
Medium Maze	Search Agent	68	269	0.1
Big Maze	Search Agent	210	620	0.1
Open Maze	Search Agent	54	682	0.1
Medium Dotted Maze	Stay East Search Agent	1	186	0.1
Scary Dotted Maze	Stay West Search Agent	68719479864	108	0.1

● Path cost obtained for UCS is optimal .

● Optimal path with a very little cost is observed for Medium Dotted Maze with Stay East Search Agent. Furthermore, an optimal path with a very large cost is observed for Medium Scary Maze with Stay West Search Agent. This is because of the restriction on the movements of Pacman, the cost increases exponentially.

4) A* search



```python
def aStarSearch(problem, heuristic=nullHeuristic):
    """Search the node that has the lowest combined cost and heuristic first."""
    # Priority Queue Settings with accumulated cost function + Heuristic "accumulated cost" function
    fn = lambda node: problem.getCostOfActions(path_helper(node)) + heuristic(node.state, problem)
    return graphSearch(problem, visited=set(), fringe=util.PriorityQueueWithFunction(fn))
```

Implementation:

- Data Structure - A* Search uses util priority Queue as a fringe.

Observations:

● If no heuristic is passed to A* search in the command line, it takes the null heuristic and performs in a similar manner as UCS. This is because the total cost estimated becomes equal to the actual path cost to reach the current position since all the paths have uniform cost of 1.

A* with manhattan				
Maze Layout	**Search Path**	**Cost**	**Nodes expanded**	**Time Spent (s)**
Tiny Maze	Search Agent	8	14	<0.1
Medium Maze	Search Agent	68	221	0.2
Big Maze	Search Agent	210	549	0.8
Open Maze	Search Agent	54	535	0.4

● It could be clearly seen that with a heuristic, the search is able to expand less number of nodes and saving space. Thus, performs better than both Uniform Cost Search and Breadth-First Search.

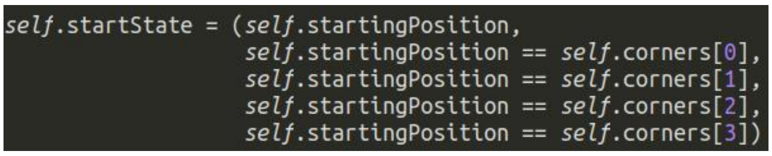
Comparison of strategies on Open Maze:

● With Open Maze all BFS, UCS, A* finds the optimal path to the goal of cost 54. While they differ in the number of nodes expanded, A* expands 535 nodes, least among the three.
● While DFS also reaches the goal state with a path cost of 298 expanding about 576 nodes. This makes it quite clear that DFS does not find an optimal solution. While BFS, UCS, A* get the optimal solution with the number of nodes expanded least in A*.

5) Finding All The Corners

Implementation:

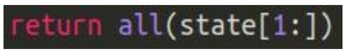
● Defining state for the problem: The state consists of the current position and a list of corners visited.
● Defining the start state:



```
self.startState = (self.startingPosition,
                   self.startingPosition == self.corners[0],
                   self.startingPosition == self.corners[1],
                   self.startingPosition == self.corners[2],
                   self.startingPosition == self.corners[3])
```

➢ The position of the agent is checked to see if it is a corner. Based on these observations state returns the current position and a list of corners visited.

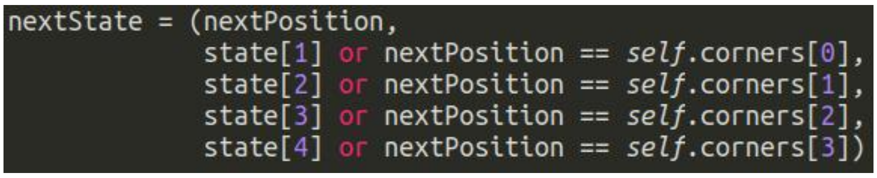
● Goal state of the problem:



```
return all(state[1:])
```

When the corner list contains true for all corners, a goal state is achieved.

● Getting successors for a node:



```
nextState = (nextPosition,
             state[1] or nextPosition == self.corners[0],
             state[2] or nextPosition == self.corners[1],
             state[3] or nextPosition == self.corners[2],
             state[4] or nextPosition == self.corners[3])
```

If the next position is not a wall, get the next position. Check it is a corner. If it is a corner that is not visited or is a corner already visited return true for that corner, update the next state with a new position and new corner list.

Observations:

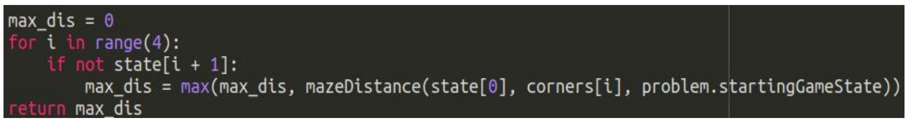
Finding All The Corners				
Maze Layout	**Search Path**	**Cost**	**Nodes expanded**	**Time Spent (s)**
Tiny Corners	BFS	28	252	<0.1
Medium Corners	BFS	106	1,966	0.2
Tiny Corners	A* (Null Heuristic)	28	252	0.1
Medium Corners	A* (Null Heuristic)	106	1,966	1.7

● BFS and A*(without any heuristics) perform the same in path cost and number of nodes expanded. This is because to reach every adjacent position, the cost is 1. Thus, these uniform costs make Uniform Cost Search reduced to Breadth-First Search.
● However, sending a null heuristic to the algorithm will not affect the result of A*. Hence, we have observed the same path costs and expansion of nodes.

6) Corners Problem: Heuristic

Implementation:

● Defining a heuristic:



```python
max_dis = 0
for i in range(4):
    if not state[i + 1]:
        max_dis = max(max_dis, mazeDistance(state[0], corners[i], problem.startingGameState))
return max_dis
```

In our implementation, we defined the heuristic as the distance from the current position to the farthest corner.

● **Admissibility of heuristic:**
The chosen heuristic is admissible as the maze distance gives the distance between two positions and since the maximum distance is defined as the distance between the current position and the farthest corner, the actual path could not get shorter than this distance.
Thus, the distance would be definitely a lower bound to the actual distance.

● **Consistency of heuristic:**
Consistent as the maze distances between the four corners remain the same after the addition of heuristics.

Observations:

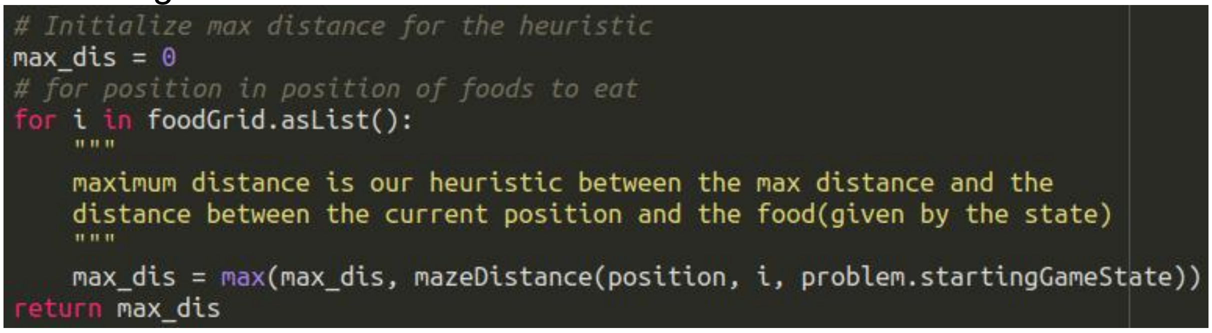
Corners Problem: Heuristic				
Maze Layout	**Search Path**	**Cost**	**Nodes expanded**	**Time Spent (s)**
Tiny Corners	A* (Corner Heuristic)	28	128	0.7
Medium Corners	A* (Corner Heuristic)	106	801	34
Tiny Corners	Uniform Cost Search	28	252	0.1
Medium Corners	Uniform Cost Search	106	1,966	1.7

● We observe that the optimal path for A* with corner heuristic remains the same as Uniform Cost Search, in other words, the A* with a null heuristic. This proves that our algorithm is not inconsistent.
● The number of nodes expanded with heuristic decreases almost to half number of nodes expanded without heuristic. Using heuristics also significantly increases the amount of time spent in seconds.
● Thus, defining admissible and consistent corner heuristic we are able to increase the performance of Pacman to reach the goal state.

7) Eating All The Dots

Implementation:

● Defining a heuristic:



```
# Initialize max distance for the heuristic
max_dis = 0
# for position in position of foods to eat
for i in foodGrid.asList():
    """
    maximum distance is our heuristic between the max distance and the
    distance between the current position and the food(given by the state)
    """
    max_dis = max(max_dis, mazeDistance(position, i, problem.startingGameState))
return max_dis
```

The heuristic considers the farthest food from the current position based on maze distance.

● Admissibility of heuristic:
The heuristic is admissible because we cannot find a lower bound for the shortest path from the agent to all the food-to-eat than the distance between the agent and the farthest food position.

Observations:

Eating All The Dots				
Maze Layout	**Search Path**	**Cost**	**Nodes expanded**	**Time Spent (s)**
Tricky Search	A* Farthest food (Maze Dist.)	60	4,137	150.3
Tricky Search	A* Farthest food (Manhattan Dist.)	60	9,551	14.5
Tiny Search	A* (Manhattan Dist.)	27	2,468	1.8
Tiny Search	A* (Maze Dist.)	27	2,372	26.8
More Comparison - Eating All The Dots				
Tricky Search	Uniform Cost Search	60	16,688	16.4
Tiny Search	Uniform Cost Search	27	5,057	2.7
Tricky Search	Breadth First Search	60	16,688	8.3
Tiny Search	Breadth First Search	27	5,057	1.3
Tricky Search	Depth First Search	216	361	0.2
Tiny Search	Depth First Search	41	59	<0.1

● We have used two different metrics(maze distance and manhattan distance) to demonstrate heuristics for A* algorithm. We have also tried to create heuristics with minimum distance, but we got the same results as uniform cost search, which accepts cost as 1. However, initiating algorithm with null heuristics has not changed the result, since minimum distance will be dominated by 0.
● In tricky search, we have observed that heuristic that are using manhattan distances finds solution pretty quickly. However, if we look into the number of nodes that are expanded, we can see that using manhattan distance for heuristic causes approximately two times of more space than heuristic using maze distance.
● The observation shows that more complex problems such as tricky search could be solved with maze distance with an optimal path by spending more time.
● A* with or without heuristic, BFS and UCS have the same cost for tiny search and for tricky search. Results are very close in tiny search excluding time.
●UCS and BFS in tricky search expands 16,688 nodes comparing to 4,137 for A*(with maze distances) or 9,551 for A*(with manhattan distances). We see that UCS and BFS behaved in the same way with Null Heuristics implemented A*.

8) Finding All the Corners

Implementation:

● Finding the path to Closest Food: Calling BFS to find the path to the closest food from current position.
● Defining goal state for any food search problem: If the current position has food, it is a goal state.

Observations:

Suboptimal Search			
Maze Layout	**Search Path**	**Cost**	**Score**
Big Search	Closest Dot (Maze Dis.)	350	2360
Tiny Search	Closest Dot (Maze Dist.)	31	569

● Since sometimes it is hard to find the optimal solution or it costs a lot, we may want to find a reasonably good path. The Closest Dot Search Agent may or may not find the shortest path through the maze. This is because at a time there might be two dots presented at the same distance. After a dot is chosen, the second dot will not be the closest dot. Due to the fact that it is not calculating all the positions of the other dots, it is suboptimal. A reasonably good path but not optimal.

● We can see this is not an optimal search clearly in the Tiny Search problem. The optimal search for Tiny Search costs 27 as we saw in question 7, but here we see it costs 31. Thus, the path cost returned here is not the optimal path cost; it is suboptimal.