Chapter 2

Problem-solving agents

2.1 Pac-Man project

The Pac-Man projects were developed for UC Berkeley's introductory artificial intelligence course, CS 188, and were released to other universities for education use `http://ai.berkeley.edu/project_overview.html`.

Download the code from `https://s3-us-west-2.amazonaws.com/cs188websitecontent/projects/release/search/v1/001/search.zip` and extract it into your own folder. You can open the folder from PyCharm or use it from command line. Start it with *python pacman.py*

Open from PyCharm Start *PyCharm*, create a new project, copy the folder `search` inside the folder of your project, and set `python 2.7` as project interpreter. Since you will run pacman with more options, you could create a run configuration for each combination (figure 2.1).

```
 $ pycharm.sh
File >> Open ... <choose search folder from your own folder>
File >> Settings >> Project:search >> Project Interpreter >> choose
   python 2.7

  Run pacman.py
Right click on pacman.py >> Run

  Add new Run configurations:
Run >> Edit Configurations >> + >> Choose Python >> Edit the
   parameters: Script, Script parameters, and Working directory
```

Figure 2.1: Run configuration for: python pacman.py -l tinyMaze -p SearchAgent -a fn=tinyMazeSearch

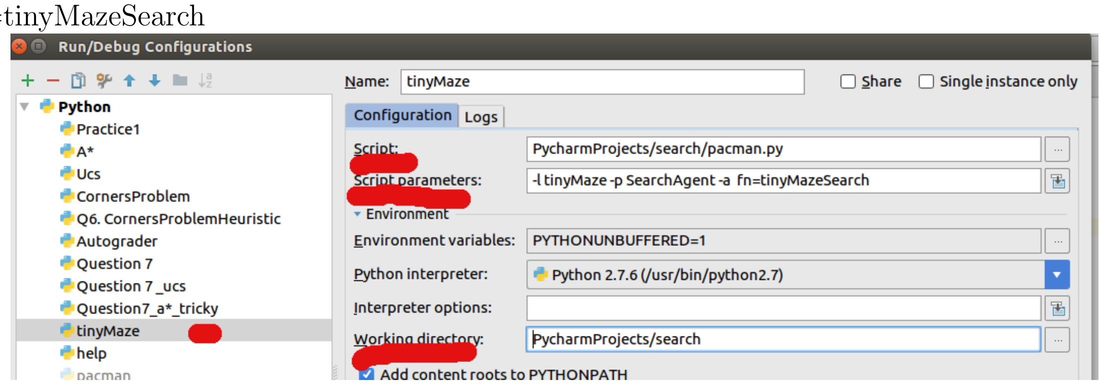


Figure 2.2: Methods of a *Search problem*

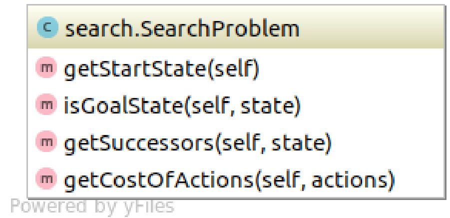
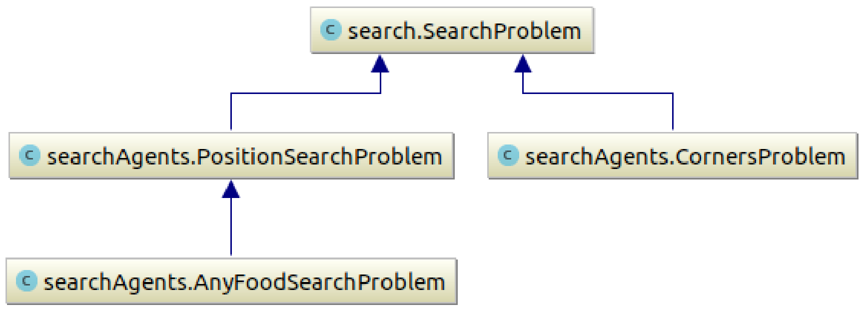


Figure 2.3: Types of *Search Problems* defined in `SearchAgens.py`



The content of *search* folder. In the extracted folder *search* there are more files from which you will need ony the following:

- Files to be changed
  - `search.py` - description of an abstract class *SearchProblem* (2.2) - you will not modify it. More importantly, the search strategies that you will implement will be here.
  - `searchAgents.py` - the search-based agents (2.4), together with the already described or ToBe described *Search problems*. The search problems are classes derived from the class *search.SearchProblem* as you can see in figure 2.3.

    Read the comments from the begining of the file: they explain how to use the options for setting the SearchAgent and **where to write your code**.

Figure 2.4: Types of Search Agents from `SearchAgens.py`

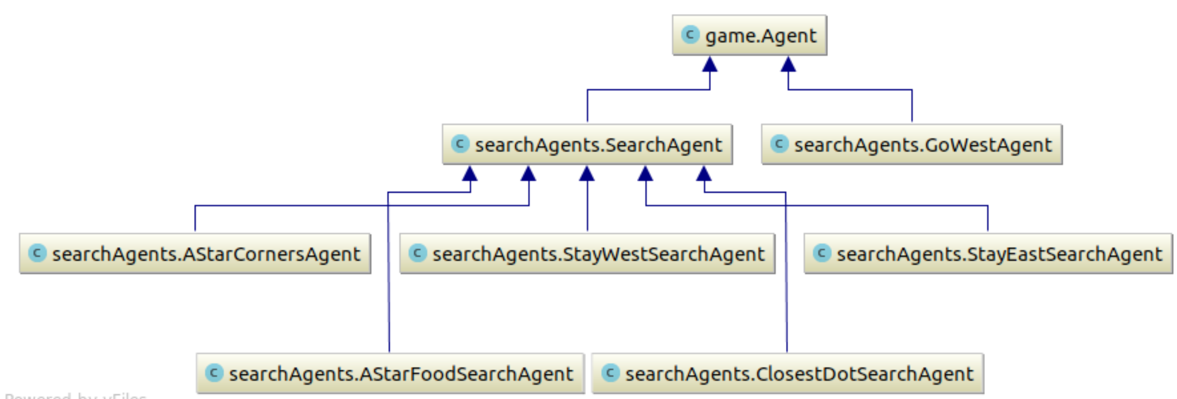
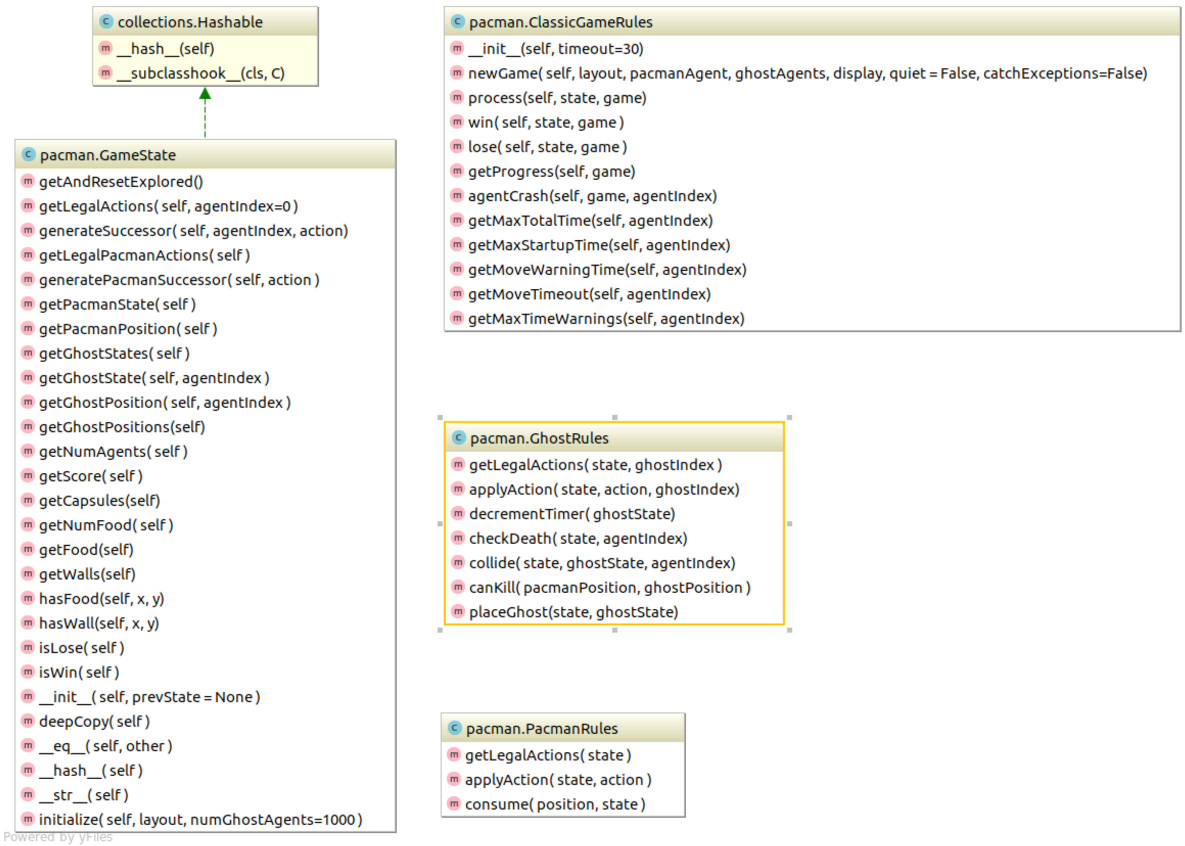


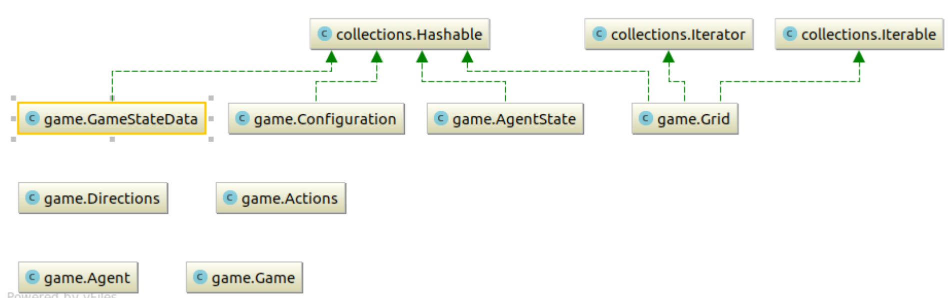
Figure 2.5: Classes and methods in `Pacman.py`. Don't change them



- Files which include worth reading parts
  - `pacman.py` - the main file for running Pacman games (figure 2.5). Read the description of GameState type which specifies the full game state; it is highly recommended that you use accessor methods for accessing the data about the state: *getLegaActions(), getPacmanState(), getPacmanPosition(), getCapsules(), hasFood()*.
  - `game.py` - the logic behind how the Pacman world works (figure 2.6). Important types: *AgentState*, *Agent*, *Direction*, *Grid*.
  - `util.py` - data structures which are recommended to be used when implementing the search algorithms

Pay special attention to the comments inside the files. In order to better understand

Figure 2.6: Classes and methods in `Game.py`. Don't change them



the code, you can create Class Diagrams in PyCharm (Right click on a file >> Diagrams).

You will most likely change only the files `search.py` and `searchAgents.py`.

2.1.1 Run Pacman

- Play a game of Pacman by running

```
python pacman.py
```

  Give the comman in command line or create a new Run COnfiguration in PyCharm.

- See the available options for pacman

```
python pacman.py -h
```

- Analyze and run the GoWestAgent from *searchAgents.py*

```
python pacman.py -l testMaze -p GoWestAgent
python pacman.py -l tinyMaze -p GoWestAgent
```

2.1.2 Preparatory exercises

Find a fixed food dot Pac-Man needs to find a certain food dot. The possible actions are North, South, East, West. The initial state and the goal state depend on the selected layout. This is formulated as a search problem and in this lab you will implement search strategies which are able to build a solution. But before that, you can do some exercises which help you to get familiar with Pac-Man framework.

1. Open `search.py` and identify *tinyMazeSearch* function. Run the command (see figure 2.1)

```
python pacman.py -l tinyMaze -p SearchAgent -a fn=tinyMazeSearch
```

   and observe what happens. If you want to slow down the movement of Pac-Man, use the option frametime *–frameTime=1*.

   - Read the output of your previous command: how many nodes were expaned? Which is the total cost of the found solution?
   - Why the agent finds the dot? Change the maze (with another one from *layouts* folder) and see what happens.

2. Go to *depthFirstSearch* function from `search.py`. Comment $util.raiseNotDefined()$ and similar to *TinyMazeSearch*, add to *depthFirstSearch* function:

```
from game import Directions
s = Directions.SOUTH
w = Directions.WEST
return [w, w]
```

   Run again

```
 python pacman.py -l smallMaze -p SearchAgent
```

   Observation: each search function must return a list of legal actions.

3. Go to *depthFirstSearch* function from `search.py` and uncomment the following:

```
print "Start:", problem.getStartState()
print "Is␣the␣start␣a␣goal?", problem.isGoalState(problem.
   getStartState())
print "Start's␣successors:", problem.getSuccessors(problem.
   getStartState())
```

Run again

```
 python pacman.py -l smallMaze -p SearchAgent
```

Analyze the result: *problem.getSuccessors(problem.getStartState)* returns a list of three tuples, one for each legal action:

```
Start: (11, 6)
Is the start a goal? False
Start's successors: [((11, 7), 'North', 1), ((12, 6), 'East', 1),
    ((10, 6), 'West', 1)]
```

4. Go to *depthFirstSearch* function from `search.py`. Get the succesors of the initial state and print the state, the action and the cost for each successor. Run again with smallMaze.

   Possible solution:

```
print "Initial␣state␣is", problem.getStartState()
for succ in problem.getSuccessors(problem.getStartState()):
    (state, action, cost) = succ
     print "Next␣state␣could␣be", state, "␣with␣action␣", action,
         "and␣cost␣", cost
```

5. Go to *depthFirstSearch* function from `search.py`. Return a sequence of two legal actions from the initial state.

   Possible asnwer:

```
(next, next_action, _) = problem.getSuccessors(problem.
   getStartState())[0]
(next_next, next_next_action, _) = problem.getSuccessors(next)[0]
print "A␣possible␣solution␣could␣start␣with␣actions␣",
   next_action, next_next_action
return [next_action, next_next_action]
#util.raiseNotDefined()
```

6. Go to *depthFirstSearch* function from `search.py`. Create a new data-structure with two components: name and cost. Create two instances of the new data structure and add them to a *Stack* described in `util.py`. Pop an element from the stack and print it.

   Possible answer with a new class

```
node1 = CustomNode("first", 3) # creates a new object of class
   CustomNode
node2 = CustomNode("second", 10)
print "Create␣a␣stack"
my_stack = util.Stack() # creates a new object of the class Stack
    defined in \texttt{util.py}
```

```
print "Push␣the␣new␣node␣into␣the␣stack"
my_stack.push(node1)
my_stack.push(node2)
print "Pop␣an␣element␣from␣the␣stack"
extracted = my_stack.pop() # call a method of the object
print "Extracted␣node␣is␣",extracted.getName(),"␣" extracted.
   getCost()
util.raiseNotDefined()
```

with the class defined also in `search.py`

```
class CustomNode:

    def __init__(self, name, cost):
        self.name = name # the attribute name of the class
            CustomeNode
        self.cost = cost # the attribute cost of the class
            CustomeNode

    def getName(self):
        return self.name

    def getCost(self):
        return self.cost
```

2.2 Reading exercises

2.2.1 Search problems

Read from AIMA what are and how to formalize *Search problems* in sections 3.1.1 and 3.1.2.

A search problem can be defined formally by:

- initial state
- possible actions
- transition model: Result(s,a). A **successor** state is a state reachable from a given state by a single actions
- goal test
- path cost

All the search problems from Pac-Man project are described in these terms (see figure 2.2). The solution to a search problem is a sequence of actions which if executed from the initial state, reaches a goal state (where the goal test is true).

2.2.2 Searching for solutions - Tree search & graph search

Read from AIMA about *Tree search* and *Graph search* as general methods for searching for solution of *Search problems* in section 3.3.

The general algorithm is described in the following pseudocode:

```
function TREE-SEARCH(problem) returns a solution, or failure
initialize the frontier using the initial state of problem

loop do
   if the frontier is empty then return failure
   choose a leaf node and remove it from the frontier
   if the node contains a goal state
       then return the corresponding solution
   expand the chosen node, adding the resulting nodes to the frontier
```

```
function GRAPH-SEARCH(problem) returns a solution, or failure
 initialize the frontier using the initial state of problem
 initialize the explored set to be empty

loop do
  if the frontier is empty then return failure
  choose a leaf node and remove it from the frontier
  if the node contains a goal state
    then return the corresponding solution
  add the node to the explored set
  expand the chosen node, adding the resulting nodes to the frontier
     only if not in the frontier or explored set
```

Node for search algorithms

It is recommended to have the following structure for nodes in tree/graph search algorithm:

- state: the state in the state space to which the node corresponds;
- parent: the node in the search tree that generated this node;
- action: the action that was applied to the parent to generate the node;
- path-cost: the cost $g(n)$ of the path from the initial state to the node

2.2.3 Uninformed search strategies

Read from AIMA about uninformed search strategies: Depth first search (section 3.4.3), breadth-first search (section 3.4.1) and uniform cost search (section 3.4.2).

- **Depth-first search** It is a tree/graph-search with the frontier as LIFO (stack).
- **Breadth-first search** It is a tree/graph-search with the frontier as FIFO (queue). Different to general *Graph-search algorithm*, the goal test must be applied to each node before inserting the element in the frontier rather then when it is extracted from the frontier and selected for expantion.

  Observation: breadth-first search always has the shallowest path to every node on the frontier.
- **Uniform-cost** Uniform cost search: the frontier is a priority queue. It expands the node with the lowest path cost $g(n)$.

2.3 Implementing exercises

In order to obtain maximum score for the activity of this lab, you need to obtain 9 points for the first three questions available in autograder. They ask you to implement different search strategies. Implement them as graph-searches with different types of data-structures for the frontier. In file `util.py` there are implemented Stack, Queue and PriorityQueue data.

It is recommended to write your code as general as possible, therefore you should use the methods from the *SearchProblem* class:

- *getStartState*
- *isGoalState*
- *getSuccessors*

In this way, your code will apply on any problem formalized as Search Problem.

Observation: If you implement DFS as a graph search, there will be minor differences between the three strategies DFS, BFS and UCS.

1. **Question 1** In `search.py`, implement **Depth-first search** (DFS) algorithm in function *depthFirstSearch*. DFS graph search is graph-search with the frontier as a LIFO queue (Stack).

   - test your solution on more layouts:

   ```
   python pacman.py -l tinyMaze -p SearchAgent
   python pacman.py -l mediumMaze -p SearchAgent
   python pacman.py -l bigMaze -z .5 -p SearchAgent
   ```

   - Are the solutions found by your DFS optimal? Explain your answer
   - Run autograder $*python autograder.py* and check the points for Question 1.
     For more details, go to project page `http://ai.berkeley.edu/search.html`.

2. **Question 2** In `search.py`, implement **Breadth-first search** algorithm in function $breadthFirstSearch$. Similar to DFS, test your code on mediumMaze and bigMaze by using the option $-a\ fn = bfs$

   ```
   python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs
   ```

   - Is the found solution optimal? Explain your answer.
   - Run autograder $*python autograder.py* and check the points for Question 2.

3. **Question 3**: Uniform-cost graph search

   - In `search.py`, implement uniform-cost graph search algorithm in $uniformCostSearch$ function. Test it with $mediumMaze$ and $bigMaze$ and compare the results to the ones obtained with DFS. Are the solutions different? Is the number of extended(explored) states smaller? Explain your answer.

   ```
   python pacman.py -l mediumMaze -p SearchAgent -a fn=ucs
   ```

   - Consider that some positions are more desirable than others. This can be modeled by a cost function which sets different values for the actions of stepping into positions. Identify in `searchAgents.py` the description of agents *StayEastSeachAgent* and $StayWestSearchAgent$ and analyze the cost function. Why the cost $.5 * *x$ for stepping into (x, y) is associated to $StayWestAgent$?

- Run the agents *StayEastSeachAgent* and *StayWestSearchAgent* on *mediumDottedMaze* and *mediumScaryMaze* with uniform cost search.

```
python pacman.py -l mediumDottedMaze -p StayEastSearchAgent
python pacman.py -l mediumScaryMaze -p StayWestSearchAgent
```

  For more details, go to project page `http://ai.berkeley.edu/search.html`.

- Run autograder $*python autograder.py* and check the points for Question 3.