Chapter 3

Informed Search

What can an agent do when no single action will achieve its goal? SEARCH. Although BFS, DFS and UCS are able to find solutions, they do not do it efficiently. They are called **uninformed** algorithms since they use only problem definition and no other information. The search can be reduced in many situations with some guidance on where to look the solutions. **Informed algorithms** use additional information about the problem in order to reduce the search.

Learning objectives for this week are:

1. To implement A^* algorithm
2. To compare search strategies: DFS, BFS, UCS, A*
3. To formulate a Search Problem
4. To define admisible and consistent heuristic for A^*

Exercise 3.1 *Create an agent which searches a solution randomly: it picks one legal action at each step. Write the search function in* `search.py` *similar to* tinyMazeSearch *function. The function must return a list of actions from the initial state to goal state which are randomly selected from the legal actions.*

Solution 3.1 *At each step, ask for the successors of the current state and choose one randomly.*

```
def randomSearch(problem):
    current = problem.getStartState()
    solution =[]
    while (not (problem.isGoalState(current))):
        succ = problem.getSuccessors(current)
        no_of_successors = len(succ)
        random_succ_index = int(random.random()*no_of_successors)
        next = succ[random_succ_index]
        current = next[0]
        solution.append(next[1])
    print "The␣solution␣is␣", solution
    return solution
```

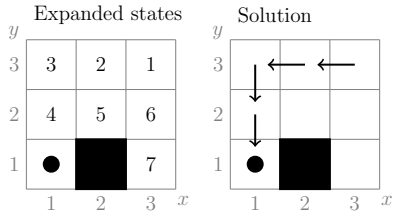
At the end of `search.py` *file add also*

```
rs = randomSearch
```

and run with option $-a\ fn = rs$

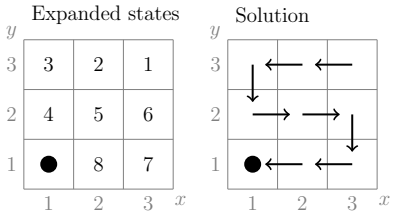
```
python pacman.py -l tinyMaze -p SearchAgent -a fn=rs
```

Exercise 3.2 *Compare the behavior of DFS on a simple layout of 3x3 with and without a wall in position (2,1). think about ways to reduce the number of expanded states and to improve the quality of the solution.*



Nodes added to the frontier	Extracted node
	(3, 3)
(3, 3)→(3, 2)(South) (3, 3)→(2, 3)(West)	
	(2, 3)
(2, 3)→(2, 2)(South) (2, 3)→(1, 3)(West)	
	(1, 3)
(1, 3)→(1, 2)(South)	
	(1, 2)
(1, 2)→(1, 1)(South) (1, 2)→(2, 2)(East)	
	(2, 2)
(2, 2)→(3, 2)(East)	
	(3, 2)
(3, 2)→(3, 1)(South)	
	(3, 1)
No succesors which were not already expanded	
GOAL	(1, 1)

Solution (3,3), (2,3),(1,3),(1,2),(1,1)



Nodes added to the frontier	Extracted node
	(3, 3)
(3, 3)→(3, 2)(South) (3, 3)→(2, 3)(West)	
	(2, 3)
(2, 3)→(2, 2)(South) (2, 3)→(1, 3)(West)	
	(1, 3)
(1, 3)→(1, 2)(South)	
	(1, 2)
(1, 2)→(1, 1)(South) (1, 2)→(2, 2)(East)	
	(2, 2)
(2, 2)→(2, 1)(South) (2, 2)→(3, 2)(East)	
	(3, 2)
(3, 2)→(3, 1)(South)	
	(3, 1)
(3, 1)→(2, 1)(West)	
	(2, 1)
(2, 1)→(1, 1)(West)	
	(1, 1)
GOAL	

Solution: (3,3), (2,3),(1,3),(1,2),(2,2), (3,2),(3,1),(2,1), (1,1)

Figure 3.1: Expanded states and solution for DFS on similar layouts: with/without wall

3.1 Informed Search A^*

Informed search strategy uses problem-specific knowledge beyond the definition of the problem itself. Best-first search is Graph-search in which a node is selected for expansion based on an **evaluation function**. Evaluation function f is an *estimation* of the real cost.

A^* algorithm is the most widely known best-first search algorithm. The evaluation of a node combines the cost to earch the node from the initial state with the estimated cost to get from the node to the goal.

$$f(n) = g(n) + h(n)$$

$h(n)$ is the heuristic. An admisible and consistent heuristic guarantees the optimality of A^*.

3.1.1 Admissibility and consistency

An admissible heuristic never overestimates the cost to reach the goal. A consistent heuristic meets the following condition for each nodes n and n':

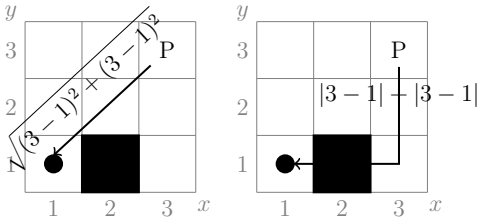
$$h(n) \leq c(n, a, n') + h(n')$$

Exercise 3.3 *Read from AIMA section 3.5.2 - Minimizing the total estimated solution cost.*

Exercise 3.4 *On the same layout 3x3 as the exercise 2 analyze the behaviour of A^* with Manhattan Distance as heuristic. Two heuristic used in Position Search Problem are: Manhattan Distance and Euclidian Distince, defined in* `searchAgents.py` *(see figure 4.2)*

Nodes added to frontier	Extracted	g	h	f
	(3, 3)			
(3, 3)→(3, 2)(South)		1	3	4
(3, 3)→(2, 3)(West)		1	3	4
	(3, 2)			
(3, 2)→(3, 1)(South)		2	2	4
(3, 2)→(2, 2)(West)		2	2	4
	(2, 3)			
(2, 3)→(2, 2)(South)		2	2	4
(2, 3)→(1, 3)(West)		2	2	4
	(3, 1)			
	(2, 2)			
(2, 2)→(1, 2)(West)		3	1	4
	(1, 3)			
(1, 3)→(1, 2)(South)		3	1	4
	(1, 2)			
(1, 2)→(1, 1)(South)		4	0	4
	(1, 1)			

Figure 3.2: Euclidian and Manhattan Distance



The cost of paths increases, while the heuristics decreases as we get closer to the goal. The value of every heuristic in goal must be zero.

3.2 Implementation exercises

Exercise 3.5

Question 4 *Go to* aStarSearch *in* ***search.py*** *and implement A^* search algorithm. A^* is graphs search with the frontier as a priorityQueue, where the priority is given by the function $g = f + h$*

Pay attention to

- the number of arguments of *aStarSearch* function
- the number of arguments of the heuristics: two heuristics are defined for *PositionSearchProblem* in *searchAgents.py* - Manhattan Heuristic and Euclidian Heuristic TODO: add drawing for each of this
- the real cost of a node from the initial state does not depend on the heuristic; only the path from the initial state to the goal state through that node depends on the heuristic
- if you already implemented DFS or UCS as graph search, for A^* the changes should relate mainly to the frontier

Test your implementation by searching for a solution for finding a food dot with the use of Manhattan Distance heuristic by using the option *heuristic*

```
$python pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=astar,
   heuristic=manhattanHeuristic
```

- Does A^* and UCS find the same solution or they are different?
- Does A^* finds the solution with fewer expanded nodes than UCS? The answer should be yes, for *bigMaze* you could get 549 with A^* vs. 620 with UCS search nodes expanded, but these values can differ according to the order you put into the frontier the successor nodes.
- For more details, go to the project's page `http://ai.berkeley.edu/search.html#Q4`
- Test with autograder that you obtain 3 points for Question 4.

A^* algorithm is a very powerful algorithm. In order to prove this, you will work with another two Search Problems: finding all four corners and eating all the food.

Exercise 3.6 ***Question 5 Finding all the corners*** *Pacman needs to find the shortest path to visit all the corners, regardless there is food dot there or not. Go to* CornersProblem *in* ***searchAgents.py*** *and propose a representation of the state of this search problem. It might help to look at the existing implementation for PositionSearchProblem. The representation should include only the information necessary to reach the goal. Read carefully the comments inside the class* CornersProblem*.*

For hint and more details, go to the project's page `http://ai.berkeley.edu/search.html#Q5`.

Test your implementation with BFS - remeber that BFS finds the optimal solution in number of steps (not necessarilly in cost), and in case of this problem, the cost for each action is the same.

```
python pacman.py -l tinyCorners -p SearchAgent -a fn=bfs,prob=
   CornersProblem
python pacman.py -l mediumCorners -p SearchAgent -a fn=bfs,prob=
   CornersProblem
```

For *mediumCorners*, BFS expands a big number - around 2000 search nodes. It's time to see that A^* with an admissible heuristic is able to reduce this number.

Exercise 3.7 ***Question 6*** *Implement a consistent heuristic for CornersProblem. Go to the function* cornersHeuristic *in* ***searchAgent.py****.*

Test it with

```
$python pacman.py -l mediumCorners -p SearchAgent -a fn=aStarSearch,
   prob=CornersProblem,heuristic=cornersHeuristic
or
$python pacman.py -l mediumCorners -p AStarCornersAgent -z 0.5
```

The heuristic is tested for being consistent, not only admissible. Grading depends on the number of nodes expanded with your solution.

Number of nodes expanded	Grade
more than 2000	0/3
at most 2000	1/3
at most 1600	2/3
at most 1200	3/3

Exercise 3.8 ***Question 7*** *Propose a heuristic for the problem of eating all the food-dots. The problem of eating all food-dots is already implemented in FoodSearchProblem in* ***searchAgents.py****.*

- *Test your implementation of A^* on FoodSearchProblem by running*

  ```
  python pacman.py -l testSearch -p AStarFoodSearchAgent
  identical to
  python pacman.py -l testSearch -p SearchAgent -a fn=astar,prob=
     FoodSearchProblem,heuristic=foodHeuristic.
  ```

  The existing heuristic foodHeuristic returns 0 for each node (trivial heuristic), therefore the previos running is the same with UCS. For the layout testSearch the optimal solution is of length 7, while for tinySearch layout is of length 27. Very importantly, the number of expanded nodes is very large: more then 5000 nodes.

- *Go to foodHeuristic function in* ***searchAgents.py*** *and propose an admissible and consistent heuristic. Test it on trickySearch layout. On this layout, UCS explores more than 16000 nodes.*

- *Test with autograder. Your score depends on the number of expanded states by A^* with your heuristic.*

Number of expanded nodes	*Grade*
more than 15000	*1/4*
at most 15000	*2/4*
at most 12000	*3/4*
at most 9000	*4/4 (full credit; medium)*
at most 7000	*5/4 (optional extra credit; hard)*