Chapter 4

Adversarial search - MultiAgent

Learning objectives for this week are:

1. To understand how optimal decision can be taken in multi-agent environment
2. To implement a good evaluation function of a Pacman game state
3. To implement MINIMAX algorithm and Alpha-beta prunning

Download multiagent from `https://s3-us-west-2.amazonaws.com/cs188websitecontent/projects/release/multiagent/v1/002/multiagent.zip` The description of the project is at `http://ai.berkeley.edu/multiagent.html`. Now we will consider also the ghosts. Pacman can do 5 actions: N, S, E, W, Stop. Pacman can eat power pellet and scare the ghosts: for a certain number of moves, the ghosts will be scared and Pacman can eat them. Rules for the score: one food: +10, win/loose: +500/-500, time: -1. All the rules are defined in class *PacmanRules* from `pacman.py` file. The class *GameState* from `pacman.py` specifies the full game state. Read its comments!

Exercise 4.1 *python exercise: Use list comprehension for computing*

$$S = \{x^2 | x \in \{0...9\}\}$$

$$T = \{1, 13, 16\}$$

$$M = \{x | x \in \ S \ and \ x \in \ T \ and \ x \ even\}$$

Solution 4.1 *List comprehension:*

```
>>> s=[x**2 for x in range(0,9)]
>>> t=[1,13,16]
>>> m=[x for x in s if x % 2 == 0 and x in t]
>>> s
[0, 1, 4, 9, 16, 25, 36, 49, 64]
>>> m
[16]
```

4.1 Reflex Agents

Exercise 4.2 *Create a reflex agent which chooses at each step a random action from the legal ones.*

Observation: this is different from the random search agent, since this is a **reflex** agent - it does not build a sequence of actions, but chooses at each step one action.

Solution 4.2 *Add to* ***multiagent.py*** *the class RandomAgent.*

```
class RandomAgent(Agent):

    def getAction(self, gameState):
        legalMoves = gameState.getLegalActions()
        chosenIndex = random.choice(range(0, len(legalMoves))) # Pick
           randomly among the legal
        return legalMoves[chosenIndex]
```

Run with

```
python pacman.py -p RandomAgent -l testClassic
```

In some situation, Pacman wins, but most of the time he looses.

Exercise 4.3 *A better reflex agent is already described in* ***multiagent.py****. Run and analyze* $ReflexAgent$ *from* ***multiagent.py****.*

```
python pacman.py -p ReflexAgent
python pacman.py -p ReflexAgent -l testClassic
```

Read the content of functions *getAction*, $evaluationFunction$, $scoreEvaluationFunction$.

Solution 4.3 *This agent is a reflex one, this means that it will choose its action based only on its current perception. The* $ReflexAgent$ *gets all its legal actions, computes the scores of the states reachable with this actions and chooses the one with the maximum score. In case more states have the maximum score, it will choose randomly one.*

Of course the agent will loose many times. Think about how you could improve it by using the variables newFood, newGhostState, newScaredTimes. Print these variables in function $evaluationFunction$ *from the* $ReflexAgent$*.*

Exercise 4.4 *For all legal actions, for the resulting states print the position of Pacman, the position of ghosts, and the distance between Pacman and the ghosts:*

Solution 4.4 *Add in* $evaluationFunction$ *from* $ReflexAgent$*:*

```
distanceToGhosts = [manhattanDistance(newPos, gp) for gp in
   successorGameState.getGhostPositions()]
print "New position", newPos
print "Ghost positions", successorGameState.getGhostPositions()
print "Distance to ghosts", distanceToGhosts
```

Hint: The Grid class (used for food) has the method asList() which returns a list of positions.

Exercise 4.5 *Improve the ReflexAgent such that it chooses a better the action. Include in the score food locations and ghost locations. The layout testClassic should be solved more often. test wit with autograder for* ***Question 1***

```
python pacman.py -p ReflexAgent -l testClassic
```

You can turn off the animation with the option $frameTime$. The number of ghost is given with option k.

```
python pacman.py --frameTime 0 -p ReflexAgent -k 1 -l mediumClassic
python pacman.py --frameTime 0 -p ReflexAgent -k 2 -l mediumClassic
```

For an average evaluation function, the agent will loose for 2 ghosts. More details can be found at `http://ai.berkeley.edu/multiagent.html`.

Grading:

```
python autograder.py -q q1
or
python autograder.py -q q1 --no-graphics
```

out of 10 runs on openClassic layout:	
0	the agent times out or looses all the time
1	the agent wins at leat 5 times
2	the agent wins all 10 games
+1	the average score is > 500
+2	the average score is > 1000

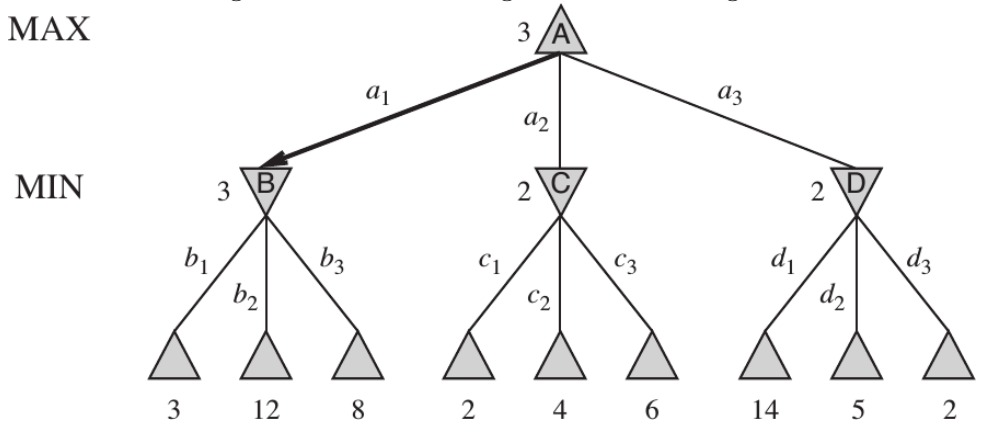
4.2 Minimax

In case the world where the agent **plans** ahead includes other agents which plan against him, adversarial search can be used. One agent is called MAX and the other one MIN. $Utility(s, p)$ (called also payoff function or objective function) gives the final numeric value for a game that ends in terminal state s for player p. For example, in chess the values can be +1, 0, $\frac{1}{2}$. The *game tree* is a tree where the nodes are game states and the edges are moves. MAX's actions are added first, then for each resulting state, the action of MIN's are added, and so on. A game tree can be seen in figure 4.2.

Optimal decisions in games must give MAX's move for the initial state, then MAX's moves in all the states resulting from each possible response by MIN, and so on. MINIMAX value ensures optimal strategy for MAX and identifies the action for MAX. For example, for the agent in figure 4.2, the best move is a_1 and the minimax value of the game is 3. (see chapter 5 from AIMA).

$MINIMAX(s) =$
=UTILITY(s) if TERMINAL-TEST(s)
$=max_{a \in Actions(s)}$ MINIMAX(RESULT(s,a)) if PLAYER(s)= MAX
$=min_{a \in Actions(s)}$ MINIMAX(RESULT(s,a)) if PLAYER(s)=MIN

Figure 4.1: Minimax algorithm for two agents



```
function MINIMAX–DECISION (state) returns an action
   return arg max_{a∈ACTIONS(state)} MIN–VALUE(RESULT(state, a))

function MAX–VALUE(state) returns a utility value
 if TERMINAL–TEST(state) then return UTILITY(state)
 v ← −∞
 for each a in ACTIONS(state) do
   v ← MAX(v, MIN–VALUE(RESULT(state, a)))
return v

function MIN–VALUE(state) returns a utility value
 if TERMINAL–TEST(state) then return UTILITY(state)
 v ← ∞
 for each a in ACTIONS(state) do
   v ← MIN(v,MAX–VALUE(RESULT(state, a)))
return v
```

Result(*state*, *a*) is the state which results from the application of action *a* in *state*. MINIMAX algorithm generates the entire game search space. Imperfect real-time decisions involve the use of cutoff test based on limiting for example the depth for the search. When the CUTOFF test is met, the leaves are evaluated using an heuristic evaluation function instead of the Utility.

H-MINIMAX(s,d) =
=EVAL(s) if CUTOFF-TEST(s, d)
$=max_{a \in Actions(s)}$ H-MINIMAX(RESULT(s,a), d+1) if PLAYER(s)= MAX
$=min_{a \in Actions(s)}$ H-MINIMAX(RESULT(s,a), d+1) if PLAYER(s)=MIN

Exercise 4.6 *Implement H-Minimax algorithm in MinimaxAgent class from* `multiAgents.py`. *Since it can be more than one ghost, for each max layer there are one or more min layers. Test it with autograder for* ***Question 2*** *or at certain depth and layouts:*

```
python autograder.py -q q2
python pacman.py -p MinimaxAgent -l minimaxClassic -a depth=4
```

- One Pacman move and all ghost responses make one ply in the game tree. So depth 2 search involves Pacman and ghosts moving two times.
- The function *getAction* from *MinimaxAgent* class returns the minimax action from the current game state with depth having the value $self.depth$.
- You must be able to limit the game tree to an arbitrary depth mentioned with option $-a\ depth = 4$. The depth is stored in $self.depth$ attribute. In order to give the evaluation/score for the leaves of the minimax tree use $self.evaluationFunction$

For Hints and Observation, go to `http://ai.berkeley.edu/multiagent.html`

- It is normal Pacman to lose in some cases.
- For layout *minimaxClassic* the minimax values of the initial state are: 9, 8, 7, −492 for depths 1, 2, 3, and 4.
- Test Pacman on trappedClassic layout and try to explain its behaviour.

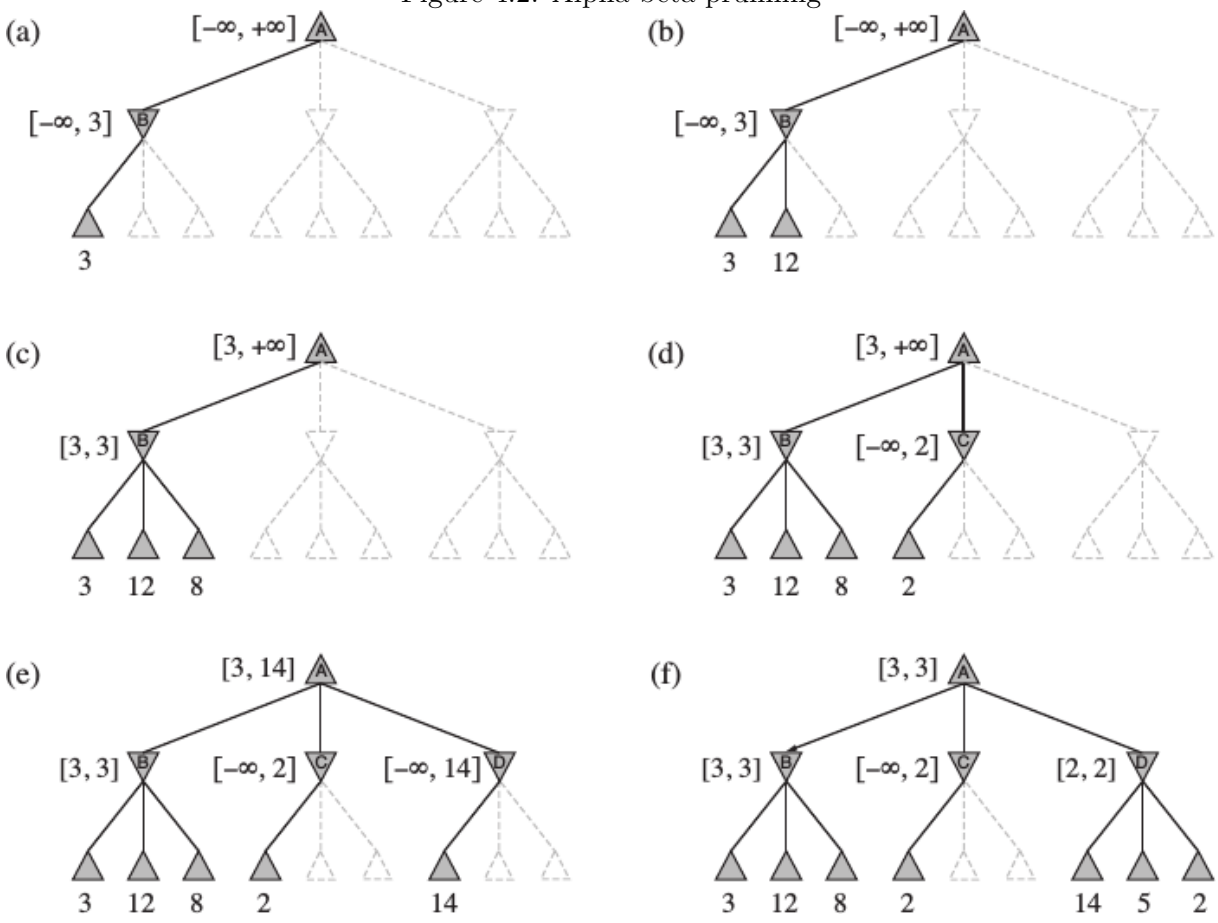
```
python pacman.py -p MinimaxAgent -l trappedClassic -a depth=3
```

Why Pacman rushes to the ghost? For random ghosts minimax behaviour could be improved.

4.3 Alpha-beta prunning

In order to limit the number of game states from the game tree, alpha-beta prunning can be applied.

Figure 4.2: Alpha-beta prunning



- α = the value of the best **(highest value)** choice there is so far at any choice point along the path for MAX
- β = the value of the best **(lowest-value)** choice there is so far at any choice point along the path for MIN

```
function ALPHA–BETA–SEARCH (state) returns an action
  v ← MAX–VALUE(state, −∞,∞)
return the action in ACTIONS(state) with value v
```

```
function MAX-VALUE(state , α,β) returns a utility value
if TERMINAL-TEST(state) then return UTILITY(state)
v ← −∞
for each a in ACTIONS(state) do
 v ← MAX(v, MIN-VALUE(RESULT(s,a), α,β))
 if v ≥ β then return v
 α ← MAX(α, v )
return v

function MIN-VALUE(state , α,β) returns a utility value
if TERMINAL-TEST(state) then return UTILITY(state)
v ← +∞
for each a in ACTIONS(state) do
 v ← MIN(v, MAX-VALUE(RESULT(s,a),α,β))
 if v ≤ α then return v
 β ← MIN(β, v )
return v
```

Exercise 4.7 *Use alpha-beta prunning in AlphaBetaAgent from* ***multiagents.py*** *for a more efficient exploration of minimax tree. Test it with autograder for* ***Question 3****.*

```
python autograder.py -q q3
or
python autograder.py -q q3 --no-graphics
```

- Similar to the previous exercise, there is one MAX agent and possible more MIN agents.
- The alpha-beta prunning with depth 3 will run comparable to minimax at depth 2. On *smallClassic* the time should be at most a few seconds per move.

  ```
  python pacman.py -p AlphaBetaAgent -a depth=3 -l smallClassic
  ```

  Don't forget that the minimax value obtained with alpha beta prunning is the same with the value obtained for minimax algorithm (both at the same depth).

Constraint given by the autograder: **you must not prune on equality**. In theory, you can also allow for prunning on equality and invoke alpha-beta once on each child of the root node.