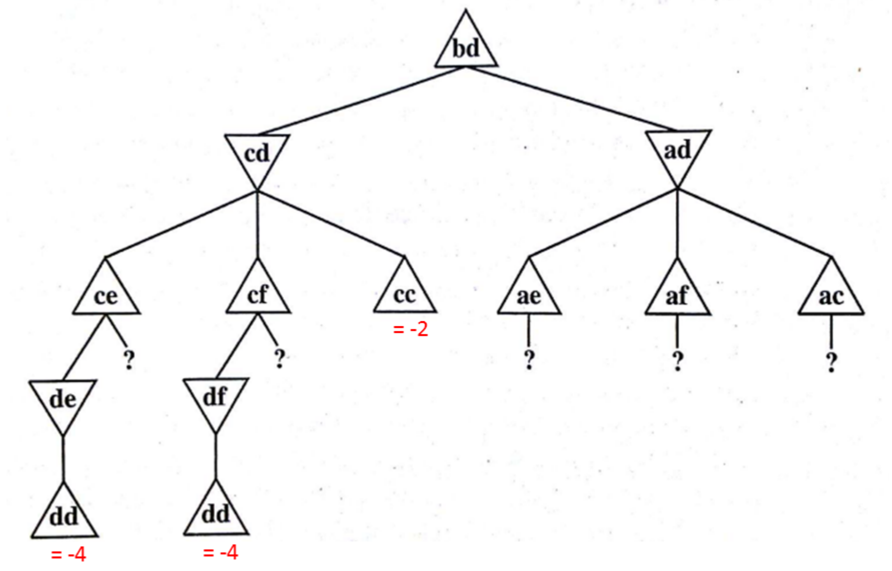
COMP 557: Assignment #2

Due on Thursday, September 15, 2016

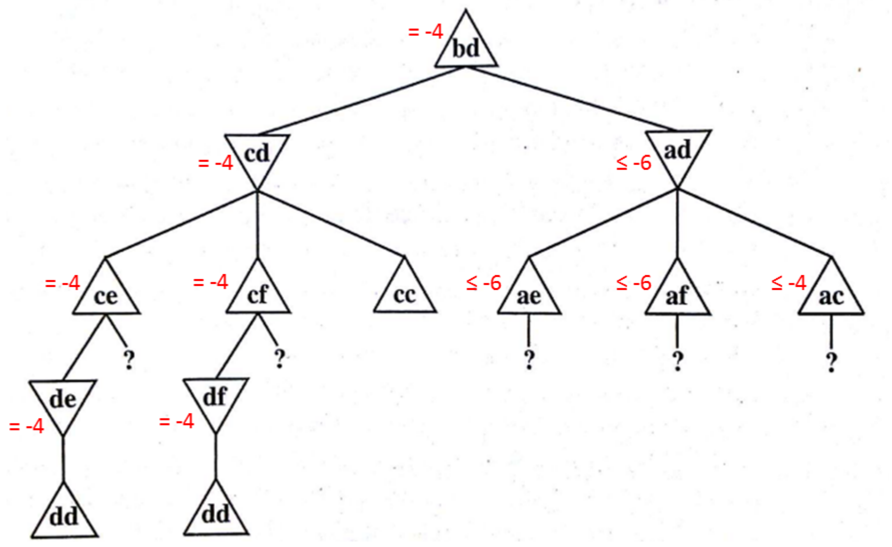
Misiura Mikita & Lee Call

Problem 1.

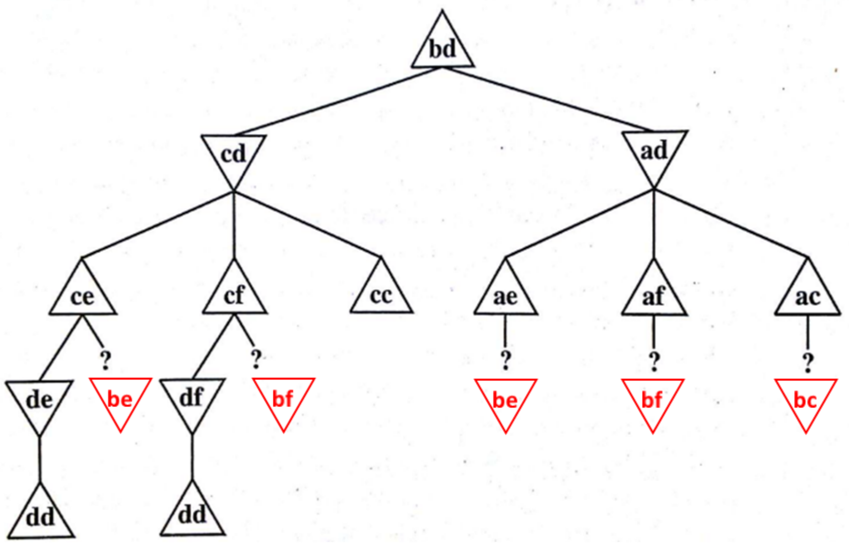
(a) see below



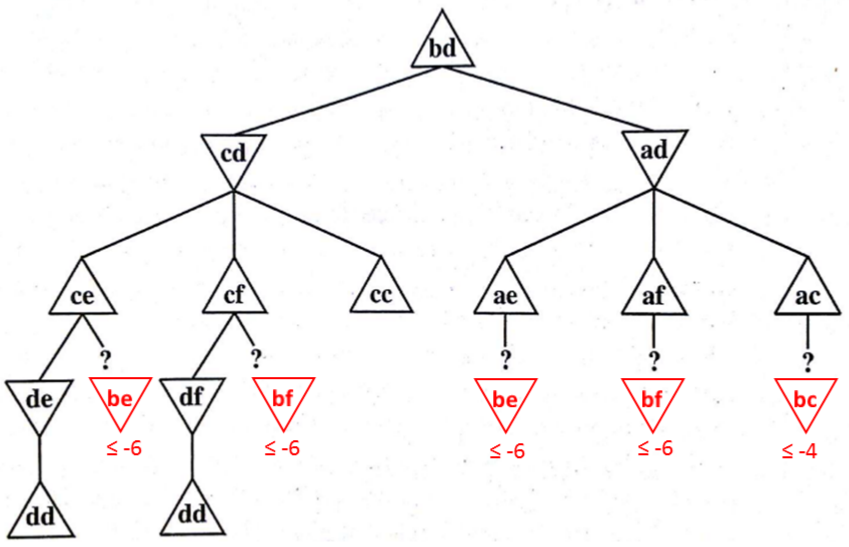
(b) see below



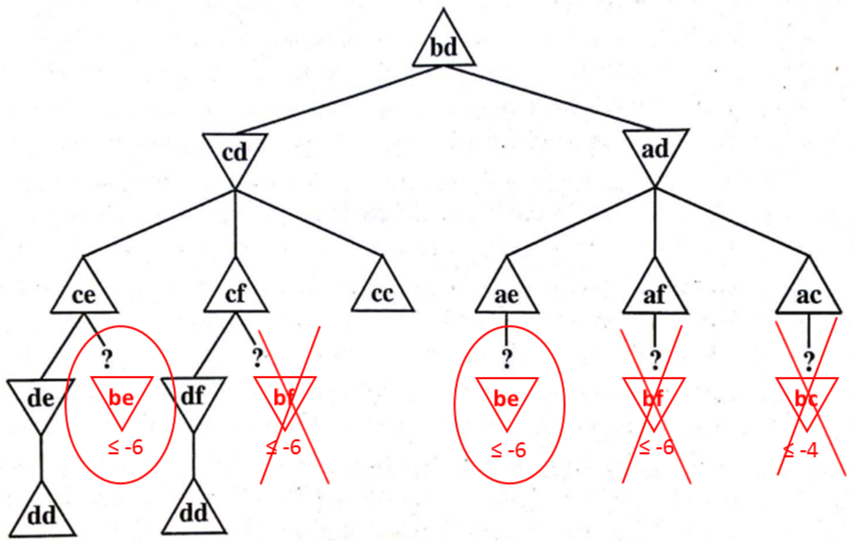
(c) see below



(d) A bound on the value of the nodes can be obtained by the shortest path length between the two players' current locations because the minimum value obtainable from a given node would be achieved if both players only moved towards each other along the shortest path between them. see below



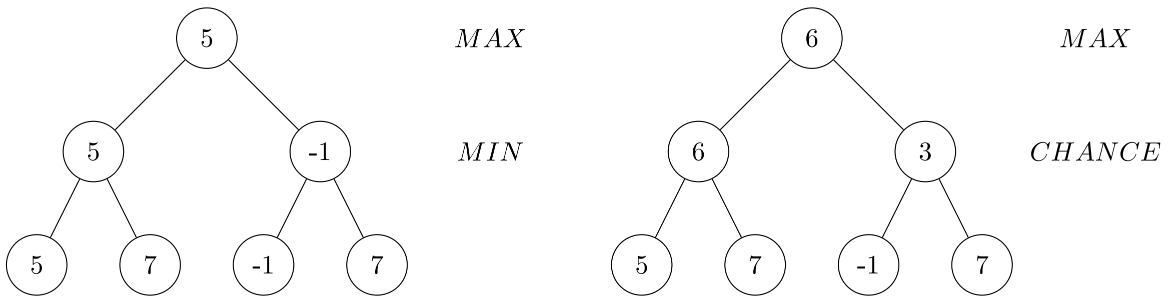
(e) see below



(f) If the playing map is a tree then the pursuer will always win. This is because as a tree, we know it contains no cyclic paths which would be the only way for the evader to win (i.e. by pursuing an infinite-length path).

Problem 2.

(a)



(b) Impossible because average is always higher than the lowest of numbers, used to calculate this average (or at least equal to all of them if they are all the same). Therefore, in case of Expectimax MAX player will always be selecting from a set of higher or equal values than in case of Minimax.

(c) MAX player should use Minimax if he believes that Player 2 is a MIN player, i.e. that opponent is *optimal*.

(d) MAX player should use Expectimax if he knows the distribution of Player 2 moves or knows that Player 2 is random.

(e) Instead of the minimum among MIN's actions, we should use minimum of the expected values of the *successors* of the MIN's actions to predict what action will MIN choose.

$$V_{opt}(s) = \begin{cases} \text{Evaluation(s) } if \text{ isEnd(s)} = True \\ \underset{a\in\text{Actions(s)}}{max} V_{opt}(successor(s,a)) \; if \; it \; is \; MAX \; turn \; in \; s \\ \underset{a\in\text{Actions(s)}}{min} \sum_{\text{successor(s,a)}} \pi(s,a) V_{opt}(successor(successor(s),a)) \; if \; it \; is \; MIN \; turn \; in \; s \end{cases} \tag{1}$$

Problem 3.

3.2.

Equation below assumes that MAX agent has index 0, all other agents are MIN agents. and Player(s) returns index of the player whose turn in the state s. There are n players in the game.

$$V_{opt}(s,d,i) = \begin{cases} \text{Evaluation(s) } if \text{ isEnd(s)} = True \; or \; d = d_{max} \\ \underset{a\in\text{Actions(s,0)}}{max} V_{opt}(successor(s,a,0),d,1) \; if \text{ Player(s)} = 0 \\ \underset{a\in\text{Actions(s,i)}}{min} V_{opt}(successor(s,a,i),d,i+1) \; if \text{ Player(s)} = 1 \, .. \, n-2 \\ \underset{a\in\text{Actions(s,n-1)}}{min} V_{opt}(successor(s,a,n-1),d+1,0) \; if \text{ Player(s)} = n-1 \end{cases} \tag{2}$$

3.2.1

Pacman can be thrashing around because it sees only consequences of the next d steps. If the dot can not be reached within that number of moves, all actions lead to the same result and if our tie braking rule is just to pick random optimal action, it will be most of the time around one place until ghost will not approach.

In our script, we partially changed this behaviour by adding some memory. Pacman will remember its last move and when he will be moving in the corridor, he will be moving in one direction if he can and if it's optimal.

Concerning trapped situation. With $d = 3$ Pacman thinks that there is no chance to escape - moving in both directions will lead to the loss. The result is that it will try to die as soon as possible, because each action costs one point. This fail is the result of our assumption that ghosts are optimal players and will always chase Pacman down if they can.

3.4.

$$V_{opt}(s,d,i) = \begin{cases} \text{Evaluation(s) } if \text{ isEnd(s)} = True \; or \; d = d_{max} \\ \underset{a\in\text{Actions(s,0)}}{max} V_{opt}(successor(s,a,0),d,1) \; if \text{ Player(s)} = 0 \\ \sum_{\text{Actions(s,i)}} \pi(s,a,i) V_{opt}(successor(s,a,i),d,i+1) \; if \text{ Player(s)} = 1 \, .. \, n-2 \\ \sum_{\text{Actions(s,i)}} \pi(s,a,n-1) V_{opt}(successor(s,a,n-1),d+1,0) \; if \text{ Player(s)} = n-1 \end{cases} \tag{3}$$

3.5.

Description of our scoring function:

As a base we use standard evaluation function and then we correct it a bit. Thus, we we are able to get the best from both approaches.

In our evaluation part we have three corrections to the state score:

1. First correction is the distance to the nearest "food" - that is everything, that can be eaten. This can be the food, capsule or even scared ghost. This decrees the final score: the bigger the distance, the lower the score. However, this part has low coefficient of 0.05. This is because of the fact that after eating the food this distance increases significantly, but we don't want to prevent Pacman from eating food. This correction should only affect our decision if there is no food around to eat and we need to direct Pacman to the nearest food.

   To get this distance we use A* search with Manhattan distance heuristic. We used a bit simplified code from the previous assignment with deleted delivery points and modified goal - our goal now is just to pick item up. However, running A* for each piece of food on the map is too slow. That's why we use some sort of heuristic here before running A* - we calculate Manhattan distances to all goodies on the map, sort them according to this distance and then run A* method only on some of them (5 closest in this version). Logic behind this is the following. We have some walls in our world and hence Manhattan distance is not always gives us true estimate, but the points with lower Manhattan distance have higher probability to be the closest. So, picking few of them using just Manhattan distance has quite high chance to find the closest one (and at least it will give us *one of the closest* goodies).

   Our previous "submission.py" is now named "search.py" and previous "util.py" is named "searchutil.py".

2. Second correction is the number of goodies - negative again, the more goodies, the lower the score. This encourages Pacman to each everything he can. This one has higher coefficient of 1.

3. Third correction is the number of capsules. We really want to eat them, so this has coefficient of 300. After the capsule is eaten, ghosts are scared and we include them in the list of goodies encouraging Pacman to eat them.

Berserk mode

By setting $<$ berserk = True $>$ in the code you can force Pacman to forget about all other food and chase only scared ghosts if there are any (unless there is a plenty of food in the immediate vicinity, which is governed by the basic part of the scoring function). This way Pacman usually gets above 1500-1700 points if he stays alive, but this leads to higher probability to get cornered by ghosts - often Pacman runs into their respawn zone and dies there if there is "not scared" ghost around. **Berserk mode is off by default.** Table 1 below shows why - win rate is the best in this case, which is slightly worse with 100 runs, but much more safe with 10 runs.

All corrections are calculated by goodies() function. It returns the tuple of distance to the nearest food, number of foods and number of capsules.

Function create_search_problem() creates delivery problem for A* search.

Our scoring function usually leads Pacman to win (10 out of 10) with average score 1500-1600 unless Pacman is very unlucky (usually when he dies, he dies because he gets cornered by two ghosts). Now Pacman assumes that ghosts are not optimal and there is a chance to win when during the second move one of the ghosts turns away.

Benchmarks

Table 1: Performance of different evaluation functions. Data for each evaluation function is a result of one run with 100 battles (-n 100 -q). Average runtime - is a runtime per one battle.

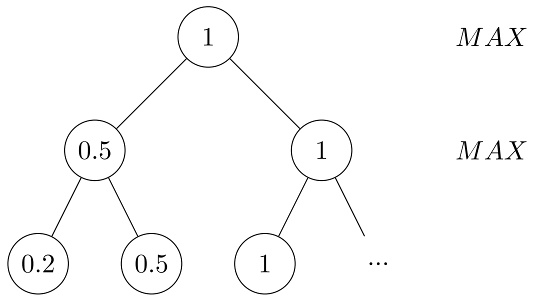
Evaluation function	Average runtime, s	Win rate	Average score
Standard	**2**	0.80	848
Berserk off	15	**0.99**	1516
Berserk on	15	0.95	**1586**

Problem 4.

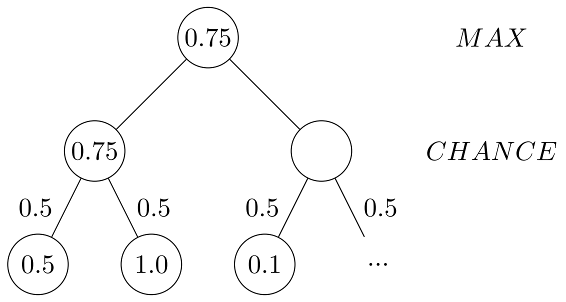
(a) Pruning is never possible in such a max tree because if no bounds are given for leaf nodes then any next leaf have a chance to have a higher value than all the previous ones.

(b) Again pruning is never possible in such a tree because with unbounded leaves there is always the possibility of finding a leaf with a larger value than the current max value already seen.

(c) Pruning is possible here. If the max node value already seen is 1, i.e. the max possible value, there is no need to look further - all other leafs can only give us a tie. Simple example:



(d) Sometimes it will be possible foe example when we know that the last leaf has to have value higher than 1 to make the chance node better than the current best. Example case:



Here the last leaf has to have value at least 1.4 for the second chance node to become better than the first one.

(e) Highest probability first can lead to pruning opportunities. Even with finite but unbounded values we can analyse outcomes of a few nodes with the highest probabilities, get a reliable estimate of the value of this node and decide to prune it if this estimate is significantly lower than the current best. If all our values belong to a certain range, this will work even better.