# CS 4710: Artificial Intelligence

# P2: Adversarial Search and Constraint Satisfaction

- Fall 2021
- Due: Sep 28, 10:00pm

In this assignment, you will implement adversarial search algorithms to help Pacman find paths in a maze while avoiding ghosts and will solve Sudoku as a CSP. Note: We will use the Pacman framework developed at Berkeley. This framework is used worldwide to teach AI, therefore it is very important that you DO NOT publish your solutions online.

## Written Assignment (10 points)

As usual please go to the PDF in the written folder for the written instructions.

Question 1 (3 points)

Question 2 (3 points)

Question 3 (4 points)

## Sudoku -- 15 Points

**Files you'll edit today (and thus submit to Gradescope for Autograding):**

`sudoku.py` Where all of your sudoku code will reside

In a traditional search problem (as in P1), we treat each state of a problem abstractly. A state has a goal test and a transition model, but we never look inside the representation of a state. In a constraint satisfaction problem (CSP), a state is no longer opaque, we can peek in to check if we are on the right track.

In this problem set, you will implement a very common CSP, Sudoku! Sudoku solvers leverage a few essential techniques for solving CSPs.

We have included an autograder as well. **Make sure to run this after every new problem you tackle!**

### Modeling (5 Points)

**Problem 0: Introduction to Sudoku**

If you are not familiar with Sudoku puzzles, read the Wikipedia page on Sudoku to familiarize yourself with the basic rules.

https://en.wikipedia.org/wiki/Sudoku

Due to popular demand, we have migrated the pset over this year from an IPython Notebook into just `.py` files. Should you want to test some functions out using a Notebook, you are free to do so -- and you can use the `prettyprint` function that makes the Sudoku board look nice! -- but we are not officially supporting this. The

easiest way to begin playing with this code is to hop into a terminal IPython session and follow along:

```
robot@robot:~/$ ipython
```

Once inside the shell, you can begin to explore what comprises a Sudoku board:

```
from sudoku import *
sudoku = Sudoku(boardHard)
sudoku.board
```

```
print sudoku.row(1)
print sudoku.col(3)
print sudoku.box(0)
```

**Problem 1: CSP Variables**

For this assignment we will be modelling Sudoku as a CSP. Each box in the grid represents a variable based on (row, col). The variable is either assigned a label (1,2,...,9) or 0 when it has not yet been assigned. Given the current assignment, the domain of each variable is also limited. When all the variables have been assigned the assignment is complete. For the first problem you should implement the following functions in the Sudoku class which model the variables of the Sudoku CSP.

NOTE: **Hints and details about these and all future functions can be found in the docstrings included under the definition of each function.**

```
sudoku.firstEpsilonVariable
sudoku.complete
sudoku.variableDomain
```

**Problem 2: CSP Factors**

Next we will implement the factors of the CSP. The rules of Sudoku say that there must be labels from 1-9 in each row, column, and box. Each of these will be represented by factors (type, id), for instance (ROW, 2) is the factor corresponding to the third-row.

NOTE: These functions should update `self.factorRemaining` and `self.factorNumConflicts`. The datatype of `self.factorRemaining[type, id]` should always be a list of length 9. When a label is no longer available, instead of removing it, it should be replaced by `None`. For conflicts, you should count the number of times any label is used more than once. We provide the helper function `crossOff` to help with this book-keeping.

For this problem, you should implement functions which keep track of the remaining labels available for a given factor as well as the number of violation of that factor in the case of inconsistent assignments. To do this, you should implement the following functions:

```
sudoku.updateFactor
sudoku.updateAllFactors
sudoku.updateVariableFactors
```

## Classical Search (5 Points)

**Problem 3: Solving Sudoku with Backtracking**

The `solveCSP` function will simply perform a depth first search on a tree of generic problem states. Running this function requires getting the next variable to search and all the possible labels that variable can take on.

First you should implement the function `getSuccessors` which should return a list of Sudoku objects representing all possible successor assignments resulting from assigning a label to a variable. Note that for simplicity, unlike the pseudo-code in class we are not doing backtracking (undoing the assignments). This function will need to call `setVariable` which copies the state to produce new assignments.

```
sudoku.getSuccessors
```

After you have implemented this method, run the program by running:

```
robot@robot:~/$ python sudoku.py
# or to see each stage use python sudoku.py --debug=1
```

Notice how many states the CSP solver explores, and how much time it takes to solve the puzzle.

**Problem 4: Improving Performance with Forward Checking**

Now, you will try to improve the performance of your Sudoku solver by implementing forward checking. Forward checking cuts off search when any variable has an empty domain. There are two ways to implement forward checking.

1. Whenever a variable is assigned an update, the domain of any associated variable (i.e. any variable two edges away in the factor graph) is updated as well.
2. Recompute the domains of all unassigned variables on-the-fly to try and find empty variables.

The first method is faster, but requires some more modifications and variable tracking. The second method is simpler, and fine for the Sudoku problem. **PLEASE IMPLEMENT THE SECOND METHOD**

First, take a look at: the function `getSuccessorsWithForwardChecking` and then implement `forwardCheck`.

```
sudoku.forwardCheck
```

Run the program and notice the number of states explored and how much time it takes.

```
robot@robot:~/$ python sudoku.py --forward 1
```

## Local Search (5 Points)

**Problem 5: Sampling Complete Solutions**

In the next several problems we consider a different approach to finding a solution to Sudoku, local search. In local search instead of working with consistent, incomplete

assignments, we will instead use inconsistent, complete assignments. To start we need to sample a random complete assignment. It is often good to start with some randomness but also satisfying some of the factors. For Sudoku we start with the following constraint:

- Sample returns a complete assignment with all Row factors satisfied.

You should implement this with the function `randomRestart`:

```
sudoku.randomRestart
```

When this works, you should be able to start with an incomplete board, call `randomRestart` and notice that the board is now complete, with all rows being consistent (having each of 1,2,...9).

```
sudoku = Sudoku(boardEasy)
sudoku.board
```

```
sudoku.randomRestart()
sudoku.board
```

**Problem 6: Neighbors**

Local search algorithms also require being able to produce neighbors for a given assignment. We will be doing this by stochastic descent, so we will never be required to fully enumerate all the neighbors. Instead we will find neighboring assignments at random.

One way to produce neighbors is to change variables at random. However for Sudoku this hardly ever results in progress towards a consistent solution. For Sudoku we can be a bit more clever and maintain consistency along some of the factors, in particular the row factors.

To produce partially consistent neighbors, we do the following swap:

- Randomly select a row
- Swap two of the entries, being careful not to change any of the original values.

For this problem you will implement this function as `randomSwap`:

```
sudoku.randomSwap
```

**Problem 7: Stochastic Descent**

Finally, you will implement the full local search algorithm. The algorithm should start with a random assignment with consistent rows. At each iteration we run the following check:

1. Sample a neighbor
2. If it has a better or equal score under $f$, then move to that neighbor
3. Otherwise, return to step 1

For the scoring function $f$ we use the current number of constraints that are violated. For efficiency, our implementation keeps a running count of the violations of each assignment. You will have to understand this representation and implement the following function:

```
sudoku.gradientDescent
```

To run the algorithm, we use the following command. Note that for this problem we will use the easier sudoku board as local search is less effective than standard DFS.

```
robot@robot:~/$ python sudoku.py --easy 1 --localsearch=1
```

Run this algorithm 5 times. How often does it find a solution? Instrument your code so that it prints out the number of constraints still being violated.

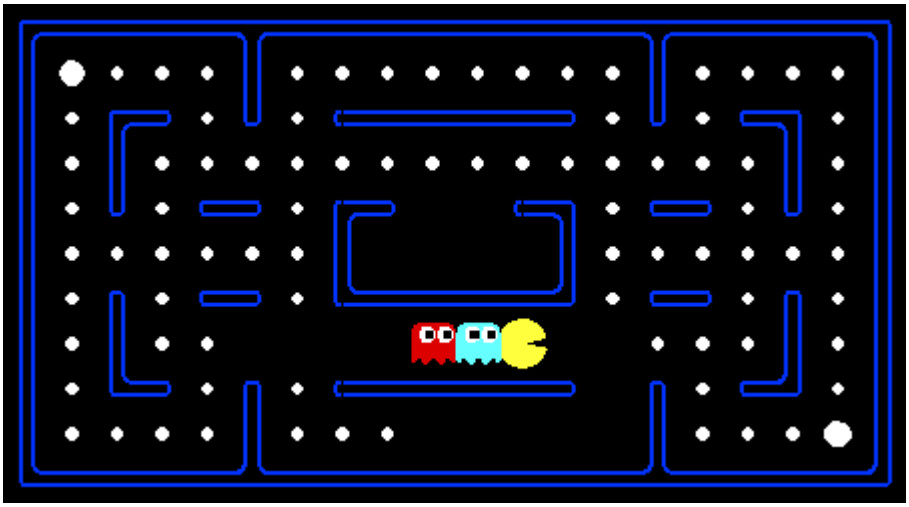
Now modify the stochastic descent function to randomly take some non-optimal moves.

1. Sample a neighbor
2. If it has a better or equal score under $f$, then move to that neighbor
3. With probability 0.001, then move to that neighbor even if $f$ is higher
4. Otherwise, return to step 1

Run this algorithm 5 times on the problem. What do you see now?]

**NOTE: At this point if everythinhg works, you should get a full 15/15 points after running the autograder!**

## Pacman (15 Points)



**Files you'll edit (and thus submit to Gradescope for Autograding):**

`multiAgents.py` Where all of your multi-agent search agents will reside.

In this project, you will design agents for the classic version of Pacman, including ghosts. Along the way, you will implement both minimax and expectimax search and try your hand at evaluation function design.

The code base has not changed much from the previous project, but please start with a fresh installation, rather than intermingling files from project 1.

As in project 1, this project includes an autograder for you to grade your answers on your machine. This can be run on all questions with the command:

`python autograder.py`

It can be run for one particular question, such as q2, by:

`python autograder.py -q q2`

It can be run for one particular test by commands of the form:

`python autograder.py -t test_cases/q2/0-small-tree`

By default, the autograder displays graphics with the `-t` option, but doesn't with the `-q` option. You can force graphics by using the `--graphics` flag, or force no graphics by using the `--no-graphics` flag.

First, play a game of classic Pacman:

`python pacman.py`

Now, run the provided ReflexAgent in multiAgents.py:

`python pacman.py -p ReflexAgent`

Note that it plays quite poorly even on simple layouts:

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

Inspect its code (in multiAgents.py) and make sure you understand what it's doing.

### Question 1 (5 points): Minimax

Now you will write an adversarial search agent in the provided `MinimaxAgent` class stub in `multiAgents.py`. Your minimax agent should work with any number of ghosts, so you'll have to write an algorithm that is slightly more general than what you've previously seen in lecture. In particular, your minimax tree will have multiple min layers (one for each ghost) for every max layer.

Your code should also expand the game tree to an arbitrary depth. Score the leaves of your minimax tree with the supplied `self.evaluationFunction`, which defaults to `scoreEvaluationFunction`. `MinimaxAgent` extends `MultiAgentSearchAgent`, which gives access to `self.depth` and `self.evaluationFunction`. Make sure your minimax code makes reference to these two variables where appropriate as these variables are populated in response to command line options.

*Important: A single search ply is considered to be one Pacman move and all the ghosts' responses, so depth 2 search will involve Pacman and each ghost moving two times.*

Grading: We will be checking your code to determine whether it explores the correct number of game states. This is the only way reliable way to detect some very subtle bugs in implementations of minimax. As a result, the autograder will be very picky about how many times you call `GameState.generateSuccessor`. If you call it any more or less than necessary, the autograder will complain. To test and debug your code, run

`python autograder.py -q q2` Note: we are skipping the Berkely Q1

This will show what your algorithm does on a number of small trees, as well as a pacman game. To run it without graphics, use:

`python autograder.py -q q2 --no-graphics`

Hints and Observations:

1. The correct implementation of minimax will lead to Pacman losing the game in some tests. This is not a problem: as it is correct behaviour, it will pass the tests.
2. The evaluation function for the pacman test in this part is already written ( `self.evaluationFunction` ). You shouldn't change this function, but recognize that now we're evaluating *states* rather than actions, as we were for the reflex agent. Look-ahead agents evaluate future states whereas reflex agents evaluate actions from the current state.
3. The minimax values of the initial state in the `minimaxClassic` layout are 9, 8, 7, -492 for depths 1, 2, 3 and 4 respectively. Note that your minimax agent will often win (665/1000 games for us) despite the dire prediction of depth 4 minimax. `python pacman.py -p MinimaxAgent -l minimaxClassic -a depth=4`
4. Pacman is always agent 0, and the agents move in order of increasing agent index.
5. All states in minimax should be `GameStates` , either passed in to `getAction` or generated via `GameState.generateSuccessor` . In this project, you will not be abstracting to simplified states.
6. On larger boards such as `openClassic` and `mediumClassic` (the default), you'll find Pacman to be good at not dying, but quite bad at winning. He'll often thrash around without making progress. He might even thrash around right next to a dot without eating it because he doesn't know where he'd go after eating that dot. Don't worry if you see this behavior.
7. When Pacman believes that his death is unavoidable, he will try to end the game as soon as possible because of the constant penalty for living. Sometimes, this is the wrong thing to do with random ghosts, but minimax agents always assume the worst: `python pacman.py -p MinimaxAgent -l trappedClassic -a depth=3` Make sure you understand why Pacman rushes the closest ghost in this case.

### Question 2 (5 points): Alpha-Beta Pruning

Make a new agent that uses alpha-beta pruning to more efficiently explore the minimax tree, in `AlphaBetaAgent` . Again, your algorithm will be slightly more general than the pseudocode from lecture, so part of the challenge is to extend the alpha-beta pruning logic appropriately to multiple minimizer agents.

You should see a speed-up (perhaps depth 3 alpha-beta will run as fast as depth 2 minimax). Ideally, depth 3 on `smallClassic` should run in just a few seconds per move or faster.

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

The `AlphaBetaAgent` minimax values should be identical to the `MinimaxAgent` minimax values, although the actions it selects can vary because of different tie-breaking behavior. Again, the minimax values of the initial state in the minimaxClassic layout are 9, 8, 7 and -492 for depths 1, 2, 3 and 4 respectively.

Grading: Because we check your code to determine whether it explores the correct number of states, it is important that you perform alpha-beta pruning without reordering children. In other words, successor states should always be processed in

the order returned by `GameState.getLegalActions`. Again, do not call `GameState.generateSuccessor` more than necessary.

**You must not prune on equality in order to match the set of states explored by our autograder.** (Indeed, alternatively, but incompatible with our autograder, would be to also allow for pruning on equality and invoke alpha-beta once on each child of the root node, but this will not match the autograder.)

The pseudo-code below represents the algorithm you should implement for this question.

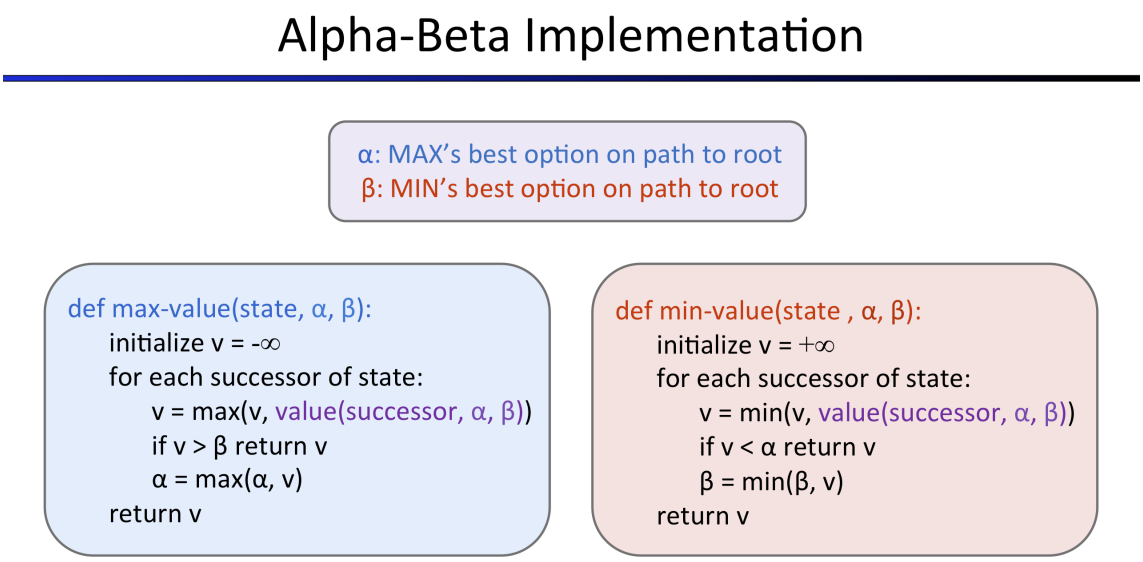
To test and debug your code, run

`python autograder.py -q q3` Again we skipped Q1 so we are off by 1 on the numbers here.

This will show what your algorithm does on a number of small trees, as well as a pacman game. To run it without graphics, use:

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

The correct implementation of alpha-beta pruning will lead to Pacman losing some of the tests. This is not a problem: as it is correct behaviour, it will pass the tests.




### Question 3 (5 points): Expectimax

Minimax and alpha-beta are great, but they both assume that you are playing against an adversary who makes optimal decisions. As anyone who has ever won tic-tac-toe can tell you, this is not always the case. In this question you will implement the `ExpectimaxAgent`, which is useful for modeling probabilistic behavior of agents who may make suboptimal choices.

As with the search and constraint satisfaction problems covered so far in this class, the beauty of these algorithms is their general applicability. To expedite your own development, we've supplied some test cases based on generic trees. You can debug your implementation on small game trees using the command:

`python autograder.py -q q4` Again we skipped the Berkeley Q1

Debugging on these small and manageable test cases is recommended and will help you to find bugs quickly. Make sure when you compute your averages that you use floats. Integer division in Python truncates, so that 1/2 = 0, unlike the case with floats where 1.0/2.0 = 0.5.

Once your algorithm is working on small trees, you can observe its success in Pacman. Random ghosts are of course not optimal minimax agents, and so modeling them with minimax search may not be appropriate. ExpectimaxAgent, will no longer take the min over all ghost actions, but the expectation according to your agent's model of how the ghosts act. To simplify your code, assume you will only be running against an adversary which chooses amongst their getLegalActions uniformly at random.

To see how the `ExpectimaxAgent` behaves in `Pacman`, run:

`python pacman.py -p ExpectimaxAgent -l minimaxClassic -a depth=3`

You should now observe a more cavalier approach in close quarters with ghosts. In particular, if Pacman perceives that he could be trapped but might escape to grab a few more pieces of food, he'll at least try. Investigate the results of these two scenarios:

`python pacman.py -p AlphaBetaAgent -l trappedClassic -a depth=3 -q -n 10`

`python pacman.py -p ExpectimaxAgent -l trappedClassic -a depth=3 -q -n 10`

You should find that your `ExpectimaxAgent` wins about half the time, while your `AlphaBetaAgent` always loses. Make sure you understand why the behavior here differs from the minimax case.

The correct implementation of expectimax will lead to Pacman losing some of the tests. This is not a problem: as it is correct behaviour, it will pass the tests.