ICS 381: Principles of AI – 242

PA 2 – Programming Instructions

General Helpful Tips:

- Do not copy others' work. You can discuss general approaches with students, but do not share specific coding solutions.

- **NOTE1:** For this homework, we will be using itertools so import it.
- **NOTE2:** For this homework, we will be using copy so import it as well.
- **NOTE3:** For this homework, we will be using numpy and chess packages.
- **NOTE4:** Be sure to implement the functions as specified in this document. In addition, you can implement any extra helper functions as you see fit with whatever names you like.

- Submit the required files only: [**csp_scheduler.py, games.py**]

Autograder Note: the autograder on gradescope will take around **100 seconds** to complete.

Implementing CSP Problems

csp_scheduler.py: Implement class for SchedulerCSP

On blackboard you are given **backtracking.py** which contains backtracking search and ac3 implementation. So, in this programming assignment, you will just implement a CSP problem.

Consider a course scheduling problem that we will model as a CSP in **csp_scheduler.py**. There are N courses for which a professor, location, and start-time need to be assigned. For locations, we will have a dictionary of capacity info. For courses, we will have a dictionary for course info where the keys are the course names and values is a list of course info properties: list of preferred professors, number of students, duration, and list of courses that must come before in the schedule.

`loc_info_dict` is location info dictionary with format `{locationname: capacity}`

`course_info_dict` is course info dictionary with format
`{coursename: [list-barred-professors, student-count, duration, list-after-courses] }`

For our CSP modelling we will have:

- Variables: the courses $C_1, C_2, \dots, C_N$
- Domains: courses are assigned a tuple value $C_i = (prof, loc, start\_time)$ indicating professor, location, and start-time. Domain of these three is determined by constructor arguments.
- Constraints:
  - No two courses C_i and C_j can be assigned the same professor at same overlapping time. Also, no two courses can be assigned same location at same overlapping time. You can construct the Boolean logic for this using the 3-tuple value in C_i and C_j, and using the `duration` course info. Notice that this constraint makes the binary constraint graph fully-connected, we check during backtracking.
  - For any C_i that has non-empty `list-after-courses`, it must be assigned a time block that **ends before** the **start** of courses in `list-after-courses`. These are binary constraints that we check during backtracking.
  - For any C_i it must be assigned a location with capacity $\geq$ `student-count`. Notice that this is a unary constraint, which we will enforce on the domains in the constructor before backtracking search.
  - For any C_i it must be assigned a professor that is **not** from `list-barred-professors`. Notice that this is a unary constraint, which we will enforce on the domains in the constructor before backtracking search.

SchedulerCSP Constructor	
Constructor arguments	Constructor body

`courses`, `professors`, `loc_info_dict`, `course_info_dict`, `time_slots`	`courses` is a python list of the N course names as strings.
	
	`professors` is a python list of the M professor names as strings.
	
	`loc_info_dict` is location info dictionary with format `{locationname: capacity}`
	
	`course_info_dict` is course info dictionary with format
	`{coursename: [list-barred-professors, student-count, duration, list-after-courses] }`
	
	`time_slots` is a python list of available time slots $[0, 1, 2, \ldots, maxT]$. A course $start\_time$ is assigned one of these values.
	
	Add the above arguments to self.loc_info_dict, self. course_info_dict, self.time_slots
	
	Then setup the following:
	Set `self.variables` to `courses`.
	
	`self.domains` is a dictionary of variable domain values for the N courses. The key should be the variable course-name and value is domain in form of list of tuples $[(prof, loc, start\_time), \ldots]$. For each C_i you want to restrict its domain as follow:
	• $prof$ should be restricted to be `professors` **not-in** `list-barred-professors` for C_i
	• loc should be restricted to be locations for which capacity $\geq$ `student-count`
	• $start\_time$ domain is the same as `time_slots`.
	• Finally, now you can construct the variable domain as a list of all possible 3-tuples (cartesian product) of the above three. You can achieve this using itertools in python (see this link)
	
	`self.adjacency` is a dictionary for constraint-graph. A keys is variable, and value is list of neighbor variables. For this problem, every variable is connected to all other variables. Just be sure not to add the same variable to its own adjacency list.

SchedulerCSP Functions			
Name	Arguments	Returns	Implementation hints/clarifications
`constraint_consistent`	`var1, val1, var2, val2`	Returns true if var1 and var2 do not violate a constraint. val1 and val2 are the tuple assignment values. $(prof, loc, start\_time)$	Notice for this problem, all variables are connected to each other with a constraint. Some may also have the extra before-constraint. Check the following: • Check1: val1 and val2 should not be assigned same prof at same overlapping time, **nor** same loc at same overlapping time. Be careful with the Boolean logic for this. See text below table. • Check2: If var2 is in var1's after-list, then check that var2's start-time comes **strictly after** var1's end-time. **Also**, check vice-versa if var1 is in var2's after-list. Note the code for this can be done in 4-5 lines. Also, it is not necessary that this check needs to be done; some variables have empty after-list. • Based on the above two checks, return true\false accordingly.

Check1: It is easier to code this in following order:

- check-overlap. Can be done using following Boolean
  not(((start_time1 < start_time2) and (end_time1 <= start_time2)) or ((start_time2 < start_time1) and (end_time2 <= start_time1)))
  Where end_time of course is its start_time + duration.
- check-same-prof-same-overlap: same prof **and** overlapping-time.
- check-same-loc-same-overlap: same loc **and** overlapping-time.
- Check1 is then given by the following Boolean
  `not(check_same_prof_same_overlap or check_same_loc_same_overlap)`

`check_partial_assignment`	`assignment`	Returns true if the partial assignment is consistent.	`assignment` is a dictionary where the key is a variable and the value is a value. If assignment is None, then return False. Check for each variable in assignment that their **assigned neighbors** do not violate constraints. Use constraint_consistent as helper. Be sure to just check **assigned neighbors**. Ignore **unassigned neighbors**.
`is_goal`	`assignment`	Returns true if assignment is **complete** and **consistent**. Otherwise, false.	`assignment` is a dictionary where the key is a variable and the value is a value. If assignment is None, then return False. **Hint**: First check if assignment is consistent. Make use of check_partial_assignment. Then check if it is complete.

Test your code on test_scheduler.py

You can test your code by running test_scheduler.py. My implementation output can be found in test_scheduler.out

Implementing Games

python-chess package

Open the command line. Make sure to activate the `ai_env.yml` that we've been using for our assignments.

```
conda activate ai_env
```

You should have the python-chess package. We will use this package for simulating chess games. Check that package is installed by opening a python terminal and try importing chess.

```
import chess
```

Also go over the introduction for this package. It is intuitive to understand how the engine works. You can also skim over the API for the core functions; you can see there are identifiers for the white and black player, chess pieces, the chess grid locations, Move functions, and Board functions. Of importance are the following Board functions:

- **legal_moves:** note this is a property of the Board class. It generates list of legal moves for the current board position. Note that the board object keeps track of the current player, so the legal moves are generated for the current player.
- **turn:** note this is a property of the Board class. It returns the current player for this turn; returns True for White's turn and False for Black's turn.
- **outcome():** checks if the game is over from various reasons. If the game is not over, this will return None. If the game is over it will return an Outcome object. This will allow us to know if we are in a terminal state and know who is the winner.
- **is_stalement():** returns True/False if position is draw/stalemate. This will allow us to know if are in a terminal state.
- **push(*move: Move*):** Updates the board with the given move and puts it onto the move stack.
- **pop():** undoes the last move made. This is helpful if we want to backtrack when searching the game-tree.
- **pieces(*piece_type: chess.PieceType*, *color: chess.Color*):** Gets pieces of the given type and color. This is helpful for us to get the current board position for both players (White and Black)

See the example code in the introduction to get some intuition on how to use these functions to simulate a chess game. Note that the game proceeds in turns with the White player starting first. So when you progress a turn by calling board.move, it is the next player's turn when you call move again.

For this assignment, we will implement minimax algorithm with heuristics and alpha-beta pruning to play chess. The MAX player will be the White player, with MIN being the Black opponent. For **terminal** states (checkmate) we will assign the utilities of: +1 when White wins, -1 when White loses, 0 when it is a stalemate/draw. The following code should give you an idea on how to check if the game has ended and who is the winner:

```
outcome = board.outcome()

if outcome is None: # if outcome is None the game is still going; i.e. non-terminal state
    print('game has not ended yet')
else: # terminal states
    winner = outcome.winner # True for WHITE player and False for BLACK player
    termination = outcome.termination

    if termination == chess.Termination.CHECKMATE:
        if winner:
            print('White player has won')
        else:
            print('Black player has won')

    else: # for other outcomes, the game is drawn
        print('the game is draw')
```

Recall from class that it would take too long to search the full game-tree for chess. So we will make use of heuristic to compute the estimated utility for **non-terminal** nodes. For this assignment we will start by implementing a heuristic for chess, then implement heuristic-minimax, then implement heuristic-minimax with alpha-beta pruning.

game.py: Implement heuristic function for non-terminal nodes

Function Name: `heuristic_chess(board)`
Arguments: `board` is the current Board object. We can use this to get current board state information like pieces for both players and their positions.
Returns: `heuristic_value` this is the estimated utility of the current board state for WHITE player perspective (MAX).

Implementation:
If the current board state is a terminal state (use board.outcome), then return +1, -1, or 0 depending on the outcome and winner.

If the current board state is a non-terminal state, then compute the following features of the board state:

- $f_{pawn} = (num\ White\ pawns - num\ Black\ pawns)$
- $f_{knight} = (num\ White\ knights - num\ Black\ knights)$
- $f_{bishop} = (num\ White\ bishops - num\ Black\ bishops)$
- $f_{rook} = (num\ White\ rooks - num\ Black\ rooks)$
- $f_{queen} = (num\ White\ queens - num\ Black\ queens)$

Note that you can gather pieces info by making use of the board.pieces function. For example, to get number of black pawn pieces, you can use: `len( board.pieces(chess.PAWN, chess.BLACK) )`

Now compute the heuristic value of the board as:

$$heuristic\_value = \frac{1.25f_{pawn} + 3.5f_{knight} + 5f_{bishop} + 5f_{rook} + 10f_{queen}}{100}$$

We divide by 100 to scale it in the range between -1 and +1.

game.py: Implement is_cutoff as helper for minimax

Function Name: `is_cutoff(board, current_depth, depth_limit=2)`
Arguments: `board` is the current Board object. We can use this to get current board state information like pieces for both players and their positions.
`current_depth` is the current depth of the game-tree. We want to keep track of this to know when we should cut the search.
`depth_limit` This is the cutoff depth to stop searching and compute heuristics.
Returns: True or False

This is a simple function that will be used as a helper for minimax implementation. Given the arguments, do the following:

- If the current board state is a terminal state (use the check board.outcome is not None), then return True.
- If the current_depth is equal to depth_limit, return True.
- Else return False.

game.py: Implement heuristic minimax algorithm

Function Name: `h_minimax(board, depth_limit=2)`
Arguments: `board` is the current Board object. We can use this to get current board state information like pieces for both players and their positions.
`depth_limit` This is the cutoff depth to stop searching and compute heuristics.
Returns: `heuristic_value, move` returns the heuristic value and move that should be made by MAX (White) player. Note this will be a chess.Move object.

Implementation:

Follow the pseudocode in slide 46 of Adversarial Search and Game slides. Implement the two helper functions:

- `max_node(board, current_depth, depth_limit)` to implement MAX node behavior.
- `min_node(board, current_depth, depth_limit)` to implement MIN node behavior.

Some notes for both helper functions:

- You can import numpy and use np.inf and -np.inf to represent $+\infty$ and $-\infty$.
- You can loop on available actions by using board.legal_moves.
- Current_depth should start at 0 for the first MAX node at the root, then it should be incremented by 1 for each subsequent max_node or min_node call (as you move to a lower depth in the tree, increment by 1).
- Be sure to use is_cutoff function to stop the search, and use heuristic_value function to compute the estimated utility. For the move, just return None (null) like in the pseudocode in slide 46.
- **Important**: While looping on actions, to progress the board, you can use board.push(move). Then you can call the MIN or MAX node call with the progress board. However, you definitely want to undo the move once a search call is done by calling board.pop().

game.py: Implement heuristic minimax algorithm with alpha-beta pruning

Function Name: `h_minimax_alpha_beta(board, depth_limit=2)`
Arguments: `board` is the current Board object. We can use this to get current board state information like pieces for both players and their positions.
`depth_limit` This is the cutoff depth to stop searching and compute heuristics.
Returns: `heuristic_value, move` returns the heuristic value and move that should be made by MAX (White) player. Note this will be a chess.Move object.

Implementation:

Implement the two helper functions:

- `max_node_ab(board, current_depth, depth_limit, alpha, beta)` to implement MAX node behavior.
- `min_node_ab(board, current_depth, depth_limit, alpha, beta)` to implement MIN node behavior.

NOTE: I highly recommend copying your code from the previous task. You just need to augment the code with the alpha-beta parameters which requires few modifications to arguments and few additional lines of code. See the pseudocode in slide 47 of Adversarial Search and Game slides.

Test your code

You can test your code by running test_h_minimax.py, test_alpha_beta.py, and test_profiling.py

The results of my implementation for these test files can be found in the refouput*.txt files.

Of interest is the results from test_profiling.py. This test code will run the python profiler to count the number of calls for min and max nodes. For a correct implementation, you should see that alpha beta pruning significantly reduces the number of calls (which reduces the search).