Name: Ly Phung

Student ID: 1696163 lynphung

CM 146 Midterm Exam

11/6/20

This is a take-home exam, **due Tuesday, 11/10/20 at 11:59 PM**. It is open notes & materials.

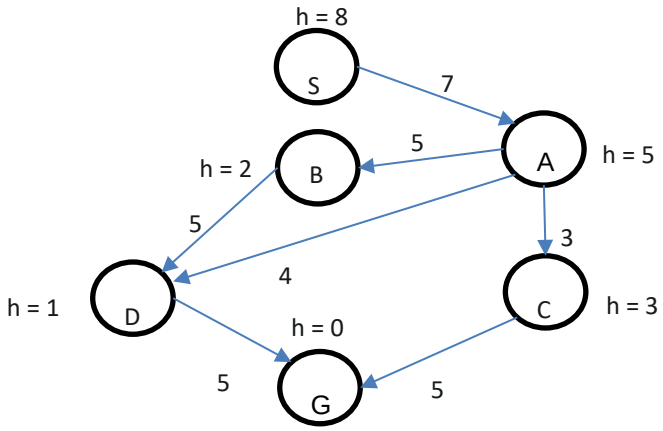
You should submit it electronically via the submission link in the assignment. Please submit a single pdf file with your answers to the problems in order and with problem numbers labeled. Remember to include your name in the file.

As usual, this exam must represent your personal work, per University policy. Please do not discuss the exam with other students until after Monday's class.

Please conduct all Q&A about the exam on Canvas (vs Discord).

You may post clarification questions but not answer them for other students. We will monitor the Canvas discussion and try to answer questions as soon as possible.

CM 146, Midterm
20 points



1. Consider the following directed graph where S is the start and G is the goal, transition costs are on arcs, and h is the heuristic estimate of distance to the goal.

A. **8 points**. Apply Greedy Best First Search to find a path from S to G. Fill out the following table, showing the full contents of the priority queue at the start of each step (before a node is popped from the queue). Show the final path and cost in the last line of the table. Write your entries in this form:

(<node> priority: <priority>, <node>←<parent>) *

Here, * means 0 or more entries of this form. For example,

(D priority: 2, D←A)

indicates D is the only node on the queue, its priority is 2, and D was found by traversing a link from A.

(S priority: 8, S <- None)
(A priority: 5, A <-S)
(D priority: 1, D<-A) (B priority: 2, B<-A) (C priority: 3, C<-A)
(G priority: 0, G<-D) (B priority: 2, B<-A) (C priority: 3, C<-A)
(B priority: 2, B<-A) (C priority: 3, C<-A)
Done! Path: [G D A S] Cost: 16

B. **8 Points.** Apply A* to find a path from the S to G. For each step of the search, fill out the table as above. Identify the final path and cost in the last line of the table.

(S priority: 8, S<-None)
(A priority: 12, A<-S)
(D priority: 12, D<-A) (C priority: 13, C<-A) (B priority: 14, B<-A)
(C priority: 13, C<-A) (B priority 14, B<-A) (G priority 16, G<-D)
(B priority 14, B<-A) (G priority 15, G<-C)
(G priority 15, G<-C) (D priority: 18, D<-B)
(D priority: 18, D<-B)
Done! Path: [G C A S] Cost: 15

C. **4 Points:** Explain why the paths found in parts A and B are the same, or different.

Answer:

The paths found in Part A and Part B are different because Greedy First Search only uses the heuristic value to prioritize the node, but A* uses the heuristic value + the current pathcost, which in turn discovers the most optimal path based on the pathcost, not by just the heuristic value. In this case, since the heuristic value was not accurate, Greedy First Search came up with the wrong path.

2. Watch the great chomper scene from Galaxy Quest (click on the image) or chase this link: https://www.youtube.com/watch?v=gqRdT8m1Suo

If anyone were so cruel as to build the chompers shown in this movie, they would also need a method of cleaning up after any mess the chompers would sadly produce. Here, we delegate that task to a cleaning robot (not seen in Galaxy Quest) that is controlled by the same mechanism that operates the chompers.

Your task is to compose a single Hierarchical Finite State Machine that controls a system composed of two chompers, like those in the movie, together with the cleaning robot. Submit a drawing of your HFSM that clearly labels the states, transitions, transition events, and actions that take place in each state (if they are not synonymous with the state's name). Use hierarchy to prevent duplication of arcs with the same transition condition. Use parallelism as appropriate.

Your HFSM can employ the following events:

- Heartbeat Detected (H)
- No Heartbeat Detected (NH)
- Mess (M)
- No Mess (NM)
- nSec – n seconds have elapsed since entering a state

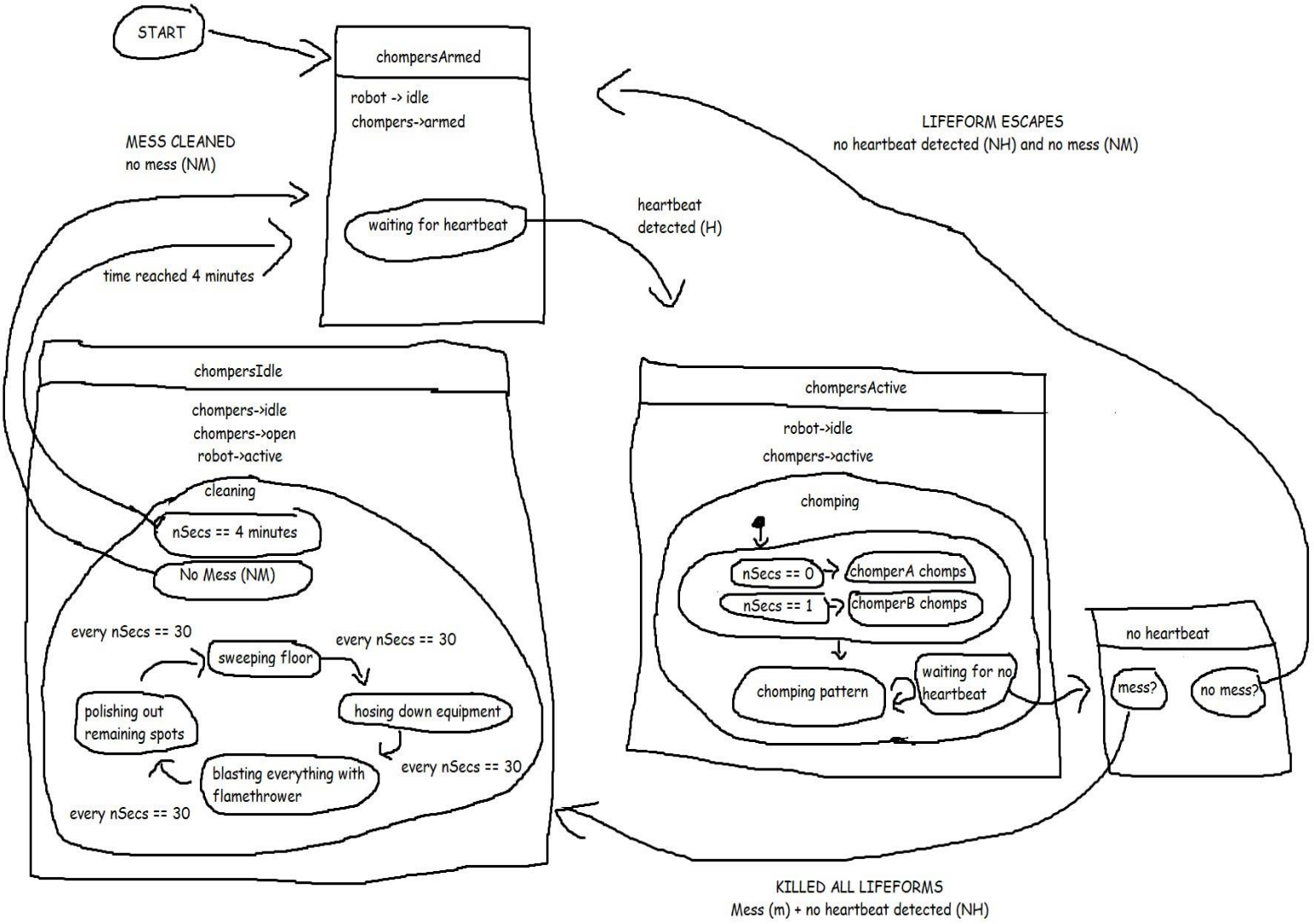
Your HFSM should have the following behaviors:

- While the chompers are active, the robot is idle.
- While the robot is active the chompers are idle. Assume idle chompers are open.
- Chompers begin in an armed state, waiting for anything with a heartbeat to arrive.
- They return to the armed state if the lifeform (or lifeforms) with the heartbeat escapes.
- Once active, chompers keep chomping until nothing with a heartbeat remains.
- The robot is only active when there is a mess.

CM 146, Midterm
20 points

- The robot returns to idle when the mess is gone. It never stays active for more than 4 minutes.
- The robot cleans by sweeping the floor, hosing down the equipment, blasting everything with a flamethrower, and then polishing out any remaining spots, each for 30 seconds. It repeats this process indefinitely, until it is told to stop.
- Each chomper follows a regular, timed pattern of open and close. The patterns are different.
- Chomper A's first chomp is immediate. Chomper B's first chomp is 1 second later.

Sketch your HFSM here or insert an extra sheet.



3. The behavior trees presented in class have encoded deterministic (if time variant) mappings from situation to action. However, characters often seem smarter if they are less predictable. One way to create that effect is to add nondeterminism to behavior trees.

For parts (a) and (b) below, use the expression child.execute() to invoke the Behavior Tree interpreter on a child node.

For parts (a) through (d), assume that the return values of True and False map onto Succeed and Fail, respectively. Also, assume the decorator Until Fail returns true when its child node fails, and Until Succeed returns true when its child succeeds.

(a) **2 points.** Write pseudo-code that implements a *Random Selector*, which is identical to a normal sequence except that it considers its children in a random order, vs a fixed order.

Answer:

```
Random_Selector(children) {
        copy children into child_nodes
        randomize_order(child_nodes)

        for each child in child_nodes:
                if child.execute() Succeed:
                        return Succeed

        return Fail
}
```

(b) **2 points.** Write pseudo-code for a *Random Sequence*, which is identical to a normal Sequence, except it considers its children in a random order, vs a fixed order.

Answer:

```
Random_Sequence(children) {
        Copy children into child_nodes
        Randomize_order(child_nodes)

        For each child in child_nodes:
                If child.execute() Fail:
                        Return Fail

        Return Succeed
}
```

CM 146, Midterm
20 points

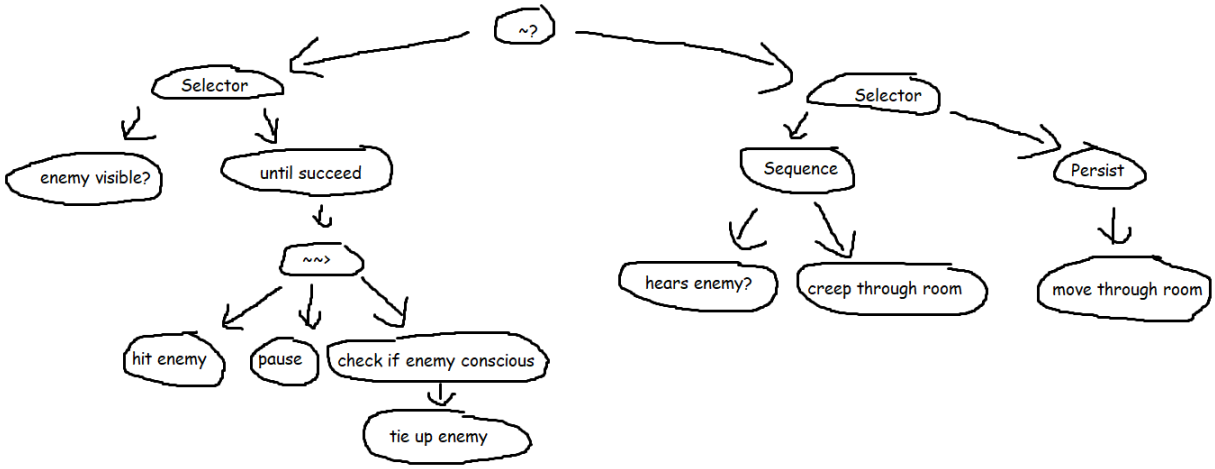
(c) **6 points.** Use these new node types together any of the following (Sequence; Selector; actions; checks; and the decorators Inverter, Until Fail, Until Succeed, Persist) to draw a behavior tree with this specification:

- Your character needs to move through a room. That room might contain a single enemy.
- The character will randomly consider fighting the enemy and crossing the room without a fight.
- Creeping through the room is only relevant if the character hears the enemy. Moving through the room is always relevant. Creeping is preferred.
- Fighting is only relevant if the enemy is visible.
- To fight, the character will hit the enemy, pause, and check to see if the enemy is conscious in a random order. This continues until the enemy is unconscious.
- When the enemy is unconscious, the character ties it up.
- On completion of this behavior, the character has either crossed the room, or is in the room facing a restrained and unconscious enemy.

In your drawing, denote a Random Selector by

and a Random Sequence by

Insert an extra sheet with your answer.



(d) **10 points.** We built a HFSM for a trash collecting robot in class. This question asks you to define a portion of its activity as a behavior tree with parallel branches.

We define a parallel node, denoted as a non-primitive node with an arbitrary number of children. All of those children will be tried once, and all must succeed for the parallel node to succeed. All of the children are launched in parallel. The moment any child returns Fail, the parallel node returns Fail.

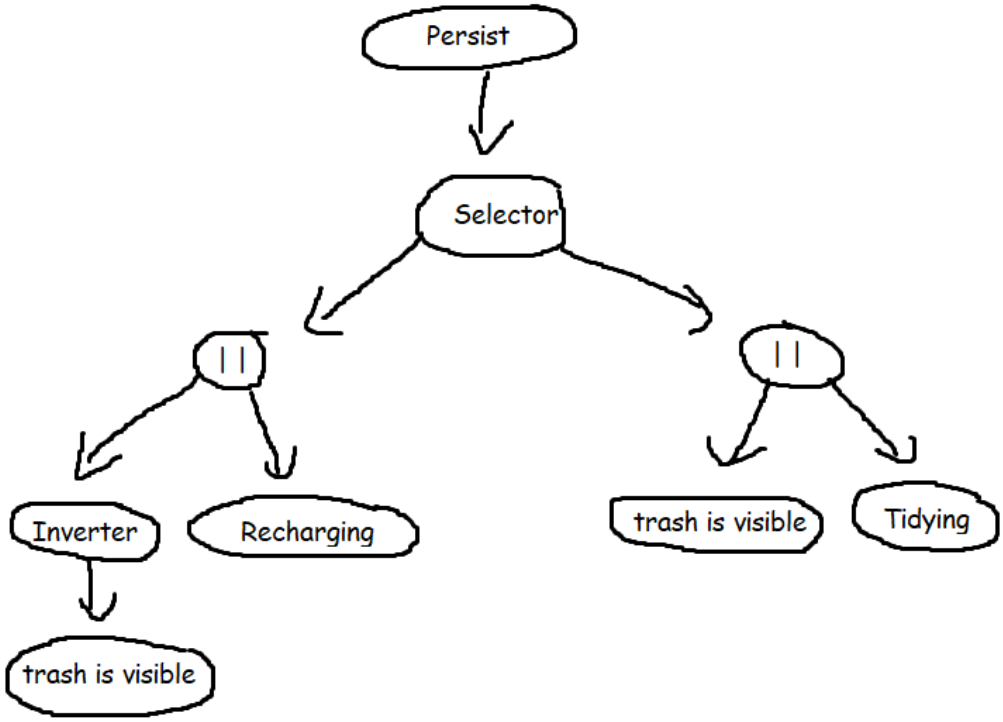
Draw a behavior tree with the following specification:

- The robot is either tidying up trash, or recharging.
- Tidying is only relevant so long as trash is visible.
- Recharging is only relevant so long as trash is not visible.
- The tidying action never terminates of its own (it neither returns Success or Failure).
- The recharging action never terminates of its own (it neither returns Success or Failure).
- The robot continues to execute its tidying and recharging behavior forever.

Denote a parallel node by

As before, you may use Sequence; Selector; actions; checks; and the decorators Inverter, Until Fail, Until Succeed in your tree.

Draw your tree here or insert and extra sheet with your answer.



CM 146, Midterm
20 points

Note: Your responses for this problem should be attempted in Python for clarity. However, you will not be penalized for syntactic errors.

4. Consider the implementation of the Monte Carlo Tree Search below in order to solve this question for some imagined *single-player* game.

```
class Game:
    def __init__(self, *params): pass
def legal_actions(self): pass def
apply_move(self, move): pass
    def is_terminal(self): pass    def
    score(self): pass

class Node:    def    __init__(self,    last_action,    action_list,
parent_node=None):
        self.child_nodes = {}
self.score = 0        self.visits = 0
self.parent = parent_node
self.parent_action = last_action
        self.untried_actions = action_list

def ucb(child_node):
    parent_node = child_node.parent        return
(child_node.score/child_node.visits) + \
sqrt(log(parent_node.visits)/child_node.visits)

def traverse_nodes(node, game, bot_identity):
    while not game.is_terminal() and not node.untried_actions:
        child_node = max(node.child_nodes.values(), key = ucb)
        game.apply_move(child_node.parent_action)
        node    =    child_node
return node

def    backpropagate(score,    node):
while node:
        node.score += score
node.visits += 1
        node = node.parent
```

(a) **8 points.** Partial expansion is a modification to MCTS used primarily for games with large branching factors, i.e. having numerous possible legal actions at every state of the game. It is useful, and sometimes necessary to influence the search tree towards depth vs breadth in this circumstance. One method is to continue the traversal below a node that has untried actions if the number of untried actions is greater than some threshold.

Reimplement the function traverse_nodes with partial expansion, where the threshold to continue traversal compares the number of untried actions against the square root of the node's visit count.

Answer:

```
def traverse_nodes(node, game, bot_identity):
 while not game.is_terminal() and len(node.untried_actions) > math.sqrt(node.visits):
                child_node = max(node.child_nodes.values(), key = ucb)
                game.apply_move(child_node.parent_action)
                node = child_node
return node
```

(b) **8 points.** "Mixmax rewards" is a modification to MCTS that seeks to retain information regarding strong outcomes seen in nodes farther down the tree. It requires two changes:

- Each node must store the maximum of the win rates of any of its children.
- The value for a node---typically just its win rate ---during the tree traversal is now calculated as 80% of its win rate + 20% of the maximum win rate of its children.

Change the appropriate functions to implement mixmax rewards. Assume search tree nodes have an additional field called "max_win_rate".

Answer:

```
def ucb(child_node):
        parent_node = child_node.parent
        return (0.8 * (node.score / node.visits) + 0.2 * node.max_win_rate) + \
                sqrt(log(parent_node.visits)/child_node.visits)

def backpropagate(score, node):
        max_win = node.max_win_rate
        win_rate = 0
        while node:
                node.score += score
                node.visits += 1
                win_rate = node.score/node.visits
                if max_win < win_rate:
                        node.max_win_rate = win_rate
                        max_win = win_rate
                node = node.parent
```

CM 146, Midterm
20 points

(c) **4 points.** Describe the behavior of MCTS with mixmax rewards as you vary the relative importance of a node's win rate and the maximum win rate of its children.

Please insert extra sheet(s) with your answers to 4a - 4c.

Answer:
If you increase the importance of a node's maximum win rate, and decrease the important of the average win rate, your MCTS would focus on nodes with higher maximum win rates, essentially concentrating on fewer nodes, going "depth" first through the tree. However, if you decrease the important of a node's maximum win rate, and increase the importance of the average win rate, your MCTS would focus on exploring averaged win rate nodes, causing your search to be more "breadth" first.

5. Short Questions.
Insert extra sheet(s) with your answers.

(a) **2 points.** Given a 2D square grid that allows only 4 directions of movement (left, right, up, down), a starting cell (x1,y1), a goal cell (x2, y2), and a current cell (x,y). Which of these heuristics for path search makes A* explore the least amount of tiles:

a. (x2 - x)^2 + (y2 - y)^2
b. abs(x2 - x) + abs(y2 - y)
c. abs(x2 - x)
d. abs(y2 - y)
e. Euclidean distance to the goal

(b) **2 points.** Suppose you had a perfect heuristic for an A* search that returned the true cost from a node to the destination. What nodes would A* explore?
a. Nodes on and near the optimal path(s).
b. All nodes that have a path cost less than the total path cost to the goal.
c. Only nodes on the optimal path(s).

(c) **2 points.** By using GOAP, the designers of F.E.A.R were able to reduce the Finite State Machine component of their AI to two states (move to, animate). What work did employing GOAP save relative to encoding the NPC behaviors in F.E.A.R as FSMs (or HFSMs)?

Answer:
Using GOAP allowed the developers of FEAR to only need to use two states to define their AI: move to and animate. That is because GOAP handles a lot of the individual actions that occur within these states. GOAP essentially handles the actions and state transitions between the states, instead of the FSM handling it. GOAP categorizes the actions into general states, reducing the amount of states to a very small amount. The actions that happen inside of each state is handled by GOAP instead of FSM, allowing for more dynamic/ responsive actions to occur, instead of having to define separate states in the FSM for each game state.

(d) 2 points. What capabilities do Behavior Trees bring to the problem of specifying behavior that HFSMs lack (or make difficult to express)? Check all that apply:

1. more reactive response
2. explicit transition logic
3. parallelism
4. specialized control structures
5. separates goals and actions
6. alternative methods of accomplishing a task
7. hierarchical decomposition of tasks

(e) **2 points**. It is often difficult to find heuristics for guiding Goal Oriented Action Planning. The lecture on Advanced Planning suggests using the minimum number of steps required to find the goal if you ignore all items in the delete list of actions. Is this heuristic admissible for GOAP? In 1-2 sentences explain why or why not.

Answer:
This heuristic is admissible for GOAP because by ignoring certain actions of a state, such as actions that are impossible in a state or actions that are unrelated to the Goal state, you can reduce the amount of processing needed to find an optimal path to a State. For example: in the Minecraft GOAP assignment, your goal is to create certain tools. But if those tools don't require iron, you don't need to spend time checking over all the iron related paths, so you ignore those actions and states and narrow down your actions to only those related to the goal.

(f) **2 points**. Consider the following rules, which were present in PromWeek:

The enemy of my enemy is my friend
(Enemies ?x ?y)(Enemies ?y ?z) !(Friends ?x ?z) => (Friends ?x ?z)

The Enemy of my friend is my enemy
(Friends ?x ?y) (Enemies ?y ?z) !(Enemies ?x ?z) => (Enemies ?x ?z)

Assume that working memory contains the following instances:

(Enemies John George)
(Enemies George Herbert)
(Enemies Herbert Rufus)

CM 146, Midterm
20 points

What are the contents of working memory after running the rules in the order shown?

Answer:
(Enemies John George)
(Enemies George Herbert)
(Enemies Herbert Rufus)
(Friends John Herbert)
(Friends Rufus George)
(Enemies John Rufus)

(g) **3 points**. You need to decide what classes to take next quarter. You must take 3 classes out of the following list.

Introduction to belly-button gazing (IBBG)	MWF	12:50 - 13:50
Advanced pumpkin carving (APC)	WTh	13:30 - 15:00
Introduction to Symmetry and its Applications to Quantum Mechanics (IQ)	TTh	15:00 - 16:30
Binge-watching The Good Place (TGP)	WF	14:00 - 23:59
Star Trek: Theory and Applications (STTA)	MWF	14:00 - 15:00

Formulate your scheduling problem as a constraint satisfaction task (show variables and domains, name at least two constraints). You can express constraints in English.

Answer:

Variables: Class, Class time, Class Day

Domains: (Only 5 classes listed above), (12:50-23:59 are only possible class times), (MTuWThF are only possible class days)

Constraints: 1. Classes must not overlap on the same day, at the same time, 2. You must take 3 classes, 3. You cannot take a class twice

Process:

Choose a starting class to start off with. Then choose another class, if that conflicts by same day and same time, then ignore that class and try another class. If you have 3 classes that fit then stop, but if you don't have 3 classes, choose a different starting class and then restart the process. Make sure to not repeat classes.

(h) **2 points**. An HTN approach to Minecraft planning would decompose the task of making a bench into two component tasks:

```
def make_bench_from_planks (state):
  return [(have_4_planks, state), (craft_bench_from_planks, state)]
```

What is the advantage of Hierarchical Task Planning relative to a forward (or backward) search approach that only has access to primitive actions? (1-3 sentences)

Answer:
HTP's advantages is that it decomposes the higher level actions of a task into smaller more primitive operators. In the Minecraft example above: it takes 4 planks to create a bench. HTP would break down that task into two task: having 4 planks and crafting a bench. HTP would then break down those tasks into smaller tasks, such as crafting 4 planks from wood, and having 1 wood. This is better than a search approach because the search approach would cycle through an enormous amount of options, check all of their states, and then give you a path, while HTP simple breaks down the path and returns it.

(i) **3 points**: The following version of traverse_nodes refuses to return a terminal game state; it returns the most desirable non-terminal node in the MCTS tree (or None, if no such node exists). Assume that is_terminal returns true for a terminal game state, and returning None from traverse_nodes causes the calling function (Think) to stop MCTS search and play its best move.

What is the benefit of this code relative to returning a terminal node from traverse_nodes and passing it directly to backpropagate as a game with a known score? (1-3 sentences)

```
def traverse_nodes(node):
        if node.is_terminal() return None
        if node.untried_actions return node for n in
        sort(node.child_nodes.values(), key = ucb):
```

```
        res = traverse_nodes(n)
        if res return res return
    None
```

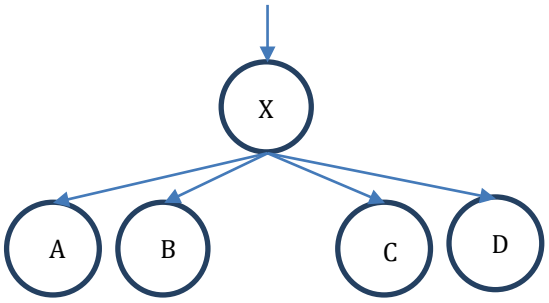
Answer:

Terminating a node at a terminal state is much better than backpropagating a terminal node because it reduces the amount of MCTS loops you need and keeps your values consistent. By using backpropagating a terminal node, its win/loss values would be added to each of its parent nodes, changing the win rates of all the nodes. This might cause your win rates to be incorrect, especially if you exponentially backpropagate your wins/losses. This would cause your MCTS to have less accurate values.

Extra Credit

7 points

Consider the following behavior tree:



X is either a Sequence or Selector node (you do not know which). **A**, **B**, **C**, and **D**, are arbitrary subtrees.

Let **O** be an optional behavior, whose introduction should have no impact on the semantics of the above tree – its Success or Failure, the execution order of its components, or the number of times they are executed.

(a) **2 points.** An unnamed Professor claims that you cannot introduce a subtree between **B** and **C** that encodes an optional behavior, **O**, while leaving the remainder of the tree unchanged (and without altering the implementation of the node type **X**). Is this statement true, or false? Explain why in one or two sentences.

Answer:

This statement is true, if X was a sequence, then O would always have to return true, which is easy using a Succeeder Decorator, which always returns True. However if X is a Selector, this becomes impossible. If O returns true, then the Selector would choose option O and that would disturb the structure of the tree, since the original structure involves C and D. In this case, the selector would ignore C and D. This makes it impossible to add O in between B and C.

(b) **5 points.** Suppose you could transform the tree shown above into another tree by duplicating structure, rewiring structure, and/or introducing additional Sequence and Selector nodes. Draw a transformed tree that encodes an optional behavior, **O**, that will be considered after **B** and before **C**, while preserving the semantics of the original tree.