EECS 649: PROBLEM SET 1

Reading:

- R&N 1.1-5, 27.1-3, Turing's "Computing Machinery and Intelligence" (linked)
- Some of these problems are from previous editions of R&N and other books, so use my problem numbering and disregard any numbers that may appear in problem scans.

Total Points: 100

Format:

- Use standard sheets of paper (8.5 by 11 inches)
- Perform all work neatly. When asked to write a paragraph (as in 1.1-4 below),they should be typed (e.g., using a computer and LaTeX or Word). Other work may bescanned to pdf for submission to Gradesope.
- Submit only a single (combined) PDF file to Gradescope.

Problems 1.1-4 are taken from the linked page above. Write a paragraph for each. **Answers for 1.1-4 must be typed.** A sample problem/answer/rubric similar to 1.1-4 appears below.

SAMPLE Problem (R&N)
Every year the Loebner Prize is awarded to the program that comes closest to passing a version of the Turing test. Research and report on the latest winner of the Loebner prize. What techniques does it use? How does it advance the state of the art in AI?

SAMPLE Answer (due to former student Nathan Makowski)

1.3 The most recent Loebner prize was Robert Medeksza who won for the Ultra Hal Assistant, a digital secretary that learns through interaction and remembers information that the user wants it to save.
Ultra Hal uses language processing in order to have natural conversations with the user. Ultra Hal advances the state of the art by keeping information the user inputs, learning voice recognition and learning about the user. It also interacts with other computer programs in order to store information as well as to help research different topics or access the web.

SAMPLE Grading Rubric
Student has substantially and correctly answered the question (8-10 points). Points will be taken off for missing a piece of the question (e.g., how does the winner advance the state of the art in AI) or for making a technical mistake (e.g., saying an NP-complete problem can't be solved in a finite amount of time). Points will be taken off for incomplete or faulty reasoning. Missing the point or not really answering the question will result in at least half the points off.

Problem 1.1 [10 points] *The Turing Test*
Do the problem below, but only answer the third question: "Can you think of new objections ... ?"

1.2 Read Turing's original paper on AI (Turing, 1950). In the paper, he discusses several objections to his proposed enterprise and his test for intelligence. Which objections still carry weight? Are his refutations valid? Can you think of new objections arising from developments since he wrote the paper? In the paper, he predicts that, by the year 2000, a computer will have a 30% chance of passing a five-minute Turing Test with an unskilled interrogator. What chance do you think a computer would have today? In another 50 years?

Problem 1.2 [10 points] *Are reflexes intelligent?*
Answer the following questions:

Are reflex actions (such as flinching from a hot stove) rational? Are they intelligent?

Problem 1.3 [10 points] *Surely computers can't be intelligent ...*
Answer the following question:

1.11 "Surely computers cannot be intelligent—they can do only what their programmers tell them." Is the latter statement true, and does it imply the former?

Problem 1.4 [20 points] *State of the Art in AI*
Do the following problem. But just pick two of the eleven that interest you the most and write a paragraph for each. You might highlight one to three successes for a particular part if there are competing systems or if "solving" the problem is broad. Make sure your answers are distinct from those in R&N, Section 1.4.

1.14 Examine the AI literature to discover whether the following tasks can currently be solved by computers:

a. Playing a decent game of table tennis (Ping-Pong).
b. Driving in the center of Cairo, Egypt.
c. Driving in Victorville, California.
d. Buying a week's worth of groceries at the market.
e. Buying a week's worth of groceries on the Web.
f. Playing a decent game of bridge at a competitive level.
g. Discovering and proving new mathematical theorems.
h. Writing an intentionally funny story.
i. Giving competent legal advice in a specialized area of law.
j. Translating spoken English into spoken Swedish in real time.
k. Performing a complex surgical operation.

Problem 1.5 [20 points] *Design and build a program to pass one piece of the Turing test*
Formulate a simple cognitive model for addition of pairs of integers. Test it on a few examples. Implement it using a programming language of your choice.

Turn in: your code and some example input/output pairs.

How might you modify or use your model as part of a cognitive model of multiplication?

Notes: For this problem, you may wish to add randomness (to make mistakes at random) and timing (to simulate the amount of time humans need to add a certain number of digits). For some pointers on random numbers, see the last paragraph of the notes on Problem 1.6 below. For a timer, you can use various pause/sleep commands. For example, Matlab has a `pause(n)` *command that pauses for n seconds; Python has* `time.sleep(n)` *with the same functionality.*

You can also repeatedly query a global time function, or use the following structure to add a delay of M seconds:

```
FOR j = 0 to M
   j++;
   FOR i = 0 to BIGNUM
      i++;
```

where `BIGNUM` *is chosen large enough that it takes you computer 1 second to do this. Note that computers today are so fast you may have to use a double loop to achieve delays this long without overflowing the constant integer data type.*

Here are the results of an in-class Turing Test on this question for a previous semester (sample size: 60 total responses)

105721: 54
105,721: 1
105361: 1
105711: 1
105724: 1
115721: 2

Times given as interval (a, b] for a < time <= b
(0,5]: 1; (5,10]: 10: (10,15]: 7; (15,20]: 11; (20,25]: 10; (25,30]: 7; (30,35]: 3; (35,40]: 2; (40,45]: 3; (45, 50]: 1; (50, 55]: 2; (55-60]: 2; plus one at 120.

Problem 1.6 [30 points] *Shannon's "Mind Reading" Machine*

a. Toss a coin 32 times and record the outcome (as a string of H and T);
b. Compute the experimentally observed probability of heads over tosses 2 through 21, inclusive (20 outcomes);
c. Compute the Markov chain transition probabilities over the first 21 tosses (viz., the first 20 transitions).
d. Repeat (a)-(c) above with a sequence verbally derived from a friend not in the class who does not know the underlying model you are trying to construct.
e. Using any language you wish, implement a computer program that uses the Markov chain model to predict the final ten transitions of each data set (throws **23-32** given the values of throws **22-31**, respectively). That is, given the previous state (throw i), compute the next state (throw i+1) using the model, compare with the actual data, and tally the error function (# of wrong guesses).

Again, turn in your code and any input/output pairs used in testing. Comment on your results.

The model you compute in part (c) should be a FIXED model computed ONCE for the ENTIRE test data set (all first twenty transitions).

The states of your Markov chain will be H and T, representing that the current toss is heads or tails, respectively. The four transition probabilities are P(H|H), P(T|H), coming out of H and going to H and T, respectively; plus P(H|T), P(T|T), coming out of T and going to H and T, respectively. These are computed as follows:

$$P(H|H) = \#(HH) / [\ \#(HH) + \#(HT)\ ]$$
$$P(T|H) = \#(HT) / [\ \#(HH) + \#(HT)\ ]$$
$$P(H|T) = \#(TH) / [\ \#(TH) + \#(TT)\ ]$$
$$P(T|T) = \#(TT) / [\ \#(TH) + \#(TT)\ ]$$

Thus, P(H|H)+P(T|H)=1 and P(H|T)+P(T|T)=1.

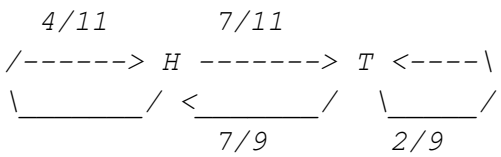
For example, consider the following data set of 31 tosses:

H H H T T H T H T H H T H T H H T H T T H | H H T H T H T H T H

The first transition is HH, the second HH, the third HT, ..., the twentieth TH.

Counting up, the number of HH tranisitions in the first twenty is #(HH) = 4.
Counting up, the number of HT tranisitions in the first twenty is #(HT) = 7.
Counting up, the number of TH tranisitions in the first twenty is #(TH) = 7.
Counting up, the number of TT tranisitions in the first twenty is #(TT) = 2.

This would lead to the model:



These may be computed by hand. ***You do not need to write a program to compute these!***

Hint: for the above programs, you may need a function that picks/returns a random number between 0 and 1. Matlab has `rand` *and Python has* `random.random()`*. Other programming languages might have commands/classes that produce a random integer,* `R`*, from 1 to some integer* `LARGE`*. You can convert this by setting* `r=R/LARGE`*.*

Specifically, for the model above, if the last toss was H, you pick a random number in the interval [0, 1]. If it is less than 7/11, you guess T; else, you guess H.

EECS 649: PROBLEM SET #2

Reading:

- R&N 2.1-4, Braitenberg's *Vehicles*, pp. 1-42 (linked)

Total Points: 100

Format:

- Use standard sheets of paper (8.5 by 11 inches) for your scans
- Perform all work neatly. When asked to write a paragraph, it should be typed (e.g., using a computer and LaTeX or Word).

Problem 2.1 [20 points]
For each of the following, give a PEAS description of the task environment -- in a grid as in Figure 2.5 of R&N -- and characterize it in terms of the properties described in Section 2.3.2 of R&N, putting your answers to the **that** part in a grid as in Figure 2.6 of R&N.

- Exploring the subsurface oceans of Titan
- Shopping for used AI books on the Internet

Problem 2.2 [10 points]
The discrete photovore we talked about in class had a one-dimensional percept with two values: {NoLight, Light}. Thus, using the notation in the middle of page 48 of R&N it has |P|=2 (that is, the total number of percepts is two).

It also had two actions A={Stay, Move}, so |A|=2. Thus, there are a total of 2^2 stimulus/response (S/R) agent programs or **agent functions** (also known as look-up tables or LUTs):

```
NoLight | Stay
Light   | Stay

NoLight | Stay
Light   | Move

NoLight | Move
Light   | Stay

NoLight | Move
Light   | Move
```

How many S/R agents are there in general? Write your answer in terms of the number of percepts and actions, |P| and |A|. Hint: You may wish to test your answer in special cases, such as |A| = 1 or |A| = 2 while |P| = n.

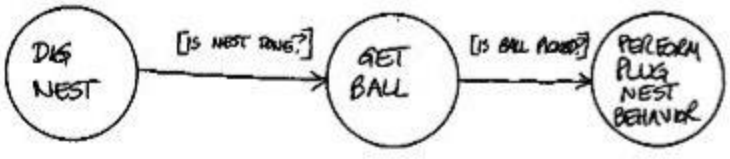
Use your answer to compute the number of S/R agents for the vacuum-cleaner agent pictured in Figure 2.2 (and described on page 37) of R&N.

Hint: Think of the number of functions from the domain to the range.

Problem 2.3 [10 points]
Draw a state machine capturing the behavior of the sphex wasp described on page 41 of R&N.

SAMPLE Problem Like Problem 2.3
Draw a state machine capturing the behavior of the **dung beetle** described on page 41 of R&N.

Sample Answer:



Problem 2.4 [10 points]
Design (the stimulus response curves and connections for) a vehicle that circles a source at a fixed distance, as does the middle vehicle in Braitenberg's *Vehicles,* Figure 7 (page 18).

Problem 2.5 [10 points]
Recall the equations we wrote in class for Figure 10b on page 23 of Braitenberg's *Vehicles*. Do the same for the circuit in Figure 10a, calling the input s, and the outputs of the successive threshold units (left to right) v, w, x, y, and z.

You may do this by hand or write a computer program to figure these out. Your choice. Whatever choice, make sure the answer is organized in a well-labeled table as we did in class.

Problem 2.6 [10 points]
a. Draw the memory circuit described for Vehicle 5 (at the top of page 24 of *Vehicles*).
b. Draw one that will remember if it has seen more than one red light.

Note: we will use the discrete-time model of Problem 2.5 above. Thus, "2 lights" means a red light was seen at two different integer times (which might or might not be contiguous).

Thus, you should draw a total of TWO circuits:

(a) the indicated circuit from the book, which latches (and continues to ring) if the vehicle has seen a light in at least one previous time instance
(b) one of your own design that latches/rings if the vehicle has seen a light in at least two different time instances

As suggested above, to help make your designs more specific, include the buzzer/bell (like Braitenberg's example did), whose ringing signifies the condition in question.

Problem 2.7 [30 points]
Implement a simple reflex agent for the vacuum environment in Figure 2.2. Run the environment with this agent for all possible initial dirt configurations and agent locations. Record the performance score for each configuration and the overall average score.

Note: this requires building a simulator for the vacuum environment in Figure 2.2 and described on page 37. You may write it yourself or use/start with code in R&N's code repository: https://github.com/aimacode/aima-python/blob/master/vacuum_world.ipynb
If you write your own simulator, you do **not** have to make a modular implementation in terms of size and shape, only initial configurations of dirt and robot. You **only** have to simulate the specific size and shape of the environment depicted in Figure 2.2.

NOTE: you may assume that once clean, the rooms stay clean.

There is **no need** to turn in any code. Just turn in a short explanation of how your program is structured. Cite any repository code used and note any significant modifications.

You must report the S/R table used and the performance score for each configuration and its overall average score.

Here are some more details that may help you out.

Overview. You need to write a simulator for a stimulus-response agent (of your design) acting in the 1x2 world. You need to compute its performance measure (of your design) for each of the 8 possible worlds.

The simulation. Your simulator need only work for a stimulus-response agent acting in the 1x2 world. Your simulator should be able to take an arbitrary environment (of the eight possible) as the start location. In that case, you would have to run it 8 times to gather all 8 performance measures for your agent. Alternatively, you could easily loop over all environments, simulating your agent in each one to obtain its performance measure in that environment.

The S/R agent. You need to make an STIMULUS-RESPONSE agent, not an arbitrary agent. For example, my S-R agent (yours can vary) is the following:

A, Clean --> Left
A, Dirty --> Left
B, Clean --> Left
B, Dirty --> Left

If this agent started in room B, with both squares initially dirty, it would move to room A after the first step and then (trying to move Left from there on each step, but running into the wall each time) stay there for as many steps as the simulation lasted.

The performance measure. This is not stated and is up to you. An example of one that makes sense to me for judging/comparing different S/R agents in this environment is as follows: simulate the agent for 10 steps, penalizing it 1 point for each square that is dirty in each time step. So, for the initial state and agent mentioned in the previous point, the performance would be -2 per time step (both rooms are dirty for each time step).

Write-up. Your writeup definitely needs to detail the S/R agent you used as well as the performance measure you used. You also must report the the performace measure on each of the environemts. Finally, it would be appropriate to BRIEFLY explain a little about how your simulator code works (data structures, looping, etc.)

EECS 649: PROBLEM SET #3

Reading:

- R&N 3.1-6

Total Points: 100

Format:

- Use standard sheets of paper (8.5 by 11 inches) for scanning.
- Perform all work neatly. When asked to write a paragraph, it should be typed (e.g., using a computer and LaTeX or Word/Docs).

Problem 3.1 [15 points] *Search problem formulation*

Give a complete problem formulation for each of the following scenarios. In each case, choose a formulation that is precise enough to be implemented.

a. Using only four colors, you have to color a planar map in such a way that no two adjacent regions have the same color.

d. You have three jugs, measuring 12 gallons, 8 gallons, and 3 gallons, and a water faucet. You can fill the jugs up or empty them out from one to another or onto the ground. You need to measure out exactly one gallon.

For each part, also identify the entire state space, *X*.

Problem formulation means States, Initial State, Actions, Transition model, Goal test, Path cost (plus I asked you to add the entire State Space as well.) Examples of these descriptions appear in R&N in Sections 3.2.1 and 3.2.2. For the 8-puzzle, the entire state space is the set of 9! configurations of the tiles.

Problem 3.2 [15 points] *Search spaces*

Do the following problem:

3.15 Consider a state space where the start state is number 1 and each state k has two successors: numbers $2k$ and $2k + 1$.

a. Draw the portion of the state space for states 1 to 15.

b. Suppose the goal state is 11. List the order in which nodes will be visited for breadth-first search, depth-limited search with limit 3, and iterative deepening search.

c. How well would bidirectional search work on this problem? What is the branching factor in each direction of the bidirectional search?

d. Does the answer to (c) suggest a reformulation of the problem that would allow you to solve the problem of getting from state 1 to a given goal state with almost no search?

e. Call the action going from k to $2k$ Left, and the action going to $2k + 1$ Right. Can you find an algorithm that outputs the solution to this problem without any search at all?

Problem 3.3 [20 points] *Understanding A**

(a) Trace the operation of A∗ search applied to the problem of getting to Bucharest from Lugoj **(see Figure 3.1)** using the straight-line distance heuristic. That is, show the sequence of nodes that the algorithm will consider and the f, g, and h score for each node.

Draw a figure like the one in **Figure 3.18(f)** of R&N but make sure each node is labeled with f = g+h and a circled number relating to its order of expansion. The map is **Figure 3.1** and the straight-line distances are in **Figure 3.16**.

(b) Redo (a) using a different heuristic, h', based on h and **one-step look-ahead**. That is, let

h'(n) = min over all i that are successors n of { c(n, i) + h(i) }.

So, let's say a node n has two neighbors: a and b

Then h' = min(cost(n, a) + h(a), cost(n, b) + h(b))

The following problem requires significant programming:

Problem 3.4 [50 points] *Planning in a grid world*

This problem is about planning obstacle-free paths on grids, which arises in applications involving autonomous vehicles and video games.

Specifically, you are to implement routines that solve planning problems that take place in an N by M grid of cells, with initial and goal cells specified by integer pairs: (iinit, jinit) and (igoal, jgoal). The actions available are moving left (decrementing i), right (incrementing i). Down (decrementing j), and up (incrementing j). Moves are possible only if the destination cell is within the grid **and** not an obstacle cell. Each move has a cost of 1.

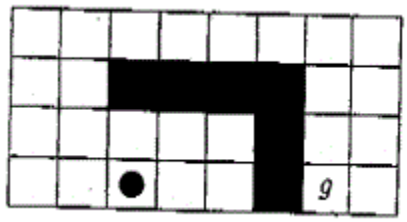
In all cases, the cost-to-reach is the number of moves and the heuristic estimate of the cost-to-go you should use is the "straight-line" Manhattan distance, that is, the sum of the absolute differences in both dimensions (which is admissible because it is the cost associated with the relaxed problem involving no obstacles). For example,

h(xinit) = | iinit - igoal | + | jinit - jgoal |

(a) Implement a general program for such worlds with routines for
   (i) Breadth-First Search (BFS)
   (ii) Greedy Best-First Search
   (iii) A*

In each case, implement your routine *according to the template we used in lecture for FORWARD-SEARCH (also covered in this handout*). You may also see the this linked note. **Turn in your code.**

(b) As a test, try your routines on the example grid we used in class, this linked handout:



Briefly comment on, **but do not turn in**, your results.

(c) Run your routines on the 100 by 100 discrete "bug trap" problem [linked page is from an early edition of Steve LaValle's *Planning Algorithms*], with initial, goal, and obstacle locations given by the following linked code code in both Matlab and Python. Matlab also appears belowl **[Note: Matlab indexes from 1, while Python indexes from 0]**:

```
iinit = 50;
jinit = 55;
igoal = 75;
jgoal = 70;

obs=zeros(100,100); % initialize to "no obstacles"
for i=1:100,
  for j=1:100,
    if i<50,     % rear of bugtrap
      d=abs(i-51)+abs(j-50);
      if d==50, obs(i,j)=1;  end;
    else         % front of bugtrap
      if j>50,   % upper lobe
        d=abs(i-50)+abs(j-75);
        if d==24, obs(i,j)=1; end;
      end;
      if j<50,   % lower lobe
        d=abs(i-50)+abs(j-25);
        if d==24, obs(i,j)=1; end;
      end;
    end;
  end;
end;
```

For each of the three routines, report the number of cells in *closed*, the number of cells on *fringe*, and the length of the path found. There is no need to draw/report the path. Above, *closed* refers to those cells that have been reached and expanded; *fringe* are those that have been reached, but not yet expanded.

(d) **Repeat** the experiment in (c) above with the initial and goal points reversed.

(e) Comment on your results and on the potential of bidirectional search for this class of problems.

EECS 649: PROBLEM SET 4

Reading:

- R&N 4.1, including box on page 118; 4.2 as needed for review; 6.1-4

Total Points: 100

Format:

- Use standard sheets of paper (8.5 by 11 inches) before scanning
- Perform all work neatly. When asked to write a paragraph, it should be typed (e.g., using a computer and LaTeX or Word).

---

*The answers to the following two problems should be **typed**:*

Problem 4.1 [10 points]
[Nilsson] Specify fitness functions for use in evolving agents that

a. control an elevator,
b. control stop lights on a city main street.

Problem 4.2 [10 points]
Give a precise formulation for the following as a constraint satisfaction problem (CSP):

> Class scheduling: There is a fixed number of professors and classrooms, a list of classes to be offered, and a list of possible time slots for classes. Each professor has a set of classes that he or she can teach.

In formulating your answer, consider the definition of CSPs at the start of Section 6.1 and specify X, D, C.

*This problem requires hand calculation and need **not** be typeset:*

Problem 4.3 [20 points]
Solve the cryptarithmetic problem in Figure 6.2 of R&N by hand, using the strategy of backtracking with forward checking and the MRV and least-constraining-value heuristics.

Record the steps and the reasoning/method behind them.
(You solve the problem once, using forward checking, making use of the two stated heuristics to pick variables and values, and recording what you used when.)

Does min-conflicts make sense for this type of problem?

The following two problems require programming:

Problem 4.4 [30 points]
Repeat the one-dimensional optimization experiment from the eHandout of Local Search lecture. **(See the bottom page 2 of the Local Search Handout for that day.)** Specifically, maximize

```
F(x) = 4 + 2x  + 2 sin(20x) - 4 x^2
```

on the interval [0, 1] using fitness-proportional selection (aka roulette selection) of individuals (points of the form 0.01*k, k= 0, ..., 100) and simple mutation (x-epsilon, with probability 0.3; copy with probability 0.4; x + epsilon, with probability 0.3). You may use epsilon=0.01.

Use at least N=10 individuals in your population. Report N and comment on your experiments and results. Turn in documented **code**.

Repeat the above, but add a crossover operator that is a convex combination of two individuals, x and y: a x + (1-a)y, for 0 <= a <= 1.

Use roulette selection instead of the "if fitness(x)>r" selection in my example. Specifically, choose an individual to be "reproduced" proportional to its fitness. So, if in the current generation, you had x1, …, x10, the total population fitness would be TF = sum_i F(xi). Then these would be chosen for "reproduction" in the next generation with probabilities (F(x1)/TF, F(x2)/TF, …, F(x10)/TF).

After mutation, individuals should be clipped to remain within the interval [0,1].

CROSSOVER IS LIKE THIS [you may enforce both parents to be different below if you wish]:
For each individual I want to produce in the next generation:
I pick two parents, each using a different "spin" of roulette selection:
And combine them using crossover, picking a random a

Problem 4.5 [30 points]
Consider the **8-queens Problem** from R&N. Here, you will solve 8-queens using

- Random-restart hill-climbing (RRHC; cf. Section 4.1.1 of R&N)

You will be **minimizing** "fitness," defined as the number of **non-attacking** pairs of queens (as on the bottom of p. 117 of R&N), which is 28 minus "the number of pairs of queens that are attacking each other, either directly or indirectly" (see note in the middle of p. 112 of R&N regarding intervening pieces). Thus, in this problem, when you find a configuration/state with fitness = 28, you have found a solution.

a. Implement the RRHC above. Write your algorithm so that it exits as soon as a solution is found, prints the solution (as a string of digits like those depicted in Figure 4.7 of

R&N), and prints the total number of **fitness evaluations** required from the start of the algorithm.

b. Test your routines enough to convince yourself that they work and that solutions are being found appropriately. **(Do not report on this.)**

c. You will gather statistics regarding the operation of your algorithms by running each algorithm 100 times (from different random starting positions/populations) and report only the **average** of the number of evaluations until a solution is found. **(You may wish to add a cut-off number of iterations that are not exceeded; in this case, don't include failed searches in your average.)**

Turn in your **code** and a **summary** of your results.

Note:
I don't expect you to program this from scratch. This problem is **much, much easier** if you look at or use the code that is already available:

- I have placed some (non-optimized, no-guarantees) C++ code that I used for a more extensive exploration of local search algorithms for the 8-queens problem at 8queens.ipynb. **It also includes a translation to Python of the main routines.**
- R&N's On-Line Code Repository contains code for the n-queens problem written in a variety of languages. See the linked github repository, which includes Python, Java, ...

Further Notes:

- PS5 will also use the 8-queens problem in an effort to amortize your time and effort
- Also, there is a linked EXTRA CREDIT OPPORTUNITY involving the 8-queens problem

EECS 649: PROBLEM SET #5

Reading:

- R&N 5.1-7

Total Points: 100

Format:

- Use standard sheets of paper (8.5 by 11 inches) for scanning
- Perform all work neatly. When asked to write a paragraph, it should be typed (e.g., using a computer and LaTeX or Word).

---

*The answer to the following problem should be **typed**:*

Problem 5.1 [10 points] *Describe all the elements of solving a game*
Describe the state descriptions, move generators, terminal tests, utility functions, and evaluation functions for **only one** of the following stochastic games: Monopoly, Scrabble, bridge play with a given contract, **OR** Texas hold'em poker.

> *The following two problems require hand calculation and are much faster if you do **not** typeset them (unless you are using LaTeX …):*

Problem 5.2 [20 points] *Minimax vs. Alpha-Beta*
[Rich and Knight] Print out this linked game tree (twice).

a. Fill in the minimax values on one copy. Beneath the tree, note what move Player 1 should make at Node A and the minimum payoff the player is assured if they make that move.
b. Perform alpha-beta search on the second copy. Then, below the tree, list the nodes that would not have been expanded if alpha-beta pruning had been used (assuming nodes are expanded in **LEFT TO RIGHT ORDER**).

Example Solutions for (b): Here is a linked handout with two alpha-beta search examples on the first page (which model solutions for you); plus a detailed walkthrough of the algorithm on the second page (if needed to understand it; your solutions do not have to include that detail).

Problem 5.3 [30 points] *Basic concepts of game playing (using tic-tac-toe)*
[R&N] This problem explores some basic game-playing concepts, using tic-tac-toe (aka noughts and crosses) as an example. We define X_n as the number of rows, columns, or diagonals with exactly *n* X's and no O's. Similarly, O_n is the number of rows, columns, or diagonals with just *n* O's. The utility function assigns +1 to any position with $X_3 = 1$ and −1 to any position with $O_3 = 1$.

All other terminal positions have utility 0. For nonterminal positions, we use a linear evaluation function defined as Eval (s) = $3X_2(s) + X_1(s) - (3O_2(s) + O_1(s))$.

a. Approximately how many possible games of tic-tac-toe are there?
b. Show the whole game tree starting from an empty board down to depth 2 (i.e., one X and one O on the board), taking symmetry into account.
c. Mark on your tree the evaluations of all the positions at depth 2.
d. Using the minimax algorithm, mark on your tree the backed-up values for the positions at depths 1 and 0, and use those values to choose the best starting move.
e. Circle the nodes at depth 2 that would not be evaluated if alpha–beta pruning were applied, assuming the nodes are generated in the optimal order for alpha–beta pruning.

Notes:

a. *The s in Eval(s) refers to the current state(or configuration). You only have to approximate/bound the number of games.*
b. *Please take symmetry into account, as noted. Thus, instead of four initial boards with an X in each corner, you only have to deal with one in the upper left corner, e.g., because of the rotational symmetry.*
c. *In part (e), "optimal order" would be the one that leads to the most pruning after looking at a single node, typically this would be expanding the least value first at a min node and the largest value first for a max node.*

*The following problem requires **programming**:*

Problem 5.4 [40 points] *Min-conflicts aka 8-queens Revisited*
Consider the **8-queens** problem from R&N again. (Hopefully, this will drastically amortize the time you've spent on it last week.)

Solve it using

- Min-conflicts, choosing the variable (column) to minimize over at random
- Min-conflicts, choosing the variable to minimize in cyclic order: 1, 2, ..., 8, 1, 2, ..., 8, ...

Note: this is really the same algorithm, with just a slight difference in the way the variable to "de-conflict" is picked.

Again, you will be **maximizing** "fitness," defined as the number of **non-attacking** pairs of queens, which is 28 minus "the number of pairs of queens that are attacking each other, either directly or indirectly". Thus, when you find a configuration/state with fitness equals 28, you have found a solution.

Note: You can set a maximum number of iterations (report that number) and keep track of the number/percent of those that reach it bc they are stuck in a local min ("failures").

Also, if more than one value minimizes the conflicts, choose among them at random.

a. Implement **one program** encompassing both algorithms above (the difference between them is very slight; use a #define, global variable, command-line option, **or** user input to switch between them). In each case, the algorithm should exit as soon as a solution is found, print the solution (as a string of digits like those depicted in Figure 4.6 of R&N), and print the total number of **fitness evaluations** required from the start of the algorithm.

b. Test your program enough to convince yourself that it works and that solutions are being found appropriately. **(Do not report on this.)**

c. Gather statistics regarding the operation of the two different algorithms, recording their performance. Here, you are to run each algorithm 100 times (from different random starting positions/populations) and report **only the average** of the number of evaluations until a solution is found.

Freely use the code you wrote in Problem Set 4 or the C++ or Python I supplied for it in the linked notebook from PS4.

Hints: Pseudo-code appears in Figure 6.9 (p. 198) of R&N. Note that Min-Conflicts is similar to hill-climbing, except you only test 8 successors, not 56, so you can look at my supplied code for trying 56 successors and remembering the best.

Turn in your **code** and a **table** summarizing your results. How did this **compare** with your results from Problem Set 4?

EECS 649: PROBLEM SET #6

Reading:

- R&N 7.1-6; 7.7.1, 7.7.3

Total Points: 100

Format:

- Use standard sheets of paper (8.5 by 11 inches) for scanning
- Perform all work neatly. When asked to write a paragraph, it should be typed (e.g., using a computer and LaTeX or Word).

Notes:

- **There is no need to typeset any of your answers for PS#6**
- **Problem 6.6 only requires minimax and alpha-beta**
- **Problems 6.1 through 6.4 require only material from the lecture "Knowledge Representation and Predicate Logic"**
- **Problem 6.5 requires lecture on "Automated Reasoning and Logical Agents"**

Problem 6.1 [10 points] *Wumpus World*
Suppose the Wumpus Wolrd agent has progressed to the point shown in Figure 7.4(a), having perceived nothing in [1,1], a breeze in [2,1], and a stench in [1,2], and is now concerned with the contents of [1,3], [2,2], and [3,1]. Each of these can contain a pit, and at most one can contain a wumpus. Following the example of Figure 7.5, construct the set of possible worlds. (You should find 32 of them.) Mark the worlds in which the KB is true and those in which each of the following sentences is true:
α_2 = “There is no pit in [2,2].”
α_3 = “There is a wumpus in [1,3].”
Hence show that KB |= α_2 and KB |= α_3.

If you like, you can use these linked grids, kindly provided by former student Joseph Zarycki.
Note that the Wumpus can be in the same square as a pit; also, remember that there may be more than one pit.

Problem 6.2 [10 points] *Four Propositions*
Consider a vocabulary with just four propositions: A, B, C, D. How many models are there for each of the following sentences?
a. $B \vee C$.
b. $\neg A \vee \neg B \vee \neg C \vee \neg D$.
c. $(A \Rightarrow B) \wedge A \wedge \neg B \wedge C \wedge D$.
Find the number of models for which each sentence <u>is true.</u>

Problem 6.3 [10 points] *Verifying Equivalences*
Using a method of your choice (e.g., a truth table), verify the equivalences for (a) contraposition and (b,c) De Morgan's two laws in Figure 7.11

Problem 6.4 [15 points] *Valid, Unsatisfiable, or Neither*
Decide whether each of the following sentences is valid, unsatisfiable, or neither. Verify your decisions using truth tables or the equivalence rules of Figure 7.11.
a. Smoke ⇒ Smoke
b. Smoke ⇒ Fire
c. (Smoke ⇒ Fire) ⇒ (¬Smoke ⇒ ¬Fire)
d. Smoke ∨ Fire ∨ ¬Fire
e. ((Smoke ∧ Heat) ⇒ Fire) ⇔ ((Smoke ⇒ Fire) ∨ (Heat ⇒ Fire))
f. (Smoke ⇒ Fire) ⇒ ((Smoke ∧ Heat) ⇒ Fire)

Problem 6.5 [20 points] *Forward Chaining and DPLL*
Trace the behavior of DPLL on the knowledge base in Figure 7.16 when trying to prove Q, and compare this behavior with that of the forward-chaining algorithm.

Hints:

1. **We did forward chaining (and backward chaining) in class;**
2. **For DPLL, put KB in CNF -- see the caption of Figure 7.12 for the pattern: A ^ B => C becomes ~A v ~B v C;**
3. **Wikipedia's DPPL entry may be easier to understand than R&N's description:**
   https://en.wikipedia.org/wiki/DPLL_algorithm#The_algorithm
   There is also an example (using ' for negation and + for or).

The last problem requires programming to solve a non-trivial game: 4x4 Tic-Tac-Toe.
Specifically, it will involve modifying a supplied Python program.
Hence, the problem is easiest if you use Python.
It is especially straightforward if you use Google Colaboratory, as explained next.
To use Google Colaboratory, you need a Google account (create if you don't have one).
Then:

- https://drive.google.com THEN NAVIGATE +New, More >, Google Colaboratory (sometimes you have to add this as an extension if not available)
- MORE INFO AT https://colab.research.google.com/notebooks/welcome.ipynb

Problem 6.6 [35 points] *Solving games in practice*
In this linked Tic-Tac-Toe Python notebook, we provide programs for both minimax and alpha-beta search to solve Tic-Tac-Toe (on a 3x3 board).

a. Start with the program given or translate it to another language of your choice. Test it on the examples from the lecture notes to see if you get the same results. **Do not turn anything in.**

b. Copy the program into a file called TicTacToe4. Modify it to play 4x4 Tic-Tac-Toe (where you win with 4 marks across any row, column, or main diagonals). For ease in debugging and reporting, modify the printing function as well. For reference, this took me 10-15 minutes, noting patterns and applying cut-and-paste liberally. Note that there are a couple tricky places that I have marked with "**`BEWARE`**".

   Do not turn in any code. But comment at a high level on what you modified, how long it took, etc.

c. Run minimax on the ten test boards at the end of this problem set (as printed by my modified program), **as your available computer memory and time allow**.

   Notes:

   - The boards are arranged in order of time/space complexity and all **may not be solvable** in reasonable time/memory on your (or Google's) machines. Hence, you **do not need to solve every board to receive full credit**. But comment.
   - You may need or want to cutoff search at some point, say at 100 million nodes (YMMV) expanded for time or if the `seen_set` gets too large (or maxes out your RAM). This can be done with if statements in the `new_node` function.
   - To check your program: note that Board 1 is an immediate win, Board 2 is an immediate loss; Boards 3-6 are wins.

   Report your results (please delete any intermediate results printed!), including the games value, optimal move (minimax only), nodes examined, terminals examined, unique states encountered, and time taken.

d. Repeat the above using alpha-beta.

e. **Compare** your minimax and alpha-beta results. How much more efficient was it in terms of nodes examined (etc.) and time? **Did you weakly solve** 4x4 Tic-Tac-Toe? What is the **full game's value** for the first player ('X')?

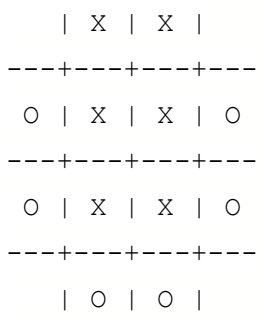
Optional, Extra Credit Problem [20 points]

Use transposition tables (aka cache-ing or memo-ization) to speed either of minimax or alpha-beta. Report and interpret your results.

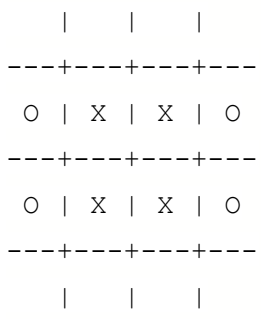
TEST BOARDS FOR 4x4 TIC-TAC-TOE

```
BOARD 1
 X | X | X | X
---+---+---+---
 O | O | O |
---+---+---+---
   |   |   |
---+---+---+---
   |   |   |

BOARD 2
 O | O | O | O
---+---+---+---
 X | X | X |
---+---+---+---
   |   |   |
---+---+---+---
   |   |   |

BOARD 3
 X | X | X |
---+---+---+---
 O | X | X | O
---+---+---+---
 O | X | X | O
---+---+---+---
   | O | O | O
```

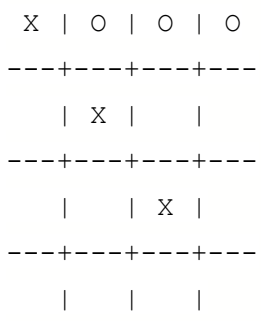
BOARD 4



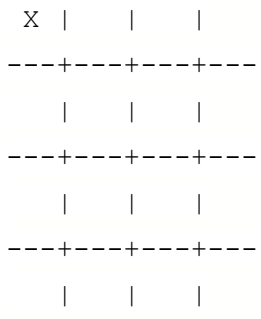
BOARD 5



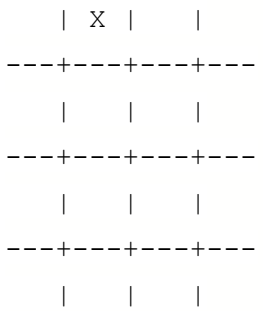
BOARD 6



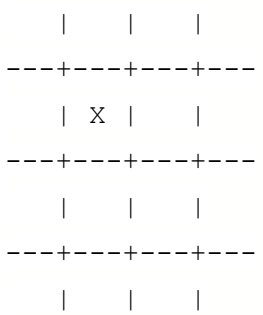
BOARD 7



BOARD 8 (aka Elon Musk’s favorite)



BOARD 9



BOARD 10