1 SATurdays are for everyone

1. Determine whether the sentences below are satisfiable or unsatisfiable (using any method you like).

   (a) $(\neg(y \vee \neg y) \vee x) \wedge (x \vee (z \iff \neg z))$

   Satisfiable

   (b) $\neg(x \vee \neg(x \wedge (z \vee T))) \implies \neg(y \wedge (\neg y \vee (\top \implies \bot)))$

   Satisfiable

   (c) $((\top \iff \neg(x \vee \neg x)) \vee z) \vee z) \wedge \neg(z \wedge ((z \wedge \neg z) \implies x))$

   Unsatisfiable

2. Suppose $A \models B$. Which of the following statements must be true for all truth assignments to A and B?

   (b)

   (a) $A \wedge B$
   (b) $A \Rightarrow B$
   (c) $B \Rightarrow A$
   (d) $A \vee B$
   (e) B

3. How would we formulate the SAT problem as a CSP? What are the variables? Domains? Constraints?

   SAT can be modeled as a CSP in which the variables are literals with domain $\{\top, \bot\}$, and the constraints are the clauses themselves.

4. Suppose we have an algorithm which determines whether a sentence is satisfiable or not. Given two sentences A, B, how could we determine whether $A \models B$?

   If $A \models B$, all models which satisfy A also satisfy B.

5. Determine whether the sentence below is satisfiable or unsatisfiable using DPLL. Break ties by assigning variables in alphabetical order, starting with **false**. If satisfiable, what model does the algorithm find?

$$(A \vee B) \wedge (B \vee C \vee D) \wedge (\neg A \vee \neg B \vee C) \wedge (\neg A \vee \neg C \vee \neg D) \wedge A \wedge (C \vee \neg D)$$

   Assign $\top$ to A (unit clause): $(B \vee C \vee D) \wedge (\neg B \vee C) \wedge (\neg C \vee \neg D) \wedge (C \vee \neg D)$
   Assign $\bot$ to B: $(C \vee D) \wedge (\neg C \vee \neg D) \wedge (C \vee \neg D)$
   Assign $\bot$ to C: $(D) \wedge (\neg D)$
   Assign $\top$ to D (unit clause - could also be $\bot$): $\neg D$ evaluates to false, so we have to backtrack.

   Assign $\top$ to C: $(\neg D)$
   Assign $\bot$ to D (unit clause)! This sentence is satisfiable; the model found is $A : \top, B : \bot, C : \top, D : \bot$.

2 All About Logic

1. Propositional Logic

   (a) Vocab check: are you familiar with the following terms?

      i. Symbols

      Variables that can be T/F (capital letter)

      ii. Operators

      and, or, not, implies, equivalent

      iii. Sentences

      Symbols connected with operators, can be T/F

      iv. Equivalence

      True in all models that a and b implies each other (a equivalent to b)

      v. Literals

      atomic sentence

      vi. Knowledge Base

      Sentences agents know to be true

      vii. Entailment

      a entails b iff $\forall$ models, a true implies b true

      viii. Query

      A sentence we want to know whether it's true (usually we want to know whether KB entails q)

      ix. Satisfiable

      At least one model makes the sentence true

      x. Valid

      True for all models

      xi. Clause - Definite, Horn clauses

      Clause - disjunction of literals; definite - clause with exactly one positive literal, horn - clause with at most one positive literal

      xii. Model Checking

      check if sentences are true in given model/checks entailment

      xiii. Theorem Proving

      Search for a sequence of proof steps. (e.g. Forward Chaining)

      xiv. Modus Ponens

      From $P, (P \longrightarrow Q)$, infer Q

2. **Chipotle?**

   (a) Sean is a student in class. In his knowledge base, he admits that if he doesnt get bored, he will pass the class no matter what he does later on. However, if he gets bored and he ends up going directly to Chipotle, he believes he can always pass the class after getting food. He doesn't really use Facebook. Represent Sean's knowledge base in propositional logic. Let B be the symbol representing if Sean gets bored, C representing going to Chipotle, F representing using Facebook, and P representing passing the class. (Select all that can represent his Knowledge Base)

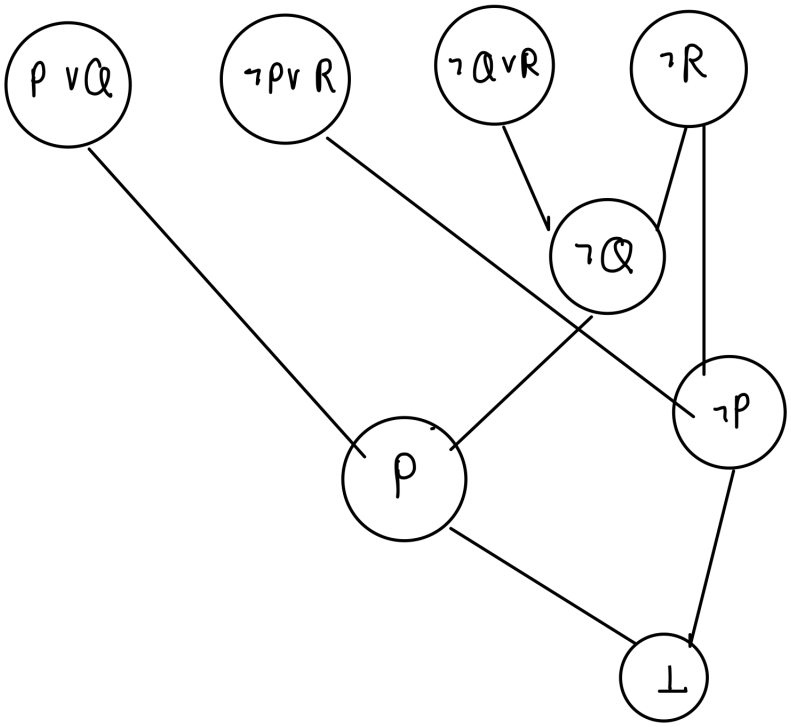
i, iv

i. $\neg B \Rightarrow P; B \wedge C \Rightarrow P$
ii. $\neg B \Rightarrow P; \neg P \Rightarrow \neg C \vee F$
iii. $B \vee P; (\neg B \wedge \neg C \wedge F) \vee P$
iv. $B \vee P; (\neg B \vee \neg C) \vee P$

3. Given the following propositional logic clauses, show R must be true and using only the resolution inference rule to derive a contradiction. Your answer should be in the form of a graph, where each resolvent is connected by lines to its two parent clauses. Use the clauses below as the initial set of nodes in the graph.

   Note: You do not need to use all the nodes, and you may use a node more than once.

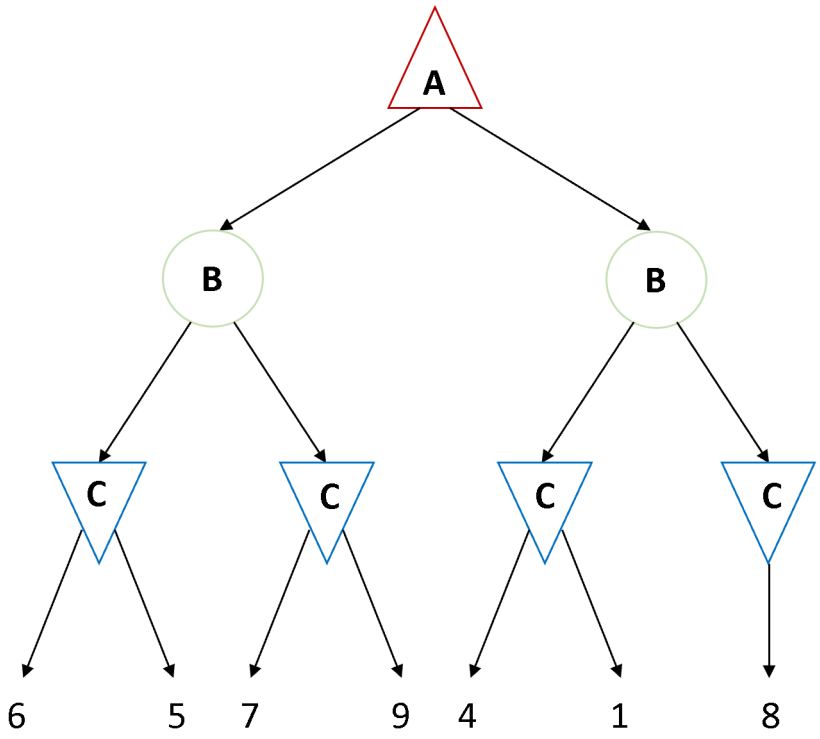
$P \vee Q$ $\neg P \vee R$ $\neg Q \vee R$



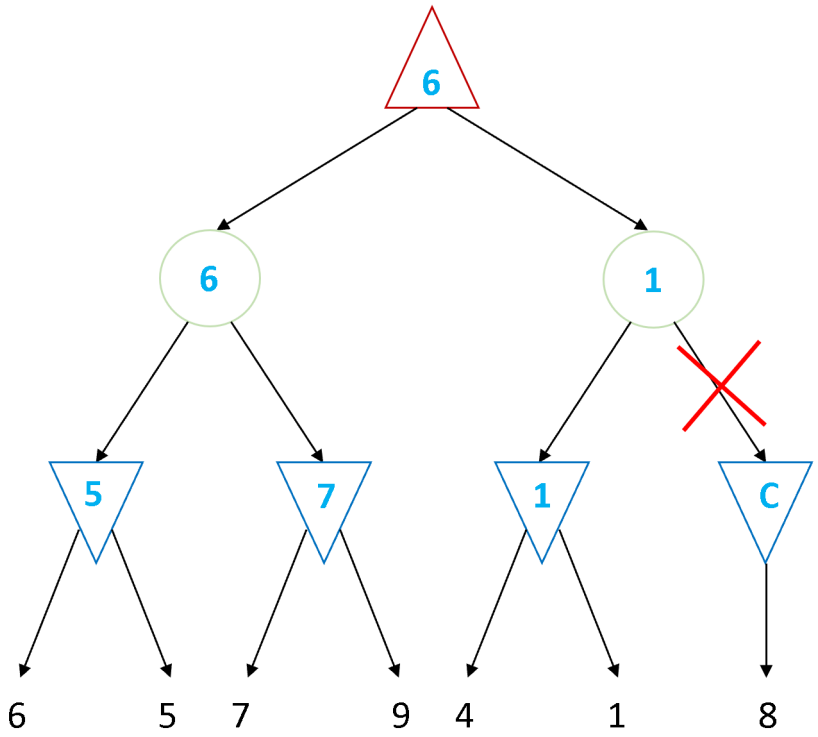
3 Adversarial Search (Minimax+Expectimax Pruning)

So we've done pruning on minimax trees, but what happens when we introduce chance nodes? Recall that a chance node has the expected value of its children, and let each child have an equal probability of being chosen.

Perform pruning on the following game tree and fill in the values at letter nodes. A is a maximizer, B is a chance node, and C is a minimizer. Assume that values can only be in the range 0-9 (inclusive).



Solution:

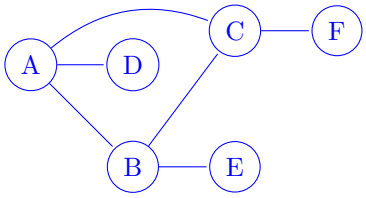


4 CSPs

You've generously saved a row of 6 seats in Rashid for 6 of your 15-281 classmates (A-F[1]), and are now trying to figure out where each person will be seated. You know the following pairs of people have some kind of binary constraint between them:

- A, B
- A, C
- A, D
- B, C
- B, E
- C, F

(a) Draw the constraint graph to represent this CSP.



(b) Some value is assigned to A. Which domains could change as a result of running forward checking for A?

B, C, D. Forward checking for A only considers arcs where A is the head. This includes $B \to A, C \to A, D \to A$. Enforcing these arcs can change the domains of the tails.

(c) Some value is assigned to A, and then forward checking is run for A. Then some value is assigned to B. Which domains could change as a result of running forward checking for B?

C, E. Similar to the previous part, forward checking for B enforces the arcs $A \to B, C \to B, E \to B$. However, because A has been assigned, and a value is assigned to B, which is consistent wit hA or else no value would have been assigned, the domain of A will not change.

(d) Some value is assigned to A. Which domains could change as a result of running AC-3 after this assignment?

B, C, D, E, F. Enforcing arc consistency can affect any unassigned variable in the graph that has a path to the assigned variable. This is because a change to the domain of X results in enforcing all arcs where X is the head, so changes propagate through the graph. Note that the only time in which the domain for A changes is if any domain becomes empty, in which case the arc consistency algorithm usually returns immediately and backtracking is required, so it does not really make sense to consider new domains in this case.

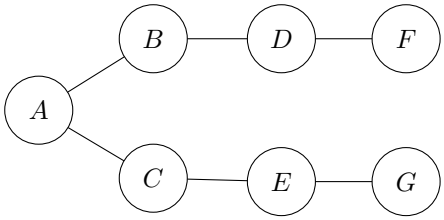
(e) Some value is assigned to A, and then arc consistency is enforced via AC-3. Then some value is assigned to B. Which domains will and will not change as a result of enforcing arc consistency after B's assignment?

After assigning a value to A, and enforcing arc consistency, future assignments and enforcing arc consistency will not result in a change to A's domain. This means that D's domain won't change because the only arc that might cause a change, $D \to A$ will never be enforced. However, the domains of C, E and F do change.

[1] not correlated with their grades

(f) You are now trying a brand new algorithm to solving CSPs by enforcing arc consistency via AC-3 initially, then **after every even-numbered assignment** of variables (after assigning 2 variables, then after 4, etc.).

You have to backtrack if, after assigning a value to variable X, there are no constraint-satisfying solutions. Mathematically, for a single variable with d values remaining, it is possible to backtrack $d-1$ times in the worst case. For the following constraint graph, assume each variable has domain of size d. How many times would you have to backtrack in the worst case for the specified orderings of assignments?



i. ABCDEFG:

0

ii. GECABDF:

0

iii. GFEDCBA:

$3(d-1)$

Note that the given constraint graph represents a tree-structured CSP. Therefore (as seen in lecture), any order of assignments which assigns variables from root to leaves in some topological ordering of the nodes guarantees that we do not have to backtrack given that root-to-leaf arcs are consistent.

ABCDEFG and GECABDF are both orderings which satisfy the above description, which means we essentially initially enforce arc consistency, then assign values to nodes from root to leaves.

GFEDBCA is not such an ordering, so while the odd assignments are guaranteed to be part of a valid solution, the even assignments are not (because arc consistency was not enforced after assigning the odd variables). This means that you may have to backtrack on every even assignment, specifically F, D, and B. Note that because you know whether or not the assignment to F is valid immediately after assigning it, the backtracking behavior is not nested (meaning you backtrack on F up to $d-1$ times without assigning further variables - otherwise, we would have a worst-case number of backtracks of n^3). The same is true for D and B, so the overall behavior is backtracking $3(d-1)$ times.

(g) Recall the CSP from part (a). The actual constraints are as follows:
Both C and E want to sit next to B.
A wants to sit next to D, but not next to B or C.
F and C had a falling out over whether AI or blockchain was cooler, so there needs to be at least 2 seats between them.

B gets to class first and sits down in seat 3. Run AC-3 to determine the final seating arrangement.

In order: F, E, B, C, D, A. There may be alternate solutions.

5 Search (Algorithms & Properties)

(a) When can we guarantee completeness and optimality (if ever) for each of the following search algorithms we've seen in class? For each algorithm, indicate under what conditions it is complete and/or optimal.

Algorithm	**Complete**	**Optimal**
Breadth-First		
Depth-First		
Iterative Deepening		
A*		

Algorithm	**Complete**	**Optimal**
Breadth-First	Always	When all edge costs are equal and non-negative
Depth-First	Never	Never
Iterative Deepening	Always	When all edge costs are equal and non-negative
A*	Always	When an admissible (trees) or consistent (graphs) heuristic is used

(b) Consider a dynamic A* search where after running A* graph search and finding an optimal path from start to goal (assuming there's only one goal), the cost of one of the edges $X \to Y$ in the graph changes. Instead of re-running the entire search, you want to find a more efficient way of

returning the optimal path for this new search problem.

For each of the following changes, describe how the optimal path cost would change (if at all). If the optimal path itself changes, describe how to find the new optimal path. Denote c as the original cost of $X \to Y$, and assume $n > 0$.

i. c is increased by n, $X \to Y$ is on the optimal path, and X was explored by the initial search.

The optimal path could change if the original cost is no longer the cheapest when adding the amount of n.
We would re-explore all previously expanded nodes with the new edge cost of X → Y. If Y is explored, we end the search for all paths that will end at Y in the previous search add the goal node together with the previous optimal path cost of Y → Goal to the frontier, then we keep searching down until we pop the goal node out of the frontier. This means that you are re-exploring every path that was originally blocked by a path that included the edge X → Y. If the cheapest path to Y is discovered, then we already know the optimal path and path cost of $Y \to$ Goal

ii. c is decreased by n, $X \to Y$ is on the optimal path, and X was explored by the initial search.

The original optimal paths cost decreases by n because $X \to Y$ is on the original optimal path. The cost of any other path in the graph will decrease by at most n (either n or 0 depending on whether or not it includes X Y). Because the optimal path was already cheaper than any other path, and decreased by at least as much as any other path, it must still be cheaper than any other path.

iii. c is increased by n, $X \to Y$ is not on the optimal path, and X was explored by the initial search.

The cost of the original optimal path, which is lower than the cost of any other path, stays the same, while the cost of any other path either stays the same or increases. Thus, the original optimal path is still optimal.

iv. c is decreased by n, $X \to Y$ is not on the optimal path, and X was explored by the initial search.

We would put previously expanded node (with their path and path cost) on the frontier and search for X, as well as the optimal path. If X is expanded this time, we clear all paths on the frontier except for the original optimal path and the cheapest path leading to X plus the edge $X \to Y$. Then we do A* search with this new frontier.

There are two possible paths in this case. The first is the original optimal path, which is considered by adding the previous goal node back onto the frontier. The other option is the cheapest path that includes $X \to Y$, because that is the only cost that has changed. There is no guarantee that the node ending at Y , and thus the subtree rooted at Y contains $X \to Y$, so the optimal path leading to X must be found in order to find the cheapest path through X → Y .

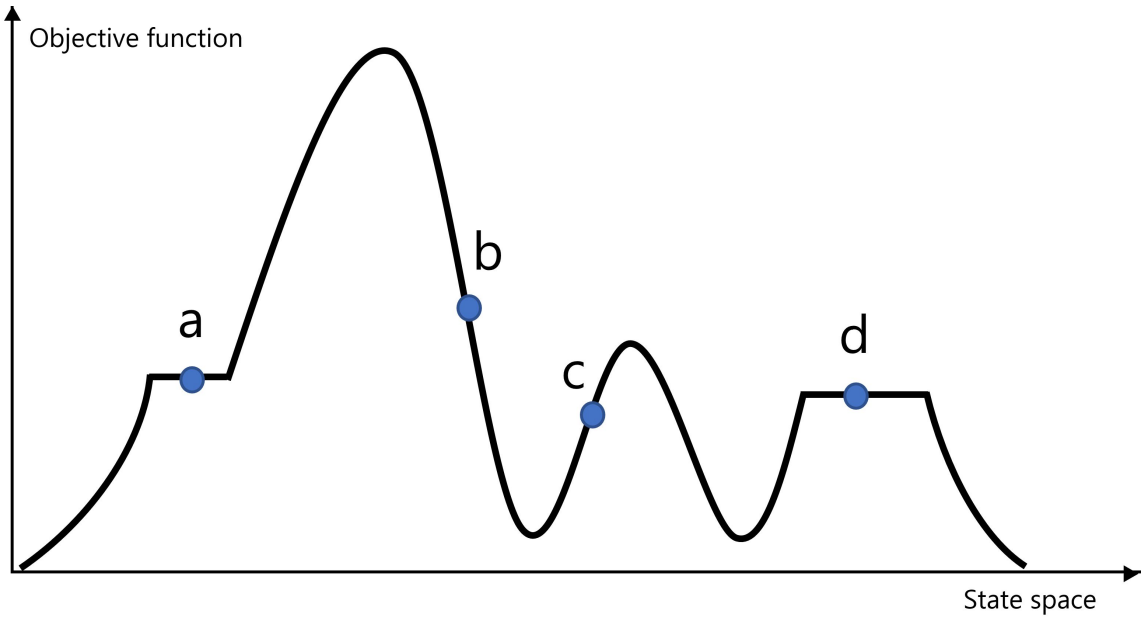
v. c is increased by n, $X \to Y$ is not on the optimal path, and was not explored by the initial search.

This is the same as part (c).

vi. c is decreased by n, $X \to Y$ is not on the optimal path, and was not explored by the initial search (assuming the edge weights c can't go negative.)

Assuming that the cost of $X \to Y$ remains non-negative, because the edge was never explored, the cost of the path to X is already higher than the cost of the optimal path. Thus, the cost of the path to Y through X can only be higher, so the optimal path remains the same.

6 Local Search



Consider how each of the following searches performs in state space above. Recall that in the context of local search, our goal is to find the state that optimizes the objective function.

(a) Hill-climbing search with start state c

Does it terminate? If so, where?

Does it find the global maximum?

It terminates at the local maximum (to the right of c). It is not complete.

(b) Random-restart hill climbing with randomly generated initial states a, d, and b

Does it terminate? If so, where?

Does it find the global maximum?

Random-restart hill climbing conducts a series of searches. For start states a and d, the search will terminate immediately since there are no better neighboring states. For start state b, the search will reach the global maximum, so it is complete. In general, random-restart hill climbing is "trivially complete with probability approaching 1, because it will eventually generate a goal state as the initial state." (AIMA Chapter 4, page 124)

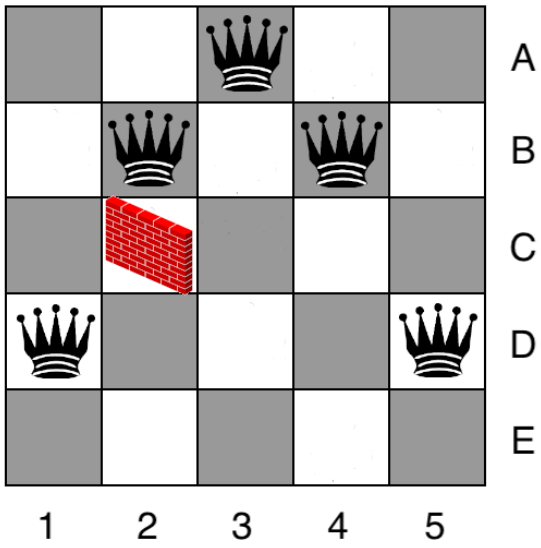
(c) Stochastic hill-climbing that allows sideways moves with start state d

Does it terminate? If so, where?

Does it find the global maximum?

This search will not terminate. Since we allow sideways movement and d is on a flat local maximum, stochastic hill-climbing will choose randomly from one of its neighboring states with the same objective function value. This will result in an endless loop. It is not complete.

Now, let's revisit the 5-QUEENS problem. Our goal is to try to place 5 queens on a 5x5 chessboard with no conflict between any two queens. However, this time, there is a twist: our chessboard has a wall!



If the wall blocks the path between two queens (horizontal, vertical, or diagonal) then those two queens will not be in conflict. The wall cannot be moved, and a queen cannot pass through the wall.

Below is some pseudocode to try to solve the 5-QUEENS problem with hill climbing.

```
function 5-QUEENS-HILL-CLIMBING(problem)
    current_state = problem.INITIAL-STATE
    loop do
        neighbor = state with lowest # of conflicts between queens obtained by
                   making one legal chess move from current_state
        if neighbor.num_conflicts >= current_state.num_conflicts
            return current_state
        current_state = neighbor
```

Using the given starting state, pseudocode, and the rules regarding walls, answer the following questions.

(a) Run 5-QUEENS-HILL-CLIMBING on the given start state until completion. Where do the queens end up? Is this a goal state?

The queens will end up at D1, B2, E3, A4, and D5. This is not a goal state because there is a conflict between D1 and D5.
Our starting state has 4 conflicts. If we move A3 to E3, we see that the new state only has 2 conflicts, which is the lowest number of conflicts among all the neighbors. From here, we can move B4 to A4 to reduce the number of conflicts to 1. This is no conflict between this new position and D1 due to the brick wall. From here, we cannot take any steps to reduce the number of conflicts.

(b) If we remove the brick wall, will 5-QUEENS-HILL-CLIMBING end up in a global optima?

If we remove the brick wall, 5-QUEENS-HILL-CLIMBING will not end up in a global optima. It will be at a state whose value is not less than its neighbors and is not a global optima. The state and all its neighboring states will have 2 or more conflicts.