CS156 Final Fall 2014 Version 1

Name:	
StudID:	

Problem	Grade	Problem	Grade
1	3	6	3
2	3	7	3
3	2.5	8	3
4	3	9	3
5	2.5	10	3

29

1. Briefly explain the AC-3 algorithm (1.5pts), and show how it might be applied to a particular CSP (1pt).

```
def AC-3(problem):
    arc_constraints = []
    # store all binary constraints
    for (xi, xj) in problem.BinaryConstraints():
        arc_constraints.append((xi, xj))
    # Keep looping until all binary constraints satisfied
    while (len(arc_constraints) > 0):
        (xi, xj) = arc_constraints.pop()
        # Check if domain xi needs to be reduced
        if (Revise(csp, xi, xj)):
            # Check if domain xi is empty
            if (len(xi.Domain)):
                return false
            # xi revised so recheck all its neighbors
            for xk in xi.Neighbors - {xj}:
                if ((xi, xk) Not in arc_constraints):
                    arc_constraints.append((xi, xk))
    return true
# check domain of xi
def Revised(problem, xi, xj):
    revised = False
    for d_i in xi.Domain():
        if (no value in xj.Domain can satisfy constraint (xi, xj) for d_i):
            xi.Domain.Delete(d_i)
            revised = True
    return revised.
```

Example See Back

3

	A	B	C	D
1	4			
2				3
3	2			
4				1

← 2x2 sudoku, By just applying arc consistency you can formulate the all-diff constraint. Example Alldiff(A_1, A_2, A_3, A_4) as binary constraints is $A_1 \neq A_2$, $A_1 \neq A_3$, $A_1 \neq A_4$, $A_2 \neq A_3$, $A_2 \neq A_4$, $A_3 \neq 4$, Hence, By using AC-3 to do forward checking in constraint satisfaction you can solve the above board with no backtracking, just by limiting each squares domain to a single possible value. This is done in AC-3 by iterating over pairs of cell to eliminate domain value,

4	3	1	2
1	2	4	3
2	1	3	4
3	4	2	1

Solved via AC3 on binary (arc) constraints.

2. (a) Let $x := f(z)$ and $y := g(w)$ explain how the unification algorithm from class would work on these inputs. (1pt) (b) Now suppose $x := [g(v), f(g(z))]$ and $y := [g(f(w)), f(w)]$. Explain how the unification algorithm from class would work on these inputs. (1.5pts)

a) Unify("f(z)", "g(w)", {})

Parameters are a term so separate args and operator

Unify("z", "w", Unify("f", "g", {}))

Fails since can't unify the two operators cannot be unified

Failed

b) Unify([[g(v), f(g(z))], [g(f(w)), f(w)], {})

Two lists so seperate lists by unifying heads of the list. I will focus on this first.

- Unify("g(v)", g(f(w))", {}) # separate op and args
- Unify("v", "f(w)", Unify("g", "g", {}))# ops same so unify
- Unify("v", "f(w)", {})# v is var so unify var on it
- Unify_var("v", "f(w)", {})

v is not in f(w) and not in the substitution list so it is added to substitution list and plugged in modified version of the original unify. These were lists but I will simplify here and treat as non list since just 1 element each

Unify("f(g(z))", "f(w)", {v ↦ f(w)})

Remove "f" operator

Unify("g(z)", "w", Unify("f", "f", {v ↦ f(w)}))

SEE BACK

3

3. Consider the problem where you have three plates on a table (unstacked) and your goal is to have a single stack of the three plates on the table. Model this problem using PDDL (1pt). Then solve it using the GraphPlan algorithm from class (1.5pt).

~~X plate ... ex ...~~

Three plates P_1, P_2, P_3

Table is a constant for Table

Plate - Predicate to check if plate.

Move(plate, x, y) - moves plate from on top of x to on top of plate y.

On(x, y) - x (plate) is on top of y (other plate or table)

Nothing on top(y) - Means nothing is on top of y (y is plate or table)

Init(Plate(P_1) ∧ Plate(P_2) ∧ Plate(P_3) ∧ On(P_1, Table) ∧ On(P_2, Table) ∧ On(P_3, Table))

need to specify

Goal (Plate(X) ∧ Plate(Y) ∧ Plate(Z) ∧ On(X, Y) ∧ On(Y, Z) ∧ On(Z, Table) ∧ (X ≠ Y) ∧ (Y ≠ Z))

ensures not the same plate

Action(Move(P, x, y)

precond: Plate(p) ∧ On(p, x) ∧ Nothing onTop(p) ∧ Nothing onTop(y)

effect: ¬On(p, x) ∧ On(p, y)) ∧ ¬NothingonTop(y)

Example Plan:

Move(P_2, Table, P_3)

Move(P_1, Table, P_2)

2.5

2B) Unify("g(z)", "ω", Unify("f", "f", {v ↦ f(ω)}))

f is the same so they unify.

Unify("g(z)", "ω", {v ↦ f(ω)})

ω is a variable so it is run through unify-var

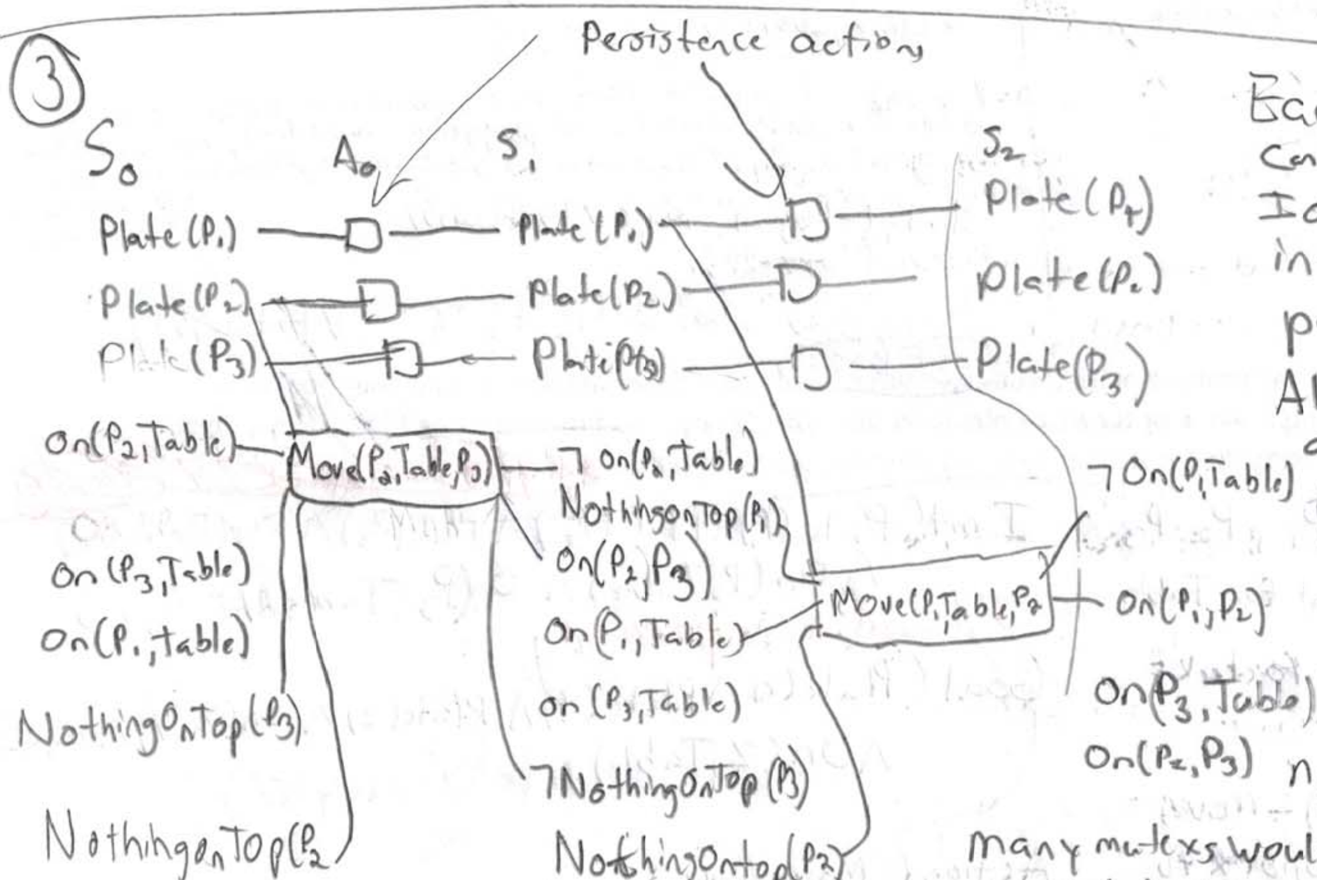
Unify-var("ω", "g(z)", {v ↦ f(ω)})

ω ↦ g(z) added to substitution list making final substitution list

$\boxed{\{v \mapsto f(\omega),\ \omega \mapsto g(z)\}}$

---

3)

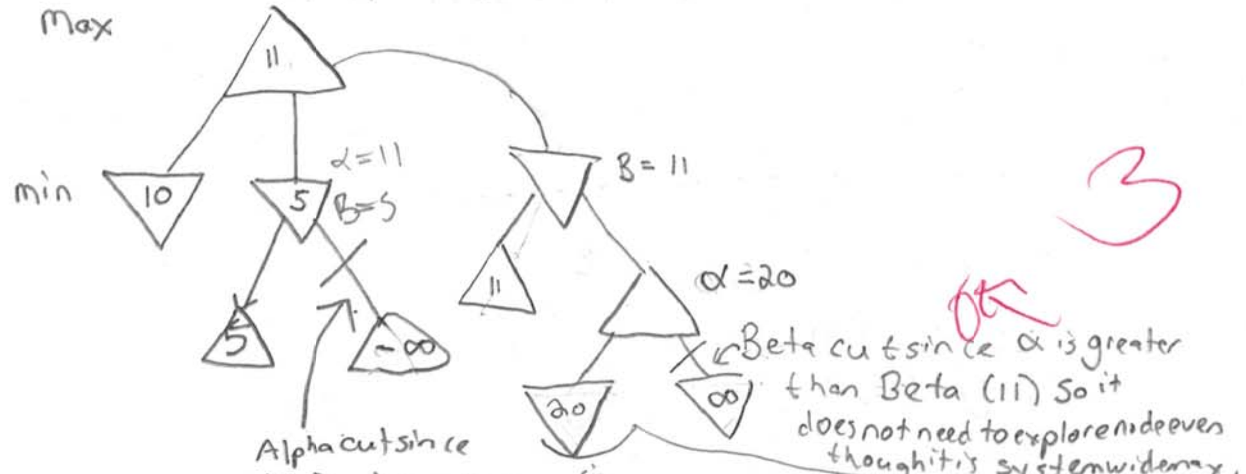


Each plate can move in a turn. I only show actions in the plan on last page for simplicity. Also persistence actions for "on" literals not shown for enhanced readability. Also for all not shown actions many mutexs would be needed but are not shown.

Graph plan would start in A_0, Goal literals not found so it would expand A_0 and S_1. In S_1 not all goals not mutex and satisfy inequality condition so it expands A_1 and S_2. Goal literals present but not mutex so it runs "Extract Solution". I will discuss csp version of extract solution where the variables are Actions at each level. Domain is IN/out of plan and constraints are mutexs. It would then set "IN" for Move(P_1,Table, in A_1 and "IN" for Move(P_2,Table,P_3) and "out" for all

4. What is the minimax function? (1pt give the definition) Given an example of a situation in which a beta-cut might arise while running the mimimax algorithm with alpha-beta pruning. (1.5pts)

$$Minimax(s) = \begin{cases} Utility(s), & \text{if Terminal-Test(s) is true} \\ \max_{a \in ACTIONS(s)} \left(minimax(Result(s,a))\right) & \text{if (Player(s)) is max} \\ \min_{a \in ACTIONS(s)} minimax(Result(s,a)) & \text{if Player(S) is min.} \end{cases}$$



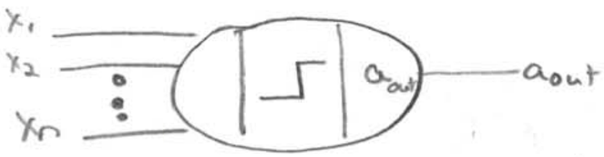
3

ok

Beta cut since α is greater than Beta (11) so it does not need to explore node even though it's system wide max.

Alpha cut since α(10) is greater than or equal to B(5). Top Max will not take this node's value since it is less than previous max.

5. Give (carefully using the definition of perceptron) a perceptron (1pt) that computes the AND of *n* input variables (1.5pts).



This perceptron has n inputs. It uses a standard threshold function as its activation function. Hence:

$$g(\vec{\omega}\vec{x}) = \begin{cases} 0, & \vec{\omega}\cdot\vec{x} < 0 \\ 1, & \text{otherwise} \end{cases}$$

2.5

Consider weight vector $\vec{\omega}$. For each input ω_i, where $i \geq 1$, that ω_i is equal to 1. Consider that its offset term ω_0 is equal to $-1\cdot(n-.5)$. Hence when all n inputs are 1, the resulting value of $\vec{\omega}\cdot\vec{x}$ is:

?

$$\vec{\omega}\cdot\vec{x} = \omega_0 + \omega_1 \cdot x_1 + \omega_2 \cdot x_2 + \ldots$$

$$\vec{\omega}\cdot\vec{x} \quad -(n-.5) + n\cdot 1 = .5$$

this could be clearer

(which corresponds to 1 on the output. For all other inputs (assuming binary inputs), the sum will be less than 0 making the output 0 exactly like an AND gate.

A perceptron is a node that takes a real valued input and maps it via a function to a real valued output. In a standard perceptron, this mapping function is the threshold function while in a sigmoid perceptron, the function is the logistic function $\frac{1}{1+e^{-t}}$.

6. For each of following reasoning frameworks, briefly define it and give an example: (a) description logic (b) circumscription (c) default logic.

Circumscription - Unless known otherwise, assume a predicate is as false as possible. vague It can be done by a circumscribed reasoner on a circumscribed predicate to achieve default logic.

Default logic - Unless you know the default not to be true, assume the default.

Example Using default logic and circumscription

Default
$$\frac{Person(x) : Awesome(x)}{Awesome(x)}$$

Default is to assume x (e.g. Zayd) is awesome unless you know otherwise. It could be rewritten with a circumscribed predicate Abnormal(x) to achieve a default value (Awesome).

Circumscription
$Person(x) \land \neg Abnormal(x) \Rightarrow Awesome(x)$

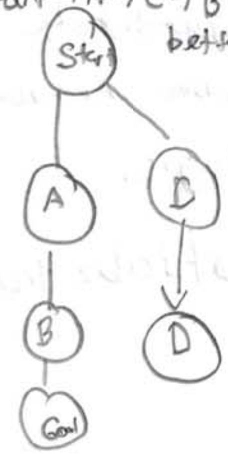
See Back

7. For each of the following algorithms say if it is complete and give its memory and time requirements: (a) breadth first search, (b) depth first search, (b) IDA^* (0.5 each). When and why might one prefer depth first search over breadth first search and vice versa? (1pt)

Depth first search is not complete. An example would be an infinite graph. DFS would continue down the infinite path and never return. DFS has space complexity is $O(d)$ where d is the depth from the start state. Assuming a finite branching factor, its time complexity is $O(b^d)$. Breadth first search is complete as it expands the frontier uniformly (assuming finite branching factor). Its time and space complexity are $O(b^d)$, where for DFS and BFS d is the depth from the start state and b is branching factor (maximum number of out nodes from any node). Iterative deepening A^* is complete ~~if~~. However the time and space complexity are dependent on the heuristic used and the problem space. For time complexity, the minimum number of steps (assuming perfect heuristic) is the minimum number of steps (edges) from the initial state to the goal.

See Back

DFS performs better here as it takes left most path first so it goes Start→A→B→Goal. BFS has uniform frontier so it goes Start→A→C→B→D→Goal making DFS bette



Whether BFS or DFS is better has to some extent to do with the graph as shown on the left. DFS will a lower asymptotic space complexity $O(d)$ than BFS ($O(b^d)$). However, DFS is not complete (unlike Iterative deepening DFS). DFS is also not optimal while BFS will be optimal if the edge weights are uniform. Since BFS expands uniform frontier it may be slower in terms of time complexity than DFS (see top left corner) in certain graph topologies.

Description logics are used to describe properties and definitions of categories. It can is some cases (e.g. CLASSIC) weaker than first order logic. Example to describe Category " "

AND(Has(Beard), HAs(Cat), One of (cool, awesome))

8. Give the hill climbing algorithm from class (1pt). Then discuss how it is modified when one does k-local beam search (0.5pts). Give an advantage and a disadvantage of the latter algorithm over the former (0.5pts each).

```
def Hill_Climbing (problem):
    Current_state = problem. Initial_STATE() # Extract initial state,
    # Keep looping until maximum(local or global) is reached,
    While (true):
        next_state = None
        # iterate through all possible actions and get one with maximum utility
        for a in ACTIONS(current_state):
            Successor = RESULT(current_state, a)
            if(next_state is None or Utility (next_state) < Utility(successor)
                next_state = Successor
        # If next_state is better update current state:
        if(Utility(next_state) > Utility (current_state)):
            Current_state = next_state
        else:
            return current_state
```

K local beam is a variant of hill climbing. Unlike hill climbing it starts with K state, It then picks the k best successors from across all k states and those form next round of K states, until a maximum is found.

K local beam has the disadvantage of requiring more memory (space) than standard hill climbing. K-local beam manages "k" states (assuming k > 1) while hill climbing tracks only one state. In contrast, K local beam traverses more of the search space than hill climbing since it has k successor states, meaning it is more likely to find a solution and not get trapped in a local maxima.

3

$A_1 \quad A_2 \quad A_3$

9. Consider the following training set of 4-tuples.

$S, \vec{x}$

(T,T,T,	F)
(T,T,F,	F)
(T,F,T,	F)
(T,F,F,	F)
(F,T,T,	T)

$\leftarrow Y$

Here T is short for true, F is short for false. The first three columns correspond to the variables x_1, x_2, x_3, the last column is the output of some function f. Calculate $Gain(x_i)$ for $i = 1,2,3$ (0.5pts each). Which variable should we use as the top of a decision tree for f? (1pt)

For attributes in a binary decision tree, the information gain for an attribute "A" is defined as:

$Gain(A) = B(\frac{p}{p+k}) - Remainder(A)$.

p - # positive examples (1 in this case)

n - # negative examples (4 in this case)

$Remaindet(A) = \sum_{V_k \in A} \frac{(p_k + n_k)}{(p+n)} \cdot B(\frac{p_k}{n_k + p_k})$

For A_1, A_2, A_3 (defined above) $B(\frac{p}{p+n})$ is

$B(\frac{1}{5}) = -\frac{1}{5}(\frac{1}{5} \cdot \lg(\frac{1}{5}) + (\frac{4}{5}) \lg(\frac{4}{5}))$ (call this value x)

$B(\frac{1}{5})$ is the same for all three attributes so it's like a constant and has no effect on attribute preference. You will now calculate Remainder(A_i)

See Back

3

10. Briefly explain how PDDL solves the frame problem. (1.5pts) Give some disadvantages to formulating problems in PDDL. (1pt)

In classical planning, most aspects of a state are unchanged as a result of an action. Actions in planning necessitate that you describe all aspects of a state after conclusion of an action (i.e. what changed and what stayed the same). The frame problem entails that it can be prohibitive to enumerate all aspects of the state that stayed the same after an action. PDDL addresses this by entailing by definition that any part of the state not described in the effect of an action stayed the same. Hence, PDDL only describes in an action's effect what changed.

3

Limitations of PDDL

1) PDDL has no significant way of entailing time. It only has a notion of before an action as time t and after the action as time $t+1$. Hence it would not be well suited for planning problems that are heavily reliant on time (e.g. scheduling).

2) PDDL has no scheme for quantifying action costs. Example I need to fly a plane from JFK to LAX. I can only do it by stopping in either London or Topeka Kansas. Clearly the flight through Topeka would be less costly (in terms of time, fuel, weather, etc). However from a PDDL perspective they have similar costs,

A_1	A_2	A_3	
T	T	T	F
T	T	F	F
T	F	T	F
T	F	F	F
F	T	T	T

Remainder$(A_1) = \frac{1}{5} \cdot B\left(\frac{1}{1}\right) + \frac{4}{5} B\left(\frac{0}{4}\right)$

$B(1) = 0$ since $\lg(1)=0$, $0 \cdot \lg(0) = 0$ so remainder (A_1) is 0

Remainder $(A_1) = 0$. Hence, since this attribute has 0 remainder it will have highest information gain (since as said before, $B(\frac{1}{5})$ can be treated as a constant since it applies for all attributes.

$$\text{Gain}(A_1) = B\left(\tfrac{1}{5}\right) - 0 = B\left(\tfrac{1}{5}\right)$$

$$\text{Gain}(A_2) = B\left(\tfrac{1}{5}\right) - \left[\tfrac{2}{5} \cdot \cancelto{0}{B\left(\tfrac{0}{2}\right)} + \tfrac{3}{5} B\left(\tfrac{1}{3}\right)\right]$$

$$\text{Gain}(A_2)\ B\left(\tfrac{1}{5}\right) - \tfrac{3}{5} B\left(\tfrac{1}{3}\right) = B\left(\tfrac{1}{5}\right) - \boxed{\tfrac{3}{5}}\underbrace{\left(\tfrac{1}{3}\lg\left(\tfrac{1}{3}\right) + \tfrac{2}{3}\lg\left(\tfrac{2}{3}\right)\right)}_{\text{greater than 0 so not selecting A2}}$$

$$\text{Gain}(A_3) = B\left(\tfrac{1}{5}\right) - \left[\tfrac{2}{5} \cdot \cancelto{0}{B\left(\tfrac{0}{2}\right)} + \tfrac{3}{5} B\left(\tfrac{1}{3}\right)\right]$$

$$\text{Gain}(A_3) = B\left(\tfrac{1}{5}\right) - \tfrac{3}{5}\underbrace{\left[\tfrac{1}{3}\lg\left(\tfrac{1}{3}\right) + \tfrac{2}{3}\lg\left(\tfrac{2}{3}\right)\right]}_{\text{greater than 0 so will not be selected since } A_1 \text{ has this term } 0}$$

A_1 selected