Summary of Course Material

The following summary contains key parts from the course lecture notes. It **does not** offer a complete coverage of course materials. You may use any of the claims shown here without proving them, unless specified explicitly in the question.

1 Search Problems

1.1 Uninformed Search

We are interested in finding a solution to a fully observable, deterministic, discrete problem. A search problem is given by a set of *states*, where a *transition function* $T(s, a, s')$ states that taking action a in state s will result in transitioning to state s'. There is a cost to this defined as $c(s, a, s')$, assumed to be non-negative. We are also given an initial state (assumed to be in our frontier upon initialization). The *frontier* is the set of nodes in a queue that have not yet been explored, but will be explored given the order of the queue. We also have a *goal test*, which for a given state s outputs "yes" if s is a goal state (there can be more than one). We have discussed two search

```
function TREE-SEARCH(problem) returns a solution, or failure
    initialize the frontier using the initial state of problem
    loop do
        if the frontier is empty then return failure
        choose a leaf node and remove it from the frontier
        if the node contains a goal state then return the corresponding solution
        expand the chosen node, adding the resulting nodes to the frontier
```

Figure 1.1: The tree search algorithm

```
function GRAPH-SEARCH(problem) returns a solution, or failure
    initialize the frontier using the initial state of problem
    initialize the explored set to be empty
    loop do
        if the frontier is empty then return failure
        choose a leaf node and remove it from the frontier
        if the node contains a goal state then return the corresponding solution
        add the node to the explored set
        expand the chosen node, adding the resulting nodes to the frontier
            only if not in the frontier or explored set
```

Figure 1.2: The graph search algorithm

variants in class, tree search (Figure 1.1) and graph search (Figure 1.2). They differ in the fact that under graph search we do not explore nodes that we have seen before.
The main thing that differentiates search algorithms is the order in which we explore the frontier. In breadth-first search we explore the shallowest nodes first; in depth-first search we explore the deepest nodes first; in uniform-cost search (Figure 1.3) we explore nodes in order of cost.

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
    node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
    frontier ← a priority queue ordered by PATH-COST, with node as the only element
    explored ← an empty set
    loop do
        if EMPTY?(frontier) then return failure
        node ← POP(frontier)  /* chooses the lowest-cost node in frontier */
        if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
        add node.STATE to explored
        for each action in problem.ACTIONS(node.STATE) do
            child ← CHILD-NODE(problem, node, action)
            if child.STATE is not in explored or frontier then
                frontier ← INSERT(child, frontier)
            else if child.STATE is in frontier with higher PATH-COST then
                replace that frontier node with child
```

Figure 1.3: The uniform cost search algorithm

We have also studied variants where we only run DFS to a certain depth (Figure 1.4) and where we iteratively deepen our search depth (Figure 1.5). Table 1 summarizes the various search algorithms' properties.

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
  return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  else if limit = 0 then return cutoff
  else
      cutoff_occurred? ← false
      for each action in problem.ACTIONS(node.STATE) do
          child ← CHILD-NODE(problem, node, action)
          result ← RECURSIVE-DLS(child, problem, limit − 1)
          if result = cutoff then cutoff_occurred? ← true
          else if result ≠ failure then return result
      if cutoff_occurred? then return cutoff else return failure
```

Figure 1.4: The depth-limited search algorithm

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
  for depth = 0 to ∞ do
      result ← DEPTH-LIMITED-SEARCH(problem, depth)
      if result ≠ cutoff then return result
```

Figure 1.5: The iterative deepening search algorithm

We also noted that no deterministic search algorithm can do well in general; in the worst case, they will all search through the entire search space.

1.2 Informed Search

Informed search uses additional information about the underlying search problem in order to narrow down the search scope. We have mostly discussed the A^* algorithm, where the priority queue holding the frontier is ordered by $g(v) + h(v)$, where $g(v)$ is the distance of a node from the source, and $h(v)$ is a *heuristic estimate* of the distance of the node from a goal node. There are two key types of heuristic functions

Definition 1.1. A heuristic h is *admissible* if it never overestimates the distance of a node from the nearest goal; i.e.

$$\forall v : h(v) \leq h^*(v),$$

where $h^*(v)$ is the *optimal heuristic*, i.e. the true distance of a node from the nearest goal.

Definition 1.2. A heuristic h is *consistent* if it satisfies the triangle inequality; i.e.

$$\forall v, v' : h(v) \leq h(v') + c(v, v')$$

where $c(v, v')$ is the cost of transitioning from v to v'.

We have shown that A^* with tree search is optimal when the heuristic is admissible, and is optimal with graph search when the heuristic is consistent. We also showed that consistency implies admissibility but not the other way around, and that running A^* with an admissible inconsistent heuristic with graph search may lead to a sub-optimal goal.

2 Adversarial Search

An extensive form game is defined by V a set of nodes and E a set of directed edges, defining a tree. The root of the tree is the node r. Let $V_{\max}$ be the set of nodes controlled by the MAX player and $V_{\min}$ be the set of nodes

Property	BFS	UCS	DFS	DLS	IDS
Complete	Yes	Yes	No	No	Yes
Optimal	No	Yes	No	No	No
Time	$\mathcal{O}(b^d)$	$\mathcal{O}(b^{1+\lfloor \frac{C^*}{\epsilon} \rfloor})$	$\mathcal{O}(b^m)$	$\mathcal{O}(b^\ell)$	$\mathcal{O}(b^d)$
Space	$\mathcal{O}(b^d)$	$\mathcal{O}(b^{1+\lfloor \frac{C^*}{\epsilon} \rfloor})$	$\mathcal{O}(bm)$	$\mathcal{O}(b\ell)$	$\mathcal{O}(bd)$

Table 1: Summary of search algorithms' properties

controlled by the MIN player. We often refer to the MAX player as player 1, and to the MIN player as player 2. A *strategy* for the MAX player is a mapping $s_1 : V_{\max} \to V$; similarly, a strategy for the MIN player is a mapping $s_2 : V_{\min} \to V$. In both cases, $s_i(v) \in chld(v)$ is the choice of child node that will be taken at node v. We let $\mathcal{S}_1, \mathcal{S}_2$ be the set of strategies for the MAX and MIN player, respectively.
The leaves of the minimax tree are *payoff nodes*. There is a payoff $a(v) \in \mathbb{R}$ associated with each payoff node v. More formally, the utility of the MAX player from v is $u_{\max}(v) = a(v)$ and the utility of the MIN player is $u_{\min}(v) = -a(v)$. The utility of a player from a pair of strategies $s_1 \in \mathcal{S}_1, s_2 \in \mathcal{S}_2$ is simply the utility they receive by the leaf node reached when the strategy pair (s_1, s_2) is played.

Definition 2.1 (Nash Equilibrium)**.** A pair of strategies $s_1^* \in \mathcal{S}_1, s_2^* \in \mathcal{S}_2$ for the MAX player and the MIN player, respectively, is a *Nash equilibrium* if no player can get a strictly higher utility by switching their strategy. In other words:

$$\forall s \in \mathcal{S}_1 : u_1(s_1^*, s_2^*) \geq u_1(s, s_2^*);$$
$$\forall s' \in \mathcal{S}_2 : u_2(s_1^*, s_2^*) \geq u_2(s_1^*, s')$$

Definition 2.2 (Subgame)**.** Given an extensive form game $\langle V, E, r, V_{\max}, V_{\min}, \vec{a} \rangle$, a subgame is a subtree of the original game, defined by some arbitrary node v set to be the root node r, and all of its descendants (i.e. its children, its children's children etc.), denoted by $desc(v)$. Terminal node payoffs the same as in the original extensive form game, and players still control the same nodes as in the original game.

Definition 2.3 (Subgame-Perfect Nash Equilibrium (SPNE))**.** A pair of strategies $s_1^* \in \mathcal{S}_1, s_2^* \in \mathcal{S}_2$ is a subgame-perfect Nash equilibrium if it is a Nash equilibrium for any subtree of the original game tree.

```
function MINIMAX-DECISION(state) returns an action
  return arg max_{a ∈ ACTIONS(s)} MIN-VALUE(RESULT(state, a))

function MAX-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s, a)))
  return v

function MIN-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← ∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s, a)))
  return v
```

Figure 2.1: The minimax algorithm (note a typo from the AIMA book - s should be $state$).

Figure 2.1 describes the minimax algorithm, which computes SPNE strategies for the MIN and MAX players, as we have shown in class. We discussed the α-β pruning algorithm as a method of removing subtrees that need not be explored (Figure 2.2).

3 Constraint Satisfaction Problems

A constraint satisfaction problem (CSP) is given by a set of variables $X_1, \ldots, X_n$, each with a corresponding domain $D_1, \ldots, D_n$ (it is often assumed that domain sizes are constrained, i.e. $|D_i| \leq d$ for all $i \in [n]$). Constraints specify relations between sets of variables; we are given $C_1, \ldots, C_m$ constraints. The constraint C_j depends on a subset of variables and takes on a value of "true" if and only if the values assigned to these variables satisfy C_j. Our objective is to find an assignment $(y_1, \ldots, y_n) \in D_1 \times \cdots \times D_n$ of values to the variables such that $C_1, \ldots, C_m$ are all satisfied. A binary CSP is one where all constraints involve at most two variables. In this case, we can write the relations between variables as a *constraint graph*, where there is an (undirected) edge between X_i and X_j if there is some constraint of the form $C(X_i, X_j)$.
In class we discussed *backtracking search* (Figure 3.1), which is essentially a depth-first search assigning variable values in some order. Within backtracking search, we can employ several heuristics to speed up our search.

1. When selecting the next variable to check, it makes sense to choose:

```
function ALPHA-BETA-SEARCH(state) returns an action
  v ← MAX-VALUE(state, −∞, +∞)
  return the action in ACTIONS(state) with value v

function MAX-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v

function MIN-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

Figure 2.2: The α-β pruning algorithm.

```
function BACKTRACKING-SEARCH(csp) returns a solution, or failure
  return BACKTRACK({ }, csp)

function BACKTRACK(assignment, csp) returns a solution, or failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(csp)
  for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
      if value is consistent with assignment then
          add {var = value} to assignment
          inferences ← INFERENCE(csp, var, value)
          if inferences ≠ failure then
              add inferences to assignment
              result ← BACKTRACK(assignment, csp)
              if result ≠ failure then
                  return result
      remove {var = value} and inferences from assignment
  return failure
```

Figure 3.1: Backtracking search

(a) the most constrained variable (the one with the least number of legal assignable values).

(b) the most constraining variable (the one that shares constraints with the most unassigned variables)

2. When selecting what value to assign, it makes sense to choose a value that is least constraining for other variables.

It also makes sense to keep track of what values are still legal for other variables, as we run backtracking search.

Forward Checking: as we assign values, keep track of what variable values are allowed for the unassigned variables. If some variable has no more legal values left, we can terminate this branch of our search. In more detail: whenever a variable X is assigned a value, the forward-checking process establishes arc consistency for it: for each unassigned variable Y that is connected to X by a constraint, delete from Y's domain any value that is inconsistent with the value chosen for X.

Arc Consistency: Uses a more general form of arc consistency; can be used when we assign a variable value (like forward checking) or as a preprocessing step.

Definition 3.1. Given two variables X_i, X_j, X_i is consistent with respect to X_j (equivalently, the arc (X_i, X_j) is consistent) if for any value $x \in D_i$ there exists some value $y \in D_j$ such that the binary constraint on X_i and X_j is satisfied with x, y assigned, i.e. $C_{i,j}(x, y)$ is satisfied (here, $C_{i,j}$ is simply a constraint involving X_i and X_j).

The AC3 algorithm (Figure 3.2) offers a nice way of iteratively reducing the domains of variables in order to ensure arc consistency at every step of our backtracking search. Whenever we remove a value from the domain of X_i with respect to X_j (the REVISE operation in the AC3 algorithm), we need to recheck all of the neighbors of X_i, i.e. add all of the edges of the form (X_k, X_i) where $k \neq j$ to the checking queue of the AC3 algorithm. We have seen in class that the AC3 algorithm runs in $\mathcal{O}(n^2d^3)$ time. In general, finding

```
function AC-3( csp) returns false if an inconsistency is found and true otherwise
  inputs: csp, a binary CSP with components (X, D, C)
  local variables: queue, a queue of arcs, initially all the arcs in csp

  while queue is not empty do
    (X_i, X_j) ← REMOVE-FIRST(queue)
    if REVISE(csp, X_i, X_j) then
      if size of D_i = 0 then return false
      for each X_k in X_i.NEIGHBORS - {X_j} do
        add (X_k, X_i) to queue
  return true

function REVISE( csp, X_i, X_j) returns true iff we revise the domain of X_i
  revised ← false
  for each x in D_i do
    if no value y in D_j allows (x,y) to satisfy the constraint between X_i and X_j then
      delete x from D_i
      revised ← true
  return revised
```

Figure 3.2: The AC3 Algorithm

a satisfying assignment (or deciding that one does not exist) for a CSP with n variables and domain size bounded by d takes $\mathcal{O}(d^n)$ via backtracking search; however, we have seen in class that if the constraint graph is a tree (or a forest more generally), we can find one in $\mathcal{O}(nd^2)$ time.

4 Reinforcement Learning

4.1 Basic Definitions

A reinforcement learning problem is defined over a Markov Decision Process. It comprises of

1. A finite set of states S.
2. A set of actions A that may be taken at states. It is possible that an action a cannot be taken at a state s.
3. A reward function $r : S \times A \to \mathbb{R}$, which specifies the (potentially negative) reward that the agent gets for taking the action a at state s. Reward functions can be *action independent*, in which case r is only a function of s, or *state independent* in which case r only depends on a. Reward functions can be *stochastic*: rather than a single numerical value $r(s, a) \in \mathbb{R}$, $r(s, a)$ specifies a probability distribution over the potential rewards one receives for taking the action a at the state s.
4. A transition model $\delta : S \times A \to \Delta(S)$, where $\Delta(S)$ is the space of probability distributions over S. Simply put, $\delta(s, a, s')$ is the probability that one transitions to the state s' when the action a is taken at the state s.

It is often assumed that our knowledge of the MDP is limited to our interactions with it. We may not know the rewards or transition model that define it, but only observe the states/actions/rewards/transitions after we have taken some actions.
A *policy* π is a mapping from states to actions. If π is determinstic, then $\pi(s)$ is a single action in A (at state s, take action a); if π is stochastic, then $\pi(s)$ is a distribution over actions. The *value function* is the expected discounted reward obtained by a policy over time. The value function $V^\pi(s)$ tries to predict how good the state s is under the policy π. It equals

$$V^\pi(s) = \sum_{t=0}^{\infty} \gamma^t r_t(a_t, s_t)$$

where $s_0 = s$. Here the reward function $r_t(a_t, s_t)$ refers to the expected reward of the agent at time t, and is given by:

$$r_t(a_t, s_t) = \sum_{a \in A, s \in S} r(s, a) \times \Pr[\pi(s_{t-1}) = a \wedge \delta(s_{t-1}, a) = s];$$

in other words, it is equal to the expected reward of the agent using the policy π, given the randomness of the policy itself (i.e. that the action a_t taken at time t is selected by π), and the MDP's transition model (i.e. that $\delta(s_{t-1}, a_t) = s_t$) (for the first step, i.e. $s_0 = s$ the expectation is only over the expected action that π might take at s, not over the probability that we transition to s). Note that $V^\pi(s)$ implicitly depends on the factor $\gamma \in [0, 1]$. It is often assumed that $\gamma > 0$ and that $\gamma < 1$; if $\gamma = 0$ then the agent does not care at all about future rewards, and if $\gamma = 1$ the agent perceives rewards at future states as equivalent to a reward at the current time step. The value function satisfies a recursive definition: $V^\pi(s_t) = r(a_t, s_t) + \gamma V^\pi(s_{t+1})$.
An optimal policy π^* picks an action that maximizes $r(a_t, s_t) + \gamma V^\pi(s_{t+1} = \delta(s_t, a_t))$. The value that it derives is given by the value function $V^*(s)$.

4.2 Q-Learning

We define $Q(s, a) = r(s, a) + \gamma V^\pi(\delta(s, a))$. $Q(s, a)$ is the utility we obtain if we take action a at state s, and then follow the policy π from then on. The *optimal Q function* $Q^*(s, a)$ takes the action a at state s, and then follows the optimal policy from then on, thus its value is $Q^*(s, a) = r(s, a) + \gamma V^*(\delta(s, a))$. This immediately implies that

$$Q^*(s, a) = r(s, a) + \gamma \max_{a' \in A} Q(\delta(s, a), a')$$

The value iteration algorithm is a framework for learning optimal Q values as we traverse an unknown MDP.

Algorithm 1 Value Iteration Algorithm

Require: An MDP given by states S, actions A, reward function r and transition model δ, discount factor γ, initial state s_0.

for $s \in S$ **do**
  for $a \in A$ **do**
    $\hat{Q}(s, a) \leftarrow 0$ ▷ Initialize Q values to 0
  end for
end for
for $t = 0, \ldots, \infty$ **do** ▷ At each iteration t
  Select an action a_t ▷ Based on policy
  Receive a reward $r_t \leftarrow r(s_t, a_t)$
  $s_{t+1} \leftarrow \delta(s_t, a_t)$
  $\hat{Q}(s_t, a_t) \leftarrow r_t + \gamma \max_{a'} \hat{Q}(s_{t+1}, a')$
end for

Algorithm 1 is guaranteed to converge to the optimal Q values under some mild assumptions. In class, we argued that the update rule $\hat{Q}(s_t, a_t) \leftarrow r + \gamma \max_{a'} \hat{Q}(s_{t+1}, a')$ may lead to high variance (say, a very high/low reward in a single round may cause significant fluctuations in the value of $\hat{Q}$ even if this occurs rarely). Thus, we suggested using the update rule

$$\hat{Q}(s_t, a_t) \leftarrow (1 - \alpha_t)\hat{Q}(s_t, a_t) + \alpha_t \times (r_t + \gamma \max_{a'} \hat{Q}(s_{t+1}, a'))$$

where α_t grows smaller as we further observe the effects of taking the action a_t at the state s_t, e.g.

$$\alpha_t(s_t, a_t) = \frac{1}{1 + N[s_t, a_t]}$$

where $N[s_t, a_t]$ counts the number of times we have already taken the action a_t at the state s_t. We also discussed potential candidates for selecting the action a_t. One option is taking it greedily (a bad idea, as seen in the regret minimization lecture), semi-randomly, or using more refined techniques such as the multiplicative weights update approach: $\Pr[a_t = a \mid s_t = s] \sim \mathrm{e}^{\varepsilon \times \hat{Q}(s,a)}$.

4.3 Regret Minimization and Expert Systems

1. We are given a set of actions $N = \{1, \ldots, n\}$; at round t, each action yields a reward in $\{0, 1\}$ (and returns to the same state for simplicity). We let $v_i^t \in \{0, 1\}$ be the value of expert i at time t; let $V_i^t = \sum_{q=1}^t v_i^q$, and $V_{\texttt{best}}^t = \max_i V_i^t$ be the total value of the best (maximal reward) action up to time t.

2. An online prediction algorithm $\mathcal{A}$ assigns a probability p_i^t to picking action i at time t. It chooses these probabilities before observing actions' rewards at time t, but does have knowledge of all actions' rewards at time steps $1, \ldots, t-1$. Thus, the reward of $\mathcal{A}$ at time t is $v_{\mathcal{A}}^t = \sum_{i=1}^n v_i^t p_i^t$, and its total reward is $V_{\mathcal{A}}^t = \sum_{q=1}^t v_{\mathcal{A}}^q$. The regret of an algorithm $\mathcal{A}$ at time t is measured as $\frac{V_{\mathcal{A}}^t}{t} - \frac{V_{\texttt{best}}^t}{t}$, i.e. the difference between the average value of $\mathcal{A}$ after t rounds and the average value of the best action after t rounds.

3. We have seen three algorithms in class:

   Greedy: at each time step t, choose the value maximizing action at time t; ties are broken arbitrarily. However, the greedy algorithm may yield a reward of 0 after t time steps, while there is an action that has a reward of $\lceil \frac{n-1}{n} t \rceil$.

   Randomized-Greedy (RG): at each time step t, let $B_t = \arg\max_i V_i^t$ be the set of best actions, i.e. those who had the highest reward up to (but not including) time t. The RG algorithm assigns a probability of $\frac{1}{|B_t|}$ on all actions in B_t (i.e. it picks an action from B_t uniformly at random). We have shown that RG has at most $\mathcal{O}(\log n)$ time steps that it has a reward of 0 for every time that the best action has a reward of 0.

   The multiplicative weights update algorithm (MWU): at time t, let $w_i^t = w_i^{t-1} \times \mathrm{e}^{\varepsilon v_i^{t-1}}$, where $\varepsilon > 0$. Then, $p_i^t = \frac{w_i^t}{\sum_{j=1}^n w_j^t}$. We have shown that

   $$\frac{V_{\texttt{MWU}}^t}{t} - \frac{V_{\texttt{best}}^t}{t} \leq \frac{\log n}{\varepsilon t} + \varepsilon$$

   In other words, the average regret of MWU decays linearly in t.

5 Logical Agents and Inference

A knowledge base KB is a set of logical rules that model what the agent knows. These rules are written using a certain language (or *syntax*) and use a certain truth model (or *semantics* which say when a certain statement is true or false). In propositional logic sentences are defined as follows

1. Atomic Boolean variables are sentences.
2. If S is a sentence, then so is $\neg S$.
3. If S_1 and S_2 are sentences, then so is:
   (a) $S_1 \wedge S_2$ "S_1 and S_2"
   (b) $S_1 \vee S_2$ "S_1 or S_2"
   (c) $S_1 \Rightarrow S_2$ "S_1 implies S_2"
   (d) $S_1 \Leftrightarrow S_2$ "S_1 holds if and only if S_2 holds"

We say that a logical statement a models b ($a \models b$) if b holds whenever a holds. In other words, if $M(q)$ is the set of all value assignments to variables in a for which a holds true, then $M(a) \subseteq M(b)$.
An inference algorithm $\mathcal{A}$ is one that takes as input a knowledge base KB and a query α and decides whether α is derived from KB, written as $KB \vdash_{\mathcal{A}} \alpha$. $\mathcal{A}$ is sound if $KB \vdash_{\mathcal{A}} \alpha$ implies that $KB \models \alpha$; $\mathcal{A}$ is complete if $KB \models \alpha$ implies that $KB \vdash_{\mathcal{A}} \alpha$.

5.1 Inference Algorithms

We saw algorithms that simply check all possible truth assignments, and then ensure that $KB \Rightarrow \alpha$, i.e. that for any truth assignment for which KB is true, the query α is true as well. Truth-table enumeration does so by brute force, whereas DPLL employs heuristics similar to those we've seen in CSPs (see Figure 5.1). Inference algorithms try to obtain new knowledge, i.e. new logical formulas, from the knowledge base. We have seen a few inference techniques:

Modus Ponens: $\alpha \wedge (\alpha \Rightarrow \beta)$ implies β.

And Elimination: $\alpha \wedge \beta$ implies β.

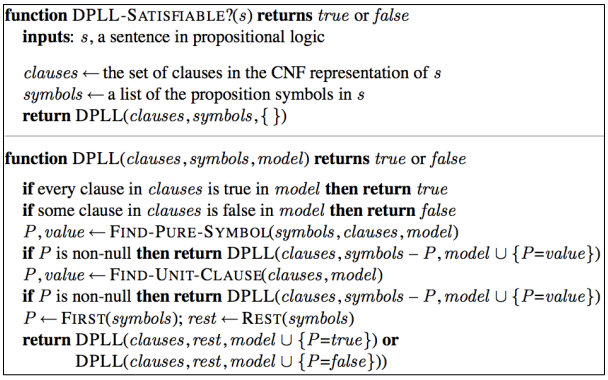

Figure 5.1: The DPLL algorithm

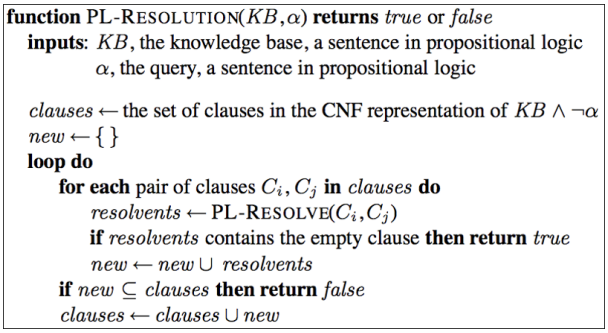

Figure 5.2: Resolution algorithm for propositional logic

Resolution: $\alpha \vee \beta$ and $\neg\alpha \vee \gamma$ implies $\beta \vee \gamma$.

In order to use resolution, KB must be in conjunctive normal form: it must comprise of a list of rules (clauses) that are of the form $\alpha_1 \vee \cdots \vee \alpha_k$. See Figure 5.2 One can also run a forward checking procedure in order to infer, but then KB must be in horn form: rules of the format $\alpha_1 \wedge \cdots \wedge \alpha_k \Rightarrow \beta$ with some goal query. See Figure 5.3 for details. We have also explored a goal based approach: backwards chaining, which is effectively the backtracking search algorithm for CSPs (Figure 3.1).

6 Inference in First Order Logic

First order logic (FOL) is a more expressive logic language, which includes existential quantifiers ($\exists x : P(x)$) and universal quantifiers ($\forall x : P(x)$). A key notion in FOL is **substitution**: a universal quantifier entails any substition of a constant into the variable: $\forall x : P(x)$ entails $P(a)$ for all a. This is also true for existential quantifiers, provided that the symbol used does not appear anywhere in the KB: $\exists x : P(x)$ entails $P(a_0)$ for some

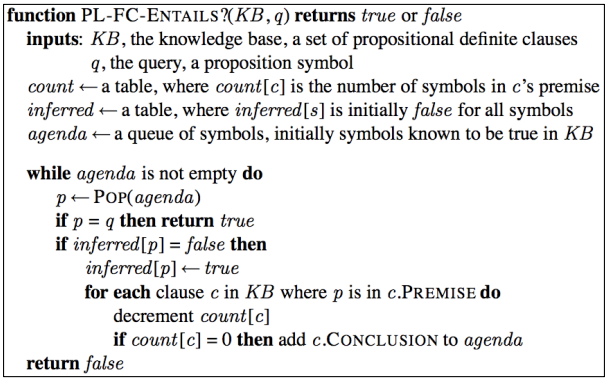

Figure 5.3: Forward chaining in propositional logic

```
function UNIFY(x, y, θ) returns a substitution to make x and y identical
  inputs: x, a variable, constant, list, or compound expression
          y, a variable, constant, list, or compound expression
          θ, the substitution built up so far (optional, defaults to empty)

  if θ = failure then return failure
  else if x = y then return θ
  else if VARIABLE?(x) then return UNIFY-VAR(x, y, θ)
  else if VARIABLE?(y) then return UNIFY-VAR(y, x, θ)
  else if COMPOUND?(x) and COMPOUND?(y) then
      return UNIFY(x.ARGS, y.ARGS, UNIFY(x.OP, y.OP, θ))
  else if LIST?(x) and LIST?(y) then
      return UNIFY(x.REST, y.REST, UNIFY(x.FIRST, y.FIRST, θ))
  else return failure
```

```
function UNIFY-VAR(var, x, θ) returns a substitution

  if {var/val} ∈ θ then return UNIFY(val, x, θ)
  else if {x/val} ∈ θ then return UNIFY(var, val, θ)
  else if OCCUR-CHECK?(var, x) then return failure
  else return add {var/x} to θ
```

Figure 6.1: The unification algorithm

```
function FOL-FC-ASK(KB, α) returns a substitution or false
  inputs: KB, the knowledge base, a set of first-order definite clauses
          α, the query, an atomic sentence
  local variables: new, the new sentences inferred on each iteration

  repeat until new is empty
      new ← { }
      for each rule in KB do
          (p1 ∧ ... ∧ pn ⇒ q) ← STANDARDIZE-VARIABLES(rule)
          for each θ such that SUBST(θ, p1 ∧ ... ∧ pn) = SUBST(θ, p'1 ∧ ... ∧ p'n)
                      for some p'1, ..., p'n in KB
              q' ← SUBST(θ, q)
              if q' does not unify with some sentence already in KB or new then
                  add q' to new
                  φ ← UNIFY(q', α)
                  if φ is not fail then return φ
      add new to KB
  return false
```

Figure 6.2: Forward chaining for FOL

new constant a_0. This is called **Skolemization**. Unification is simply a variable substitution that makes two FOL sentences identical. We have seen the unification algorithm (Figure 6.1). Unification plays a key role in inference. For example, **generalized modus ponens** states that given sentences $P_1, \ldots, P_k$ and $R_1 \wedge \cdots \wedge R_k \Rightarrow Q$, if there is some substition θ that unifies R_j with P_j, then Q holds with the substitution θ; it also plays a key role in the FOL forward chaining algorithm variant (Figure 6.2).
Similarly, there is a backwards chaining variant for FOL (Figure 6.3). To run FOL resolution, one must first convert the KB to CNF form.

Standardize variables apart: each universal quantifier should have its own variable.

Skolemize: each existential variable is replaced by a Skolem function of the enclosing universally quantified variables.

Drop universal quantifiers: implicitly assumed from now.

Convert to CNF form: in the same way as you would for propositional logic.

To resolve two FOL clauses, they must contain two complementary FOL literals: these are literals for which there is a unification (and corresponding substitution θ) such that one is the negation of the other. For example if we have $P \vee S$ and $Q \vee R$ such that $S(\theta) = \neg R(\theta)$ then we can entail $P(\theta) \vee Q(\theta)$. Resolution in FOL follows the same rule as propositional resolution, with the appropriate substituted resolvents.

```
function FOL-BC-ASK(KB, query) returns a generator of substitutions
  return FOL-BC-OR(KB, query, { })

generator FOL-BC-OR(KB, goal, θ) yields a substitution
  for each rule (lhs ⇒ rhs) in FETCH-RULES-FOR-GOAL(KB, goal) do
    (lhs, rhs) ← STANDARDIZE-VARIABLES((lhs, rhs))
    for each θ' in FOL-BC-AND(KB, lhs, UNIFY(rhs, goal, θ)) do
      yield θ'

generator FOL-BC-AND(KB, goals, θ) yields a substitution
  if θ = failure then return
  else if LENGTH(goals) = 0 then yield θ
  else do
    first, rest ← FIRST(goals), REST(goals)
    for each θ' in FOL-BC-OR(KB, SUBST(θ, first), θ) do
      for each θ'' in FOL-BC-AND(KB, rest, θ') do
        yield θ''
```

Figure 6.3: Backwards chaining algorithm for FOL