Games

Joint dependency, finite (states & moves) (infinite by heuristic cutoffs), zero sums (fully competitive: one win, one loss), deterministic (no chances), perfect information (state fully observable)

A Two-Player Zero-Sum game consists of the following:

- Two **players** Max and Min.
- A set of **positions** P (states of the game).
- A **starting position** p ∈ P (where game begins).
- A set of **Terminal positions** T ⊆ P (where game can end).
- A set of directed **edges** E_{Max} between some positions, representing Max's moves.
- A set of directed edges E_{Min} between some positions, representing Min's moves.
- A **utility (or payoff) function** U : T → R, representing how good each **terminal state** is for player Max.

Game state: a state-player pair, specifies the current state and whose turn it is.

1.Minimax Search

- **strategy**: Max always plays a move to change the state to the highest valued child. • Min always plays a move to change the state to the lowest valued child.
- **utility**: Assuming player play their best move, utility for each node: U(S) = U(s) if s is a terminal; min(child) if s is Min node; max(child) if s is a Max

```
Def DFMiniMax(s, player):
  // return utility of state s given that player is Min or Max
  If s is terminal
      Return U(s)  // return terminal state utility
  // apply player's move to get success states
  ChildList = s.Successors(player)
  If player == MIN
      Return min of DFMiniMax(c,MAX) over c ∈ ChildList
  Else: // player is Max
      Return max of DFMiniMax(c, MIN) over c ∈ ChildList
```

DFS: the game tree has to have finite depth; must traverse the entire search tree to evaluate all options;

Time complexity: $O(b^d)$, b is the num of legal moves at each state, and d is maximum depth of the tree.

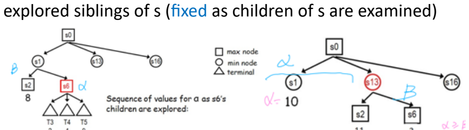
Space complexity: $O(bd)$

2.Alpha beta pruning

Alpha cuts (cutting Max nodes): at a Max node

α_s: the highest value of s's children examined so far (changing as children of s are examined).

β: the lowest value found so far by s's parent, from previously explored siblings of s (fixed as children of s are examined)



Beta cuts (cutting Min nodes): at a Min node

α: the highest value found so far by s's parent, from the previously-explored siblings of s (fixed as children of s examined)

β_s: the lowest value of s's children examined so far (changes as children of s are examined)

α: Best already explored option along the path to root for Max

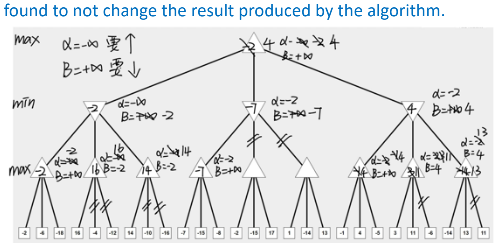
β: Best already explored option along the path to rood for Min

Alpha-Beta pruning:

- set initial values: $\alpha = -\infty$ and $\beta = \infty$
- while backing the utility values up the tree, identify α and β for each node.
- at every node s, if $\alpha \geq \beta$, prune (remaining) children of s

```
Def AlphaBeta(s, Player, alpha, beta):
// return utility of state s give that player is Min or Max
if s is TERMINAL
    Return U(s) // return terminal states utility
ChildList = s.Successors(Player)
if Player == MAX:
    ut_val = -inifity
    For c in ChildList:
        ut_val = max(ut_val, AlphaBeta(c, MIN, alpha, beta))
        If alpha < ut_val:
            Alpha = ut_val
            If beta <= alpha: break
    return ut_val
Else // player is MIN
    ut_val = inifity
    For c in ChildList:
        ut_val = min(ut_val, AlphaBeta(c, MAX, alpha, beta))
        If beta > ut_val:
            Alpha = ut_val
            If beta <= alpha: break
    return ut_val
```

- We can use heuristics to estimate the value, and then choose the child with highest (lowest) heuristic value.
- Effectiveness: with no pruning, $O(b^d)$nodes are explored. If the moving ordering for the search is optimal (meaning the best moves are searched first), the number of nodes we need to search using alpha beta pruning is $O(b^{d/2})$
- Large game: limit depth of search tree: • must stop at some non-terminal nodes. • must make heuristic estimates about the values of the non terminal position where we terminate the search • these heuristics are often called evaluation functions, • which are often a combination of learned and hard-coded rules (could be a weighted sum of features, can be learnt)
- This speeds up the minimax algorithm whenever pruning is possible, by reducing the number of nodes that need to be examined. This is achieved by pruning nodes which have been found to not change the result produced by the algorithm.



TUT: We let α(x) be the least payoff that MAX can guarantee at node x (best alternative for Max along this particular path from root to state s), and β (x) be the maximal payoff MIN has to pay at x (best alternative for Min).

a(node) = least min can pick; b(node) = most max can pick

Reasoning under uncertainty

Acting rational -> maximize one's expected utility (prob dist.)

Axioms of probability

Given a set U (universe), a prob dist. Over U is a function that maps every subset of U to a real number and that satisfies the ..

- $Pr(U) = 1$ • $Pr(A) \in [0,1]$
- $Pr(A \cup B) = Pr(A) + Pr(B) - Pr(A \cap B)$

(if A and B mutually exclusive, then $Pr(A \cap B) = 0$)

Expected value: $E[X] = \sum_{k=1}^{n} a_k p_k$

Linearity of expectation: $E[\sum_{k=1}^{n} X_i] = \sum_{k=1}^{n} E[X_i]$ (regardless of whether they are independent)

A set of atomic events F: $Pr(F) = \sum_{e \in F} Pr(e)$

Probability over feature vectors:

each different total assignment to these variables will be an **Atomic event** $e \in U$, # of atomic events = $\prod_i |Dom[V_i]|$, grows exponentially with the number of variables.

$Pr(V_1 = 1)$ indicate the set of all atomic events where $V_1 = 1$

The vector of probabilities $Pr(V_1, V_2)$ specifies the **joint distribution** of V_1 and V_2

Conditional Probability DEFINITION	$\Pr(A\|B) = \Pr(A \wedge B)/\Pr(B)$ proportional event
Summing out Rule	$\Pr(A) = \sum_{C_i} \Pr(A \wedge C_i)$ partition 2 $C_1 C_2 \ldots C_k$ ① no overlap ② sum is U when $\sum_{C_i} \Pr(C_i) = 1$ and $\Pr(C_i \wedge C_j) = 0\ (i \neq j)$
	$\Pr(A) = \sum_{C_i} \Pr(A\|C_i)P(C_i)$
Summing out Rule	$\Pr(A\|B) = \sum_{C_i} \Pr(A \wedge C_i\|B)$ when $\sum_{C_i} \Pr(C_i\|B) = 1$ and $\Pr(C_i \wedge C_j\|B) = 0\ (i \neq j)$
	$\Pr(A\|B) = \sum_{C_i} \Pr(A\|B \wedge C_i)\Pr(C_i\|B)$

$\sum_{d \in Dom[V_i]} \Pr(V_i = d) = 1$ and $\Pr(V_i = d_k \wedge V_i = d_m) = 0\ (k \neq m)$

$\sum_{d \in Dom[V_i]} \Pr(V_i = d|V_j = e) = 1$ and $\Pr(V_i = d_k \wedge V_i = d_m|V_j = e) = 0\ (k \neq m)$

Bayes rule $Pr(A|B) = Pr(B|A)Pr(A)/Pr(B)$

$P(B_k|A) = P(A|B_k)P(B_k)|(P(A|B_i)P(B_i)+..+P(A|B_n)P(B_n)$

Chain rule $Pr(A_1 \cap A_2 \ldots \cap A_n) =$

$Pr(A_n|A_1 \cap \ldots \cap A_{n-1})Pr(A_{n-1}|A_1 \cap \ldots \cap A_{n-2}) \ldots Pr(A_2|A_1)$

$P(F, D|G, C) = P(F|D, G, C)P(D|G, C)$

$P(F|D)P(D|C) = P(F, D|C)$

A and B are independent:

- $Pr(A|B) = Pr(A)$ • $Pr(A \cap B) = Pr(A)Pr(B)$

A and B are conditionally independent given C:

- $Pr(A|B \cap C) = Pr(A|C)$ • $Pr(A \cap B|C) = Pr(A|C)Pr(B|C)$

Normalize. $\boldsymbol{Normalize}([\boldsymbol{x_1}, \boldsymbol{x_2}, \ldots, \boldsymbol{x_k}])$

$= [\boldsymbol{x_1/\alpha}, \boldsymbol{x_2/\alpha}, \ldots, \boldsymbol{x_k/\alpha}], \boldsymbol{\alpha} = \sum_i \boldsymbol{x_i}$

$= \boldsymbol{Normalize}([\boldsymbol{\beta x_1}, \boldsymbol{\beta x_2}, \ldots, \boldsymbol{\beta x_k}])$

$= \boldsymbol{Normalize}(\boldsymbol{Normalize}([\boldsymbol{x_1}, \boldsymbol{x_2}, \ldots, \boldsymbol{x_k}]))$

Conditional Probability Table (CPT)

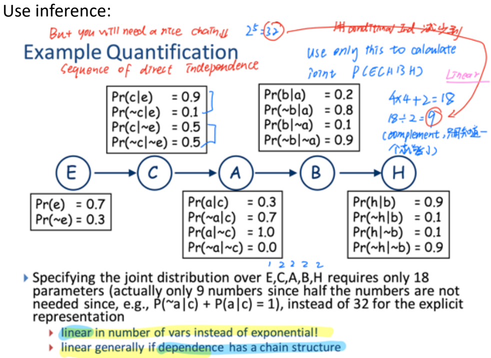
Pr(V1=1|V2=1,V3=1) Pr(V1=2|V2=1, V3=1) Pr(V1=3|V2=1 V3=1)

The value in each row for a different probability distribution

$Pr(V_1|V_2 = 1, V_3 = 2)$

- To reduce data and computational requirements

Use inference:



$Pr(H, B, A, C, E) = \Pr(H|B)\Pr(B|A)\Pr(A|C)\Pr(C|E)\Pr(E)$

$Pr(a) = \sum_{c_i \in Dom(C)} Pr(a|\ c_i)Pr(c_i) =$

$\sum_{c_i \in Dom(C)} Pr(a|\ c_i) \sum_{e_i \in Dom(E)} Pr(c_i|\ e_i)Pr(e_i)$

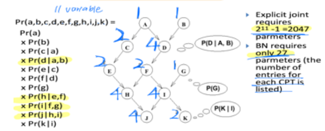
- P(c) = P(c|e)P(e) + P(c|~e)P(~e)
  = 0.9 * 0.7 + 0.5 * 0.3 = 0.78
- P(~c) = P(~c|e)P(e) + P(~c|~e)P(~e)
  = 0.1 * 0.7 + 0.5 * 0.3 = 0.22 = 1 – P(c)   complement
- P(a) = P(a|c)P(c) + P(a|~c)P(~c)
  = 0.3 * 0.78 + 1.0 * 0.22 = 0.454
- P(~a) = P(~a|c)P(c) + P(~a|~c)P(~c)
  = 0.7 * 0.78 + 0.0 * 0.22 = 0.546 = 1 – P(a)

Bayesian Networks

A BN over variables $\{x_1, x_2, \ldots, x_n\}$ consists of:

- A DAG (directed acyclic graph) whose nodes are variables
- a set of CPTS $Pr(x_i|Par(x_i))$ for each x_i
- Family of x_i is $\{x_i, \{Par(x_i)\}\})$

Example (Binary valued Variables)



Construct a Bayes Net

- Take any ordering of the variables. From the chain rule, we can write the joint distribution as

$Pr(x_1, .., x_n)$

$= Pr(x_n|x_1, .., x_{n-1})Pr(x_{n-1}|x_1, .., x_{n-2}) \ldots Pr(x_1)$

- Now for each x_i go through its conditioning set $x_1, .., x_{i-1}$, and remove all variables x_j such that x_i is conditionally independent of x_j given the remaining variables.
- The end product will be a product decomposition / Bayes net

$Pr(x_n|Par(x_n))\ Pr(x_{n-1}|Par(x_{n-1})) \ldots\ Pr(x_1)$

The numeric values associated with each $Pr(x_i|Par(x_i))$ in CPT

If each variable has d different values: table size = $d^{|x_i, Par(x_i)|}$, that is exponential in the size of the parent set.

-> **Ordering based on causality**

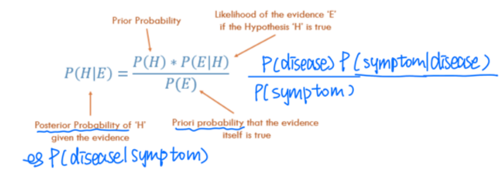
$Pr(M, F, C, Ache) = Pr(A|M, F, C)Pr(C|M, F)Pr(F|M)Pr(M)$

$= Pr(A|M, F, C)Pr(C)Pr(F)Pr(M)$

Ordering causes (M,F,C) come before effects (Aches)

$Pr(M|A, F, C)$ can't be simplified as C and F explain away Aches!

• Variable elimination



- To compute P(D|h,-i) -> normalize P(d, h, -i) and P(-d, h, -i)
- Or keep D as variable and compute a function of D, $f_k(D)$
- In general, at each stage VE will sum out the innermost variable, computing a new function over the variables in that sum • The function specifies one number for each different instantiation of its arguments • we store these functions as a table with one entry per instantiation of the variables • the size of these tables is exponential in the number of variables appearing in the sum.e.g. $\sum_F Pr(F|D)Pr(h|E, F)t(F)$ depends on the value of D and E, thus we will obtain $|Dom(D)| * |Dom(E)|$ different numbers in the resulting table.
- We call these tables of values computed by VE factors.

• Factors

1.The product of two factors

Let f(X,Y) & g(Y,Z) be two factors with variables Y in common

h(X,Y,Z) = f(X,Y) x g(Y,Z)

f(A,B)		g(B,C)		h(A,B,C)			
ab	0.9	bc	0.7	abc	0.63	ab~c	0.27
a~b	0.1	b~c	0.3	a~bc	0.08	a~b~c	0.02
~ab	0.4	~bc	0.8	~abc	0.28	~ab~c	0.12
~a~b	0.6	~b~c	0.2	~a~bc	0.48	~a~b~c	0.12

$h(Y) = \sum_{x \in Dom(X)} f(x,Y)$

f(A,B)		h(B)	
ab	0.9	b	1.3
a~b	0.1	~b	0.7
~ab	0.4		
~a~b	0.6		

Define $h = f_{X=a}$ as: h(Y) = f(a,Y)

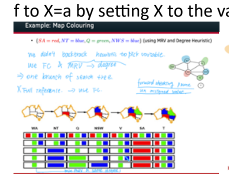
f(A,B)		h(B) = $f_{A=a}$	
ab	0.9	b	0.9
a~b	0.1	~b	0.1
~~ab~~	~~0.4~~		
~~a~b~~	~~0.6~~		

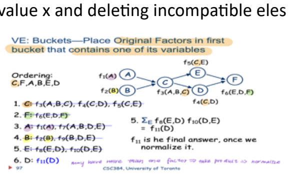
2.Summing a variable out of a factor

Let f(X,Y) be a factor with variable X (Y is a set), we sum out variable X form f to product a new factor h

3.Restricting a Factor

Let f(X,Y) be a factor with variable X (Y is a set), we restrict factor f to X=a by setting X to the value x and deleting incompatible eles

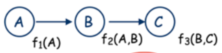




Algo: query Var **Q**, evidence vars **E** (values **e**), remaining vars **Z, F** the original CPTs.
1.replace each factor $f \in F$ that mentions a variable(s) in **E** with its restriction $f_{E=e}$ (this might yield a factor over no variables, a constant)
2.For each z_j – in the order given – eliminate $z_j \in$ **Z** as:
(a) compute new factor $g_j = \sum_{z_j} f_1 * f_2 * \ldots * f_k$, where f_i are the factors in F that include z_j
(b) Remove the factors f_i that mention z_j from F and add new factor g_j to F
3.The remaining factors refer only to the query variable Q. Take their product and normalize to produce Pr(Q|**e**)

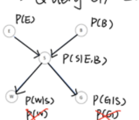
Numeric Example

- **Query**: C ⇒ cal $Pr(C)$, $C \in \{True, False\}$
- **No Evidence**



f_1(A)		f_2(A,B)		f_3(B,C)		f_4(B) $\Sigma_A f_2(A,B)f_1(A)$		f_5(C) $\Sigma_B f_3(B,C)\, f_4(B)$	
a	0.9	ab	0.9	bc	0.7	b	0.85	c	0.625
~a	0.1	a~b	0.1	b~c	0.3	~b	0.15	~c	0.375
		~ab	0.4	~bc	0.2	=0.9×0.9+0.1×0.4		0.9×0.8+0.1×0.6	
		~a~b	0.6	~b~c	0.8	=0.9×0.1+0.6×0.1=0.15		[illegible]	

(c) Query G, Evidence: W=w, order: E B, S, G



E: P(E) P(S|E,B)
B: P(B) F1(S,B)
S: P(w|S) P(G|S) F2(S)
G:

Eliminate E: F1(S,B) = Σ_E P(E)P(S|E,B) = P(e)P(s|e,B) + P(¬e)P(s|¬e,B)
F1(s,b) = P(e)P(s|e,b) + P(¬e)P(s|¬e,b) = 0.1×0.9 + 0.9×0.8 = 0.81
F1(¬s,b) = 0.19   F1(s,¬b) = 0.02   F1(¬s,¬b) = 0.98

Eliminate B: F2(S) = Σ_B P(B)F1(S,B) = P(b)F1(S,b) + P(¬b)F1(S,¬b)
F2(s) = P(b)F1(s,b) + P(¬b)F1(s,¬b) = 0.1×0.81 + 0.9×0.02 = 0.099
F2(¬s) = 0.901   F2(¬s)

Eliminate S: F3(G) = Σ_S P(w|S)P(G|S)F2(S) = P(w|s)P(G|s)F2(s) + P(w|¬s)P(G|¬s)F2(¬s)
F3(g) = P(w|s)P(g|s)F2(s) + P(w|¬s)P(g|¬s)F2(¬s) = 0.8×0.5×0.099 = 0.0396
F3(¬g) = 0.2198

Normalize F3(g): P(g|w) = $\frac{0.0396}{0.0396+0.2198}$ = 0.1527   P(¬g|w) = 0.8473

Complexity of VE

• Complexity of VE is exponential in the size of the largest factor generated during the VE, including the input CPTs • different elimination orderings can lead to different factor sizes. • heuristics can be used for picking more efficient orderings.

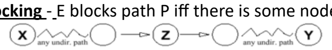
Min Fill Heuristic: always eliminate next the variables that creates the smallest size factor (# point to and point from)

• D-Separation for deriving conditional independence

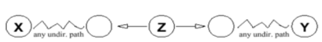
• Every x_i is conditionally independent of all of its nondescendants given it parents: $Pr(x_i|S \cup Par(x_i)) = Pr(x_i|Par(x_i))$ for any subset $S \subseteq NonDescendents(x_i)$

X and Y are conditionally independent given evidence E if E d-separates X and Y (if E = $\emptyset$, independent)

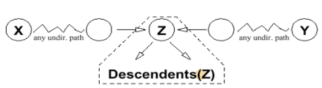
Blocking - E blocks path P iff there is some node z on the path:

(1)


If Z in evidence, the path between X and Y blocked

(2)


If Z in evidence, the path between X and Y blocked

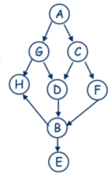
(3)


If Z is *not* in evidence and *no* descendent of Z is in evidence, then the path between X and Y is blocked

D-separation: A set of variables E d-separates X and Y if it blocks every undirected path in the BN between X and Y.

- In the following network determine if A and E are independent given the evidence:

1. A and E given no evidence? No Dep
2. A and E given {C}? No
3. A and E given {G,C}? Yes
4. A and E given {G,C,H}? Yes Independent
5. A and E given {G,F}? No
6. A and E given {F,D}? Yes
7. A and E given {F,D,H}? No
8. A and E given {B}? Yes
9. A and E given {H,B}? Yes
10. A and E given {G,C,D,H,D,F,B}? Yes



TUT:

• Show a is equivalent to b -> use a to show b

Knowledge representation and reasoning

Syntax: a grammar specifying what are legal syntactic constructs of the representation **Semantics**: a formal mapping from syntactic constructs to get theoretic assertions

Propositional logic

Syntax:

- **Propositional variable**: a variable which takes only True or False as values.
- **Propositional formula**: defined recursively.
  - Every propositional variable is a propositional formula
  - If A is a propositional formula, then so is $\neg A$
  - if A and B are propositional formulas, then so are:
    $A \wedge B$ (conjunction); $A \vee B$ (disjunction) $(\neg A \vee B)$;
    $A \rightarrow B$ (Implication); $A \leftrightarrow B$ (Bi-implication);

Semantic:

- **Truth Assignment**: a function τ from the propositional variables into the set of truth values {T, F}

Let τ be a truth assignment. The extension $\bar{\tau}$ of τ assigns either T or F to every formula and is defined as follows:

- If $A = x$, where x is a variable, then $\bar{\tau}(A) = \tau(x)$.
- $\bar{\tau}(\neg A) = T$ iff $\bar{\tau}(A) = F$;
- $\bar{\tau}(A \wedge B) = T$ iff $\bar{\tau}(A) = T$ and $\bar{\tau}(B) = T$;
- $\bar{\tau}(A \vee B) = T$ iff $\bar{\tau}(A) = T$ or $\bar{\tau}(B) = T$;
- $\bar{\tau}(A \rightarrow B) = F$ iff $\bar{\tau}(A) = T$ and $\bar{\tau}(B) = F$.
  When antecedent is false, the implication is *vacuously true* (intuitively this means that the implication does not really say anything).

- A truth assignment τ **satisfies** a formula A iff $\bar{\tau}(A) = T$

τ satisfies a set ϕ of formulas iff τ satisfies all formula in ϕ

- a set ϕ of formulas is **satisfiable** iff some truth assignment τ satisfies ϕ. Otherwise, ϕ is **unsatisfiable**.
- a formula A is a logical consequence of ϕ (denote by $\phi \mid= A$) iff for every truth assignment τ, if τ satisfies ϕ, then τ satisfies A.

Limitations: 1.only Boolean variables – cross references between individuals in statements are impossible (person vs alice, bob)
2.no quantifiers – to state a property for all (or some) members of the domain we have to explicitly list them.

First order logic: the most expressive logical language which has an (somewhat) "appropriate" automated procedure

Syntax:

For **first-order logic** the following components are required:

symbol
- A set V of variables.
- A set F of function symbols.
- A set P of predicate (relation) symbols.

f() is not a formula
terms
&
Atomic formula
vocabulary

- **Functions** and **variables** are used to construct terms. Terms denote elements of the domain
- **predicates** are define over terms. Atomic formulas denote properties and relations that hold about the elements in the domain
- **predicates** and **terms** are used to construct formulas (true or false assertion). Other formulas generate more complex assertions by composing atomic formulas. (term->formula, need predicate or connectives)

$\mathcal{L}$ –term

- A set $\mathcal{L}$ of **function** and **predicate symbols** is called a first-order vocabulary.

Let $\mathcal{L}$ be a set of function and predicate symbols.

- every variable is a term
- If f is an n-ary function symbol in $\mathcal{L}$ and $t_1,\ldots, t_n$ are $\mathcal{L}$ –terms, then $f(t_1,\ldots, t_n)$ is a $\mathcal{L}$ –term

Note: 0-ary function symbols are called constant symbols.

Vocabulary:

Let $\mathcal{L}$ be a vocabulary. The set of first-order $\mathcal{L}$-formulas is defined recursively:

1. Atomic Formula: $P(t_1, t_2, \ldots, t_n)$, where P is an n-ary predicate symbol in $\mathcal{L}$ and $t_1, t_2, \ldots, t_n$ are $\mathcal{L}$-terms.
↳ B.C
terms, → only consists function symbol & variable you can't have predicate. symbol.

2. Negation: $\neg f$, where f is a $\mathcal{L}$-formula.

3. Conjunction: $f_1 \wedge f_2 \wedge \ldots \wedge f_n$, where $f_1, f_2, \ldots, f_n$ are $\mathcal{L}$-formulas.

4. Disjunction: $f_1 \vee f_2 \vee \ldots \vee f_n$, where $f_1, f_2, \ldots, f_n$ are $\mathcal{L}$-formulas.

5. Implication: $f_1 \rightarrow f_2$, where f_1, f_2 are $\mathcal{L}$-formulas.

6. Existential: $\exists x f$, where x is a variable and f is a $\mathcal{L}$-formula.
(existential quantifier)

7. Universal: $\forall x f$, where ~~x is a variable~~ and f is a $\mathcal{L}$-formula.
(universal quantifier)

e.g. **vocabulary**: individuals -> constants (0-ary function, tony, rain); types -> unary predicates (s(x): s is a skier); relationship -> binary predicates (L(x,y): x likes y)

Semantic:

Semantics

- An interpretation (model) is a tuple $< D, \Phi, \Psi, V >$ mapping the symbols to semantic entities.
- D is a non-empty set of objects.
- Φ specifies the meaning of each constant and function symbol.
- Ψ specifies the meaning of each predicate symbol.
- V specifies the meaning of each variable.

Structure:

Let $\mathcal{L}$ be a first-order vocabulary. An $\mathcal{L}$-**structure** $\mathcal{M}$ consists of the following:

a set of function & predicate symbols

1. A **nonempty set** M called the universe (domain) of discourse.
2. For each n-ary **function symbol** $f \in \mathcal{L}$, an associated function $f^{\mathcal{M}} : M^n \rightarrow M$.
   Note: If $n = 0$, then f is a constant symbol and $f^{\mathcal{M}}$ is simply an element of M.
   $f^{\mathcal{M}}$ is called the **extension** of the function symbol f in $\mathcal{M}$.
3. relationship and properties between elements of the domain.
   For each n-ary **predicate symbol** $P \in \mathcal{L}$, an associated relation $P^{\mathcal{M}} \subseteq M^n$.
   $P^{\mathcal{M}}$ is called the **extension** of the predicate symbol P in $\mathcal{M}$.

L(x,y)
M = {a, b}   <a, b> ∈ L^M ⇒ a like b
<b, a> ∉ L^M ⇒ b doesn't like b

Suppose $\mathcal{L}_{BW}$ includes the following symbols:

- **Function Symbols:**
  - $under(x)$: the block immediately under x if x is not on table; x itself otherwise.
- **Predicate Symbols:**
  - $on(x, y)$: x is place (directly) on y.
  - $above(x, y)$: x is above y.
  - $clear(x)$: no blocks are above x. unary.
  - $ontable(x)$: no blocks are under x.

blocks word

$\mathcal{M}_1$ is a $\mathcal{L}_{BW}$-structure such that:
$M_1 = \{A, B, C, D\}$
predication $on^{\mathcal{M}_1} = \{\langle A, B\rangle, \langle B, C\rangle\}$
$above^{\mathcal{M}_1} = \{\langle A, B\rangle, \langle B, C\rangle, \langle A, C\rangle\}$
$clear^{\mathcal{M}_1} = \{A, D\}$ Nothing above.
$ontable^{\mathcal{M}_1} = \{C, D\}$
func $under^{\mathcal{M}_1}(A) = B$, $under^{\mathcal{M}_1}(B) =$ Insert Text Box
$under^{\mathcal{M}_1}(C) = C$, $under^{\mathcal{M}_1}(D) = D$

Structure :
a nonempty set M
an interpretation for every function symbol in the vocabulary
an interpretation for every predicate symbol in the vocabulary
a function needs to map every element in the domain.

Variable assignment

Let M be a structure and X be a set of variables. An object assignment σ for M is a mapping from (every) variables in X to the universe of M.

Let $\mathcal{L}$ be a vocabulary and $\mathcal{M}$ be an $\mathcal{L}$-structure.
The extension $\bar{\sigma}$ of σ is defined recursively:

1. for every variable x, $\bar{\sigma}(x) = \sigma(x)$; A
2. for every function symbol $f \in \mathcal{L}, \bar{\sigma}(f(t_1, \ldots, t_n)) = f^{\mathcal{M}}(\bar{\sigma}(t_1), \ldots, \bar{\sigma}(t_n))$.

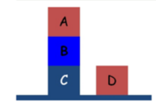
A∈M An∈M

remember for every object in the domain of M, f^M must be defined. cause f is a function. and must cover all objects in the domain of M.

look at the extention or the interpretation of f in M

Let $\mathcal{L}$ be a vocabulary and $\mathcal{M}$ be an $\mathcal{L}$-structure.
The extension $\bar{\sigma}$ of σ is defined recursively:

1. for every variable x, $\bar{\sigma}(x) = \sigma(x)$;
2. for every function symbol $f \in \mathcal{L}, \bar{\sigma}(f(t_1, \ldots, t_n)) = f^{\mathcal{M}}(\bar{\sigma}(t_1), \ldots, \bar{\sigma}(t_n))$.



$under^{\mathcal{M}}(A) = B$ $under^{\mathcal{M}}(B) = C$
$under^{\mathcal{M}}(C) = C$ $under^{\mathcal{M}}(D) = D$

$X = \{v_1, v_2, v_3, v_4\}$
$\sigma(v_1) = D$, $\sigma(v_2) = C$
$\sigma(v_3) = B$, $\sigma(v_4) = A$ what v4 is mapped.
object assigment function.

$\bar{\sigma}(under(under(v_4))) =$ $under^M(\overbrace{\bar{\sigma}(under(v_4))}^{B}) = under^M B = C$
$\bar{\sigma}(under(v_4)) = under^M(\bar{\sigma}(v_4)) = under^M(A) = B$
$\bar{\sigma}(v_4) = \sigma(v_4) = A$.

Models (interpretation):

For an $\mathcal{L}$-formula C, $\mathcal{M} \models C[\sigma]$ ($\mathcal{M}$ **satisfies** C under σ, or $\mathcal{M}$ is a **model** of C under σ) is defined recursively on the structure of C as follows (assuming A, B are $\mathcal{L}$-formulas):

$\mathcal{M} \models P(t_1, \ldots, t_n)[\sigma]$	iff	$\langle \bar{\sigma}(t_1), \ldots, \bar{\sigma}(t_n)\rangle \in P^{\mathcal{M}}$.
$\mathcal{M} \models (s = t)[\sigma]$	iff	$\bar{\sigma}(s) = \bar{\sigma}(t)$.
$\mathcal{M} \models \neg A[\sigma]$	iff	$\mathcal{M} \not\models A[\sigma]$.
$\mathcal{M} \models (A \vee B)[\sigma]$	iff	$\mathcal{M} \models A[\sigma]$ or $\mathcal{M} \models B[\sigma]$.
$\mathcal{M} \models (A \wedge B)[\sigma]$	iff	$\mathcal{M} \models A[\sigma]$ and $\mathcal{M} \models B[\sigma]$.
$\mathcal{M} \models (\forall x A)[\sigma]$	iff	$\mathcal{M} \models A[\sigma(m/x)]$ for all $m \in M$.
$\mathcal{M} \models (\exists x A)[\sigma]$	iff	$\mathcal{M} \models A[\sigma(m/x)]$ for some $m \in M$.

maps all variables exist x to the same values that σ maps to. But. x is fixed to m under this σ.

Note: $\sigma(m/x)$ is an object assignment function exactly like σ, but maps the variable x to the individual $m \in M$. That is:

For $y \neq x$: $\sigma(m/x)(y) = \sigma(y)$
For x: $\sigma(m/x)(x) = m$

Let $\mathcal{M}_3$ be a structure such that:
$M_3 = \{A, B, C, D\}$
$on^{\mathcal{M}_3} = \{\langle A, B\rangle, \langle B, C\rangle\}$ X
$above^{\mathcal{M}_3} = \{\langle A, B\rangle, \langle B, C\rangle, \langle A, C\rangle\}$
$clear^{\mathcal{M}_3} = \{A, D\}$
$ontable^{\mathcal{M}_3} = \{C, D\}$

Does $\mathcal{M}_3$ satisfy
$\forall x \forall y(above(x, y) \rightarrow on(x, y))$ False.

x=A y=C <A, C> ∈ $above^{M_3}$ <A, C> ∉ on^{M_3}

An occurrence of x in A is **bounded** iff it is in a sub-formula of A of the form $\forall x B$ or $\exists x B$. Otherwise the occurrence is **free**.
Example:
$P(x) \wedge \exists x[P(x) \vee Q(x)]$
free   bound

In a structure $\mathcal{M}$, formulas with **free variables** might be **true for some** object assignments to the free variables and **false for others**.

Example: Consider the formula $\underbrace{P(x, y) \wedge P(y, x)}_{A}$ and the following structure $\mathcal{M}$:
$M = \{a, b\}$ $P^{\mathcal{M}} = \{\langle a, a\rangle\}$

$\sigma_1(x) = a$ $\sigma_1(y) = a$ ⇒ $M \models A[\sigma_1]$
$\sigma_2(x) = b$ $\sigma_2(y) = b$ $\langle b, b\rangle \notin P^M$ ⇒ $M \not\models A[\sigma_2]$
$\sigma_3(x) = a$ $\sigma_3(y) = b$

A formula A is **closed** if it contains no free occurrence of a variable.
A **closed formula** is called a **sentence**.
Example:
$P(x) \wedge \exists x[P(x) \vee Q(x)]$ X
$\forall x P(x) \wedge \exists x[P(x) \vee Q(x)]$ ✓

map all variables to the same individual
If σ and σ' agree on the **free variables** of A, then $\mathcal{M} \models A[\sigma]$ iff $\mathcal{M} \models A[\sigma']$.
Proof: Structural induction on A.
First order formula

Corollary: If A is a **sentence**, then for any object assignments σ and σ',

$$\mathcal{M} \models A[\sigma] \quad \text{iff} \quad \mathcal{M} \models A[\sigma']$$

So, if A is a **sentence** (no free variables), σ is **irrelevant** and we omit mention of σ and simply write $\mathcal{M} \models A$.

Logical Satisfiability

Let Φ be a **set of sentences**.
↳ First order representation

- $\mathcal{M}$ satisfies Φ (denoted by $\mathcal{M} \models \Phi$) if for **every** sentence $A \in \Phi$, $\mathcal{M} \models A$.
- If $\mathcal{M} \models \Phi$, we say $\mathcal{M}$ is a model of Φ.

Satisfy all sentences at once

- We say that Φ is satisfiable if there is a structure $\mathcal{M}$ such that $\mathcal{M} \models \Phi$.

Eliminating Unintended Models: Example

Let Φ_2 be a set containing the following sentences($\mathbf{c_1}, \mathbf{c_2}$ are constant symbols):

- $\forall x(clear(x) \rightarrow \neg\exists y(on(y,x)))$
- $\forall x \forall y(on(x,y) \rightarrow above(x,y))$
- $\forall x \forall y \forall z((above(x,y) \wedge above(y,z)) \rightarrow above(x,z))$
- $on(\mathbf{c_1}, \mathbf{c_2})$
- $clear(\mathbf{c_1})$
- $above(\mathbf{c_1}, \mathbf{c_2})$

Construct **two models** of Φ_2 with **size three** (i.e., the size of the domain of each model must be three).

$M_2 = \{A, B, C\}$ $\quad c_1^{M_2} = A$ $\quad c_2^{M_2} = B$

$On^{M_2} = \{\langle A,B\rangle, \langle A,C\rangle\}$ $\quad Clear^{M_2} = \{A\}$

$above^{M_2} = \{\langle A,B\rangle \langle A,C\rangle, \langle B,C\rangle\}$

Logical Consequence

Let Φ be a set of sentences and A be a sentence.
A is a **logical consequence** of Φ (denoted by $\Phi \models A$) iff for every structure $\mathcal{M}$, if $\mathcal{M} \models \Phi$ then $\mathcal{M} \models A$.
every structure satis Φ, also satisfy A.

If A is a logical consequence of Φ, then there is no $\mathcal{M}$ such that $\mathcal{M} \models \Phi \cup \{\neg A\}$.
In other words, $\Phi \cup \{\neg A\}$ is **unsatisfiable**.

Example:
Assume Φ includes the following sentences:
$\forall x \forall y \forall z[(above(z,y) \wedge above(y,x)) \rightarrow above(z,x)]$
$above(\mathbf{c_1}, \mathbf{c_2}) \wedge above(\mathbf{c_2}, \mathbf{c_3})$ $\Big\}$ above (c_1, c_3)

KB entails f or f is a logical consequence of KB

$$\Leftrightarrow$$
$$(M \models KB) \rightarrow (M \models f)$$

f is true in every model of KB.

Knowledge base: A collection of sentences that represents what the agent/program believes about the world.
Sentences in the KB are explicit knowledge of the agent.
Logical consequences of the KB are implicit knowledge of the agent.

Proof procedure
A proof procedure is sound if whenever it produces a sentence A by manipulating sentences in a KB, then A is a logical consequence of KB (i.e., K B |= A). That is, all conclusions arrived at via the proof procedure **are correct**: they are logical consequences.
A proof procedure is complete if it can produce **all logical consequences** of KB. That is, if K B |= A, then the procedure can produce A.

Clausal form
A **literal** is an atomic formula or the negation of an atomic formula. Example: dog(fido), ¬cat(fido), P (x), ¬Q(y)
A **clause** is a disjunction of literals: Example: P (x) ⋁ ¬Q(x, y) ¬Owns(fido, fred) ⋁ ¬Dog(fido) ⋁ Person(fred)
A **clausal theory** is a set of clauses. It can also be considered as conjunction of clauses. Example:
{P (x) ⋁ ¬Q(x, y), ¬Owns(fido,fred) ⋁ ¬Dog(fido) ⋁ Person(fred)

Resolution by Refutation

implies
Resolution by Refutation to show $KB \models A$:

- Assume $\neg A$ is true to generate a contradiction. (**Refutation**)
- Convert $\neg A$ and all sentences in KB to a clausal theory C.
- Resolve the clauses in C until an empty clause is obtained.

Search problem

Resolution by Refutation: Example

Want to prove likes(**clyde,peanuts**) from:
KB
1. $elephant(\mathbf{clyde}) \vee giraffe(\mathbf{clyde})$
2. $\neg elephant(\mathbf{clyde}) \vee likes(\mathbf{clyde}, \mathbf{peanuts})$
3. $\neg giraffe(\mathbf{clyde}) \vee likes(\mathbf{clyde}, \mathbf{leaves})$
4. $\neg likes(\mathbf{clyde}, \mathbf{leaves})$

negation

A B C

Assume: 5. $\neg likes(\mathbf{clyde}, \mathbf{peanuts})$

Resolution by Refutation Proof: assign number
- $\neg likes(\mathbf{clyde}, \mathbf{peanuts})[5.]$
- 5&2: $\neg elephant(\mathbf{clyde})[6.]$
- 6&1: $giraffe(\mathbf{clyde})[7.]$
- 7&3: $likes(\mathbf{clyde}, \mathbf{leaves})[8.]$
- 8&4: ()

- $\neg\neg A$ iff A
- $\neg(A \wedge B)$ iff $\neg A \vee \neg B$
- $\neg(A \vee B)$ iff $\neg A \wedge \neg B$
- $\neg\forall x A$ iff $\exists x \neg A$
- $\neg\exists x A$ iff $\forall x \neg A$

Conversion to clausal form
1. Eliminate Implications.

$$A \rightarrow B \text{ iff } \neg A \vee B$$

2. Move Negations Inwards (and simplify ¬¬).
3. Standardize Variables: Rename variables so that each quantified variable is unique.

$$\forall x\Big[\neg P(x) \vee \Big((\forall y[\neg P(y) \vee P(f(x,y))]) \wedge (\exists y[Q(x,y) \vee \neg P(y)])\Big)\Big]$$

$$\forall x\Big[\neg P(x) \vee \Big((\forall y[\neg P(y) \vee P(f(x,y))]) \wedge (\exists z[Q(x,z) \vee \neg P(z)])\Big)\Big]$$

4. Skolemization: Remove existential quantifiers by introducing new function symbol

∃y(elephant(y) ⋀ friendly(y)) -> $elephant(\boldsymbol{a}) \wedge friendly(\boldsymbol{a})$

$\forall x \forall y \forall z \exists w(R(x,y,z,w))$ $\quad g_1(x,y,z)$

$\forall x \forall y \forall z(R(x,y,z,g_1(x,y,z)))$

$\forall x \forall y \exists w \forall z(R(x,y,w) \wedge Q(z,w))$ $\quad g_3(x,y)$

$\forall x \forall y \forall z\,(R(x,y,g_3(x,y)) \wedge Q(z,g_3(x,y)))$

$$\forall x\Big[\neg P(x) \vee \Big((\forall y[\neg P(y) \vee P(f(x,y))]) \wedge (\exists z[Q(x,z) \vee \neg P(z)])\Big)\Big]$$

$$\forall x\Big[\neg P(x) \vee \Big((\forall y[\neg P(y) \vee P(f(x,y))]) \wedge (Q(x,g(x)) \vee \neg P(g(x)))\Big)\Big]$$

5. Convert to Prenex Form. Bring all quantifier to front
6. Distribute Conjunctions over Disjunctions.

Conjunctions over Disjunctions: $A \vee (B \wedge C)$ iff $(A \vee B) \wedge (A \vee C)$

7. Flatten nested Conjunctions and Disjunctions. Remove(())
8. Convert to Clauses. Remove universal quantifiers and break apart conjunctions

- Resolution is *refutation complete*.
  If a set of clauses is *unsatisfiable* (i.e., when the answer is "YES") and so some branch contains [ ], a *breadth-first* search guaranteed to find [ ].
- But search *may not* terminate on *satisfiable* clauses (i.e., when the answer is "NO").

Decidability of FOL:
In general, • First-order unsatisfiability is semi-decidable, but not decidable. Thus, calculating entailments is semi-decidable and undecidable. • first-order satisfiability is undecidable.
Loosely speaking, a decision problem is
• decidable if there is some algorithm that correctly generates a "YES-NO" answer for every possible input. Otherwise, it's undecidable.
• semi-decidable if there is some algorithm that correctly generates "YES" answers, but does not terminate on some inputs for which the answer is "NO".

Possible Solutions:

- **Satisfiability:** Some first-order cases can be handled by converting them to a *proposi-tional form.*
- **Calculating Entailment (Unsatisfiability):**
  - Giving *control to user*.
    Example: Procedural Control of Reasoning.
  - Using *decidable fragments* of FOL (which are *less expressive*).
    Example: Description Logics, Horn Clauses.

1	Commutative law	$p \wedge q \equiv q \wedge p$	$p \vee q \equiv q \vee p$
2	Associative law	$(p \wedge q) \wedge r \equiv p \wedge (q \wedge r)$	$(p \vee q) \vee r \equiv p \vee (q \vee r)$
3	Distributive law	$p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$	$p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$
4	Identity law	$p \wedge \mathbf{true} \equiv p$	$p \vee \mathbf{false} \equiv p$
5	Universal bound law	$p \vee \mathbf{true} \equiv \mathbf{true}$	$p \wedge \mathbf{false} \equiv \mathbf{false}$
6	Idempotent law	$p \wedge p \equiv p$	$p \vee p \equiv p$
7	Negation law	$p \vee \neg p \equiv \mathbf{true}$	$p \wedge \neg p \equiv \mathbf{false}$
8	Double negation law	$\neg(\neg p) \equiv p$	
9	de Morgan's law	$\neg(p \wedge q) \equiv \neg p \vee \neg q$	$\neg(p \vee q) \equiv \neg p \wedge \neg q$
10	Absorption law	$p \vee (p \wedge q) \equiv p$	$p \wedge (p \vee q) \equiv p$
11	Implication law	$p \rightarrow q \equiv \neg p \vee q$	

Contrapositive

Search
• **Completeness**: a search algorithm is complete if whenever there is a path from the initial state to the goal, the algorithm will find it. • **Optimality**: Will the search always find the least cost solution? (when actions have costs) • **Time complexity**: What is the maximum number of nodes that can be expanded or generated? • **Space complexity**: What is the maximum number of nodes that have to be stored in memory

Uninformed Search Strategies - never look-ahead to the goal.
BFS & DEFS does not take into account edge weights.

Breadth-First Search (BFS): explores the search tree level by level: • Place the children of the current node at the end of the Frontier. • Frontier is a queue. Always extract first element of the Frontier.

• **Completeness**? Yes! non-decreasing path length complete as long as the state space has a finite branching factor
• **Optimality**: Shortest length solution? Yes! Least cost solution? Not necessarily...
• **Maximal Branching Factor b**: Maximum number of successors of any node. • **Depth of the shallowest solution d**: Length of the path from root (at depth 0) to the shortest solution at level d.
• **Time Complexity**: $1 + b + b^2 + .. + b^d + b(b^d - 1) \in O(b^{d+1})$
• **Space Complexity**: $O(b^{d+1})$
Assume 1. Expand nodes in layer d prior to discovering the goal
2.data space may not be able to explicitly represented.

Depth-First Search
- Place the children of the current node at the front of the Frontier.
- Frontier is a stack. Always extract first element of the Frontier.

• **Completeness**? No - Infinite paths cause incompleteness!
- Prune paths with cycles to get completeness, if state space is finite. • **Optimality**: No... (complete if finite depth)
• **Time Complexity**: $O(b^m)$ where m is the length of the longest path in the state space • **Space Complexity**: O(bm). Linear space complexity!

Depth Limited Search (DLS): • Truncate the search by looking only at paths of length D or less.
- Perform DFS but only to a pre-specified depth limit D.
- No nodes with path of length greater than D is placed on the Frontier.

• Benefit: Infinite length paths are not a problem.
• Limitation: Only finds a solution if a solution of depth less than or equal to D exists.

Iterative Deepening Search (IDS)
• Starting at depth limit d = 0, iteratively increase the depth limit and perform a depth limited search for each depth limit.
• Stop if a solution is found, or if the depth limited search failed without cutting of any nodes because of the depth limit.
- If no nodes were cut of, the search examined all nodes in the state space and found no solution, hence no solution exists.

• **Completeness**: Yes!
• **Optimality**: - Shortest length solution? Yes!
- Least cost solution? Not necessarily...* Can use a cost bound
• **Time Complexity**:

$$(d+1)b^0 + db + (d-1)b^2 + .. + b^d \in O(b^d)$$

• **Space Complexity**?: O(bd) Still linear!

IDS vs BFS
• Time complexity of IDS can be better than BFS since it does not expand nodes at the solution depth while BFS (in the worst case) must expand all the bottom layer nodes until it expands a goal node. - With a simple optimization BFS can achieve the same time complexity as IDS.
• Space complexity of BFS is much worse than IDS.
- In practice BFS can be much better depending on the problem: effective cycle checking can be employed with BFS.
- IDS cycle checking will make the space complex as bad as BFS.

Path Checking

In every path <pk, c>, where pk is the path <s0, s1, . . . , sk>, ensure that the final state c is not equal to any ancestors of c along this path. That is $c \notin \{s_0, s_1, \dots, s_k\}$

• Paths are checked in isolation!
• Advantage: Does not increase time and space complexity.
• Limitation: Does not prune all the redundant states. (e.g., redundant node in sibling)

Cycle Checking (aka Multiple Path Checking)
• Keep track of the all nodes previously expanded during the search using a list called the closed list.
[add to closed list, before expand]
• When we expand n_k to obtain successor c
- Ensure that c is not equal to any previously expanded node. If it is, we do not add c to the Frontier.

• Advantage: Very effective in pruning redundant states.
• Limitation: Expensive in term of space.
Space Complexity: $O(b^d)$ with optimization, $O(b^{d+1})$ without optimization (same as the space complexity of BFS).
For DFS, space complexity linear -> exponential, better use BFS

Cycle Checking and Optimal Cost
• Keep track of each state as well as the known minimum cost of a path to that state.
• If a more expensive path to a previously seen state is found, don't add the corresponding node to the Frontier.
• If a cheaper path to a previously seen state is found, add the corresponding node to the Frontier and
 * Remove other more expensive nodes to the same state from the Frontier or Lazily, ignore these more expensive nodes when/if they are removed for expansion

Uniform-Cost Search (UCS) - Finding optimal cost solution

- Always expand the least cost node on the Frontier.
  - priority queue (min heap, key=cost)
- Identical to BFS if all actions have the same cost.

1.Pop least cost node 2.Add successors

- **Completeness**: Yes, under non-zero constant lower-bound ϵ
- **Optimality:** yes
- **Time and Space Complexity**: $O(b^{floor(\frac{C^*}{\epsilon})+1})$

- UCS has to expand all nodes with cost less than C* and potentially all nodes with cost equal to C*.

[adding cycle checking to Uniform Cost Search can improve its efficiency in terms of running time and potentially reduce space complexity by avoiding redundant path explorations.

Maintenance of ordered frontier adds to space and time complexity

Heuristic Search – guess the cost to the goal through node n

Greedy Best-First Search

- Use h(n) to rank the nodes on the Frontier.
- Always expand a node with lowest h-value.
- Greedily trying to achieve a low-cost solution.
- Ignores the cost of n, so it can be lead astray exploring paths that cost a lot but seem to be close to the goal.

- Greedy search is **incomplete**.

A* Search take into account the cost of the path & heuristic

- define an evaluation function f(n) = g(n) + h(n)
- g(n): the cost of the path to n; h(n): the heuristic estimate of the cost of achieving the goal from n
- always expand the node with lowest f-value on the frontier
- f(n) is an estimate of the cost of getting to the goal via n

With cycle checking

(Node, Path)	Frontier
	{(A: 0+8=8)}
(A, A)	{(AC: 1+7=8), (AB: 4+3=7)}
(B, AB)	{(AC: 1+7=8), (ABC: 6+7=13), (ABD: 10+0=10)}
(C, AC)	{(ACB: 3+3=6), (ACD: 10+0=10), (ABC: 6+7=13), (ABD: 10+0=10)}
(B, ACB)	{(ACBC: 5+7=12), (ACBD: 9+0=9), (ACD: 10+0=10), (ABC: 6+7=13), (ABD: 10+0=10)}

Cycle-checking
1. if we already have a path to that node in frontier: keep only the cheapest path
(only where path ends matters)

Completeness

Theorem 1. A* will always find a solution if one exists as long as 1.the branching factor is finite. 2.every action has finite cost greater than or equal to ϵ; 3. h(n) is finite for every node n that can be extended to reach a goal node.

Proof:

- If a solution node n exists, then at all times either (a) n been expanded by A* or (b) an ancestor of n is on the Frontier.
- Suppose (b) holds and let the ancestor on the Frontier be n_i. Then n_i must have a finite f -value.
- As A* continues to run, the f -value of the nodes on the Frontier eventually increase. So, eventually either A* terminates because it found a solution OR n_i becomes the node on the Frontier with lowest f -value.
- If n_i is expanded, then either n_i = n and A* returns n as a solution OR n_i is replaced by its successors, one of which n_{i+1} is a closer ancestor of n.
- Applying the same argument to n_{i+1} we see that if A* continues to run without finding a solution it will eventually expand every ancestor of n, including n itself and so finds and returns a solution.

Admissibility -> optimality

Admissible heuristic

Let h*(n) be the cost of an optimal path from n to a goal node (∞ is there is no path). An admissible heuristic is a heuristic that satisfies the following condition for all nodes n in the search space: $h(n) \le h^*(n)$ [$h^*(n)$ is the actual opt cost]

To achieve **optimality**: • Each action in the search space must have cost ≥ ϵ > 0. • h must be admissible.

Ignore weights/cost: BFS, DFS; Optimal & uninformed: UCS
Informed, ignores cost: Greedy Best-First; with cost: A*
Use costs -> more optimal; Use heuristics -> faster

Intuition: • An admissible heuristic never over-estimates the cost to reach the goal, i.e., it is optimistic. • h(n) ⩽ h*(n) implies that the search won't miss any promising paths.

If it really is cheap to get to a goal via n (i.e., both g(n) and h∗(n) are low), then f (n) is also low, and eventually n will be expanded.

Theorem 2. A* with an admissible heuristic always finds an optimal cost solution, if s solution exists and as long as -the branching factor is finite - every action has finite cost greater than or equal to ϵ > 0

Proposition 1. A* with an admissible heuristic never expands a node with f -value greater than the cost of an optimal solution.

Proof: Let C* be the cost of an optimal solution.

Let p : $< s_0, s_1, \ldots, s_k >$ be an optimal solution.

So cost(p) = cost($< s_0, s_1, \ldots, s_k >$)= C*.

- It can be shown for each node in the search space that is reachable from the initial node, at every iteration an ancestor of the node is on the frontier. (induction)
- let n be a node reachable from the initial state and $n_0, n_1, \ldots, n_i, \ldots n$ be ancestors of n. So at least one of $n_0, n_1, \ldots, n_i, \ldots n$ is always on the frontier.
- We show that with an admissible heuristic, for every prefix (ancestor) n_i of n we have f (n_i) ⩽ C*:

C* = cost($< s_0, s_1, \ldots, s_k >$)
= cost($< s_0, s_1, \ldots, s_i >$)+ cost($< s_i, \ldots, s_k >$)
= g(n_i) + h*(n_i) by (1)
⩾ g(n_i)+ h(n_i) = f(n_i) by (2)

(1) g(n_i) is equal to cost(n_i) = cost($< s_0, s_1, \ldots, s_i >$)

- We know that A* always expands a node on the Frontier that has lowest f -value. So every node A* expands has f -value less than or equal to f (n_i), which is less than or equal to C*

Proof: Let C* be the cost of an optimal solution.

- If a solution exists then by **Theorem 1**, A* will terminate by expanding some solution node n.
- By **Proposition 1**, f (n) ⩽ C*.
- Since n is a solution node, we have h(n) = 0. So f(n) = g(n) = cost(n). We also have that C* ⩽ cost(n) = f (n) since no solution can have lower cost than the optimal.
- So cost(n) = C*. That is, A* returns an optimal solution.

Monotone heuristics (consistency)

A monotone (aka consistent) heuristic h is a heuristic that satisfies the triangle inequality: for all nodes n_1, n_2 and for all actions a we have that $\boldsymbol{h(n_1) \le C(n_1, a, n_2) + h(n_2)}$

where $C(n_1, a, n_2)$ denotes the cost of getting from the state of n_1 to the state of n_2 via action a

Theorem 3. Monotonicity implies admissibility. That is $(\forall n_1, n_2, a) h(n_1) \le C(n_1, a, n_2) + h(n_2) \Rightarrow (\forall \boldsymbol{n})\, \boldsymbol{h(n) \le h*(n)}$

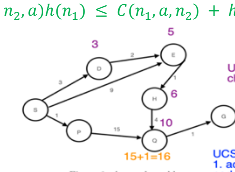


Figure 1: A search problem.

UCS finds the cheapest path to a node

UCS
1. add up cost of path = cost of node
2. take cheapest total path

(Node, Path)	Frontier
	{(S,0)}
(S, S)	{(SP,1), (SD, 3), (SE, 9)}
(P, SP)	{(SD,3), (SE, 9), (SPQ,16)}
(D, SD)	{(SDE,5), (SE,9), (SPQ,16)}
(E, SDE)	{(SDEH,6), (SE,9), (SPQ,16)}
(H, SDEH)	{(SE,9), (SDEHQ,10), (SPQ,16)}
(E, SE)	{(SEH,10), (SDEHQ,10), (SPQ,16)}
(H, SEH)	{(SDEHQ,10), (SEHQ,14), (SPQ,16) }
(Q, SDEHQ)	{(SDEHQG,11), (SEHQ,14), (SPQ,16)}
(G, SDEHQG)	{(SEHQ,14), (SPQ,16)}

Proof:
If no path exists from the state of n to a goal, then $h^*(n) = \infty$ and $h(n) \le h^*(n)$.

Else, let n_k be an arbitrary path for which there exists a path from its state to a goal state s_g.
Let $p_{k,g} : \langle s_k, s_{k+1}, \ldots, s_g \rangle$ be an *optiaml* path from the state of n_k to s_g.
The cost of $p_{k,g}$ is $h^*(n_k)$.
We prove $h(n_k) \le h^*(n_k)$ by induction on the length of $p_{k,g}$.

- **Base Case:** $s_k = s_g$.
  By our conditions on h, $h(n_k)$ and $h^*(n_k)$ are equal to zero. So $h(n_k) \le h^*(n_k)$
- **Induction Hypothesis:** $h(n_{k+1}) \le h^*(n_{k+1})$.

$$\begin{aligned} h(n_k) &\le C(n_k, a, n_{k+1}) + h(n_{k+1}) && \text{by monotonicity of } h \\ &\le C(n_k, a, n_{k+1}) + h^*(n_{k+1}) && \text{by IH} \\ &= h^*(n_k) \end{aligned}$$

- **Completeness**: Yes, see in Theorem 1.
- **Optimality**: With admissible heuristic, Yes. See in Theorem 2.
- **Space and Time Complexity**:

Hence the same bounds as uniform-cost apply: $O(b^{floor(\frac{C^*}{\epsilon})+1})$

Still exponential unless we have a very good h!

IDA* Iterative Deepening A*

- Like iterative deepening, but now the cut-off is the f -value rather than the depth. • At each iteration, the cut-off value is the smallest f -value of any node that exceeded the cut-off on the previous iteration. • Avoids overhead associated with keeping a sorted queue of nodes, and the Frontier occupies only linear space. - Reduce memory requirements for A*

CSP Backtracking Search

```
def BT(Level):
1.  if all Variables assigned
2.     PRINT Value of each Variable
3.     EXIT or RETURN                      # EXIT for only one solution
                                           # RETURN for more solutions
4.  V := PickUnassignedVariable()
5.  Assigned[V] := TRUE
6.  for d := each member of Domain(V)     # the domain values of V
7.     Value[V] := d
8.     ConstraintsOK := TRUE
9.     for each constraint C such that (i) V is a variable of C and
                                       (ii) all other variables of C are assigned:
10.        if C is not satisfied by the set of current assignments:
11.            ConstraintsOK := FALSE
12.    if ConstraintsOk == TRUE:
13.        BT(Level+1)
14.  Assigned[V] := FALSE     # UNDO as we have tried all of V's values
15.  RETURN
```

CSP & Inference

Inference: The Algorithm

```
def BT_with_Inference(Level)
1.  if all Variables assigned
2.     PRINT Value of each Variable
3.     EXIT or RETURN                      # EXIT for only one solution
                                           # RETURN for more solutions
4.  V := PickUnassignedVariable()
5.  Assigned[V] := TRUE
6.  for d := each member of CurDom(V)
7.     Value[V] := d                 assign value to node V   start inference
8.     Prune all values other than d from CurDom[V]
9. DWO: if(Inference(V) != DWO)     Domain wipe out
10. backtrack  BT_with_Inference(Level+1) # all remaining domain values are ok
11.    RestoreAllValuesPrunedByInference()
12.  Assigned[V] := FALSE     # UNDO as we have tried all of V's values
13.  RETURN

def Inference(var)
1.  VarQueue.push(var)        push node that assigned value
2.  while VarQueue not empty
3.     W := VarQueue.extract()    extract one node from Queue
4.     for each constraints C where W∈scope(C)    constraints include W
5.     for V := each member of scope(C) \ W    go over all variables in the scope of that constraint:
6.        S := CurDom[V]     eg S={1,2,3,4}
7.  eg Q1  for d := each member of CurDom[V]
8.           Find an assignment A for all other variables in scope(C)
             such that C(A ∪ V=d) is True     supporting assignment
9.           if A not found
10.              CurDom[V] = CurDom[V] - d    # remove d from the domain of V
11.              if CurDom[V] == {}     # DWO for V
12.                  empty VarQueue
13.                  return DWO         # return immediately
14.       if CurDom[V] ≠ S   cur={1,2}   # if domain of V has changed
15.          VarQueue.push(V)  -> 把V加到Q里
16.  return TRUE    # loop exited without DWO
```

keep track Curr Domain
Restore if backtrack.

Q1 Q2 Q3 Q4 U2 Q3

domain wipe out DWO -> try another value of Q

- Remove a node from queue, check each of its value in its domain, does each of them have supporting assignment, based on related node's current domain
- if no supporting assignment, prune this value in the domain -> add affected nodes to queue

Another way:
-Pop a node from queue, check its related node's domain (don't over check), if domain change, add the related node in queue.

~~C5 ≠ Bob~~

C4 {A, B}
C5 {J ≠ B}

C5 可以为 Bob
与C5 相关 2个constraint:
C5: Bob { C5 ≠ C3 => C3 = Anna ; C5 ≠ C4 => C4 = Anna } > 不要再想这个!!
1个 constraint 里 all variable 有无 supporting assi

1. Consider the following constraint satisfaction problem:

Variables:	X, Y, Z	
Domains:	$D_X = D_Y = D_Z = \{0, 1, 2, 3, 4\}$	
Constraints:	$C_1 := (X = Y + 1)$	
	$C_2 := (Y = 2Z)$	

x{~~0~~, 1, ~~2~~, 3, ~~4~~}
y{0, ~~1~~, 2, ~~3~~, ~~4~~}
z{0, 1, ~~2~~, ~~3~~, ~~4~~}

Assuming that initially no variable has been assigned and var = Y, perform inference on these constraints.

learned	Inference	Queue
pop Y		Y
x≠0	C1	
z≠3,4	C2	X, Z
pop X		
y≠4	C1	Z Y
~~z≠2~~		
pop Z		
y≠{1,3}	C2	Y
pop y		
X≠{2,4}	C1	X Z
Z≠{2}	C2	

Y X Z Y X Z

对于pop出来的node,
可以check 它的 domain是否变化 (课上)
也可以check 与它 relate 的 node 的 domain 是否变化 (左题)
不要乱!
eg pop X => 与它relate 的 y 别看它 domain.

Forward Checking

Forward Checking: The Algorithm

```
def FC_Inference(var)             only look at first level
1.  VarQueue.push(var)              don't check influence of prune
2.  while VarQueue not empty
3.     W := VarQueue.extract()    W=di
4.     for each constraints C where W∈scope(C)
5.        for V := each member of scope(C) \ W
6.           S := CurDom[V]
7.           for d := each member of CurDom[V]
8.              Find an assignment A for all other variables in scope(C)
                such that C(A ∪ V=d) is True
9.              if A not found
10.                CurDom[V] = CurDom[V] - d   # remove d from the domain of V
11.                if CurDom[V] == {}    # DWO for V
12.                   empty VarQueue
13.                   return DWO         # return immediately
14.       X { # if CurDom[V] ≠ S
15.           #    VarQueue.push(V)
16.  return TRUE   # loop exited without DWO
```

Forward Checking: Don't add variables to VarQueue at every iteration.
That is, *remove Lines 14 and 15* of $Inference(var)$

Other alternative Curdom <= 3… ==1

- In general, solving a CSP problem in the worst-case can take **exponential time**. general class of CSPs is **NP-complete**. **inference techniques and heuristics** is to solve those simpler sub-classes faster.

Degree Heuristic: Select the **variable** that is involved in the largest number of constraints on other unassigned variables. ->drive more inference -> fail as early as possible

Minimum Remaining Values Heuristics (MRV):

- **variable** with the smallest remaining values (smallest CurDom). (use min heap) ->faster to identity inconsistency

Least Constraining Value Heuristic:

- Always pick a **value** in CurDom that rules out the **least domain values** of other neighboring variables in the constraint.

-> maximum flexibility for subsequent variable assignments.
->higher probability to find solution ->avoid conflict.