Basic Search Notes

- Completeness: An algorithm is complete when, assuming a solution exists, the algorithm is guaranteed to find it in finite time
- Time complexity: Number of nodes generated/expanded
- Space complexity: Max number of nodes in memory
- Optimality: Does it always find a least cost solution

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes*	Yes*	No	Yes, if $l \geq d$	Yes
Time	b^{d+1}	$b^{\lceil C^*/\epsilon \rceil}$	b^m	b^l	b^d
Space	b^{d+1}	$b^{\lceil C^*/\epsilon \rceil}$	bm	bl	bd
Optimal?	Yes*	Yes	No	No	Yes*

A* Search

- Uses heuristic to do informed search.
- Each node assigned heuristic.
- For A* to work, the heuristic must never overestimate the cost.
- Explores similar to Uniform Cost Search
- Only add heuristic score to path of most recent node, as the heuristic is an estimate to the goal state.

Idea: avoid expanding paths that are already expensive

Evaluation function $f(n) = g(n) + h(n)$

$g(n)$ = cost so far to reach n
$h(n)$ = estimated cost to goal from n
$f(n)$ = estimated total cost of path through n to goal

A* search uses an admissible heuristic
i.e., $h(n) \leq h^*(n)$ where $h^*(n)$ is the **true** cost from n.
(Also require $h(n) \geq 0$, so $h(G) = 0$ for any goal G.)

E.g., $h_{\mathrm{SLD}}(n)$ never overestimates the actual road distance

If $h_2(n) \geq h_1(n)$ for all n (both admissible)
then h_2 dominates h_1 and is better for search

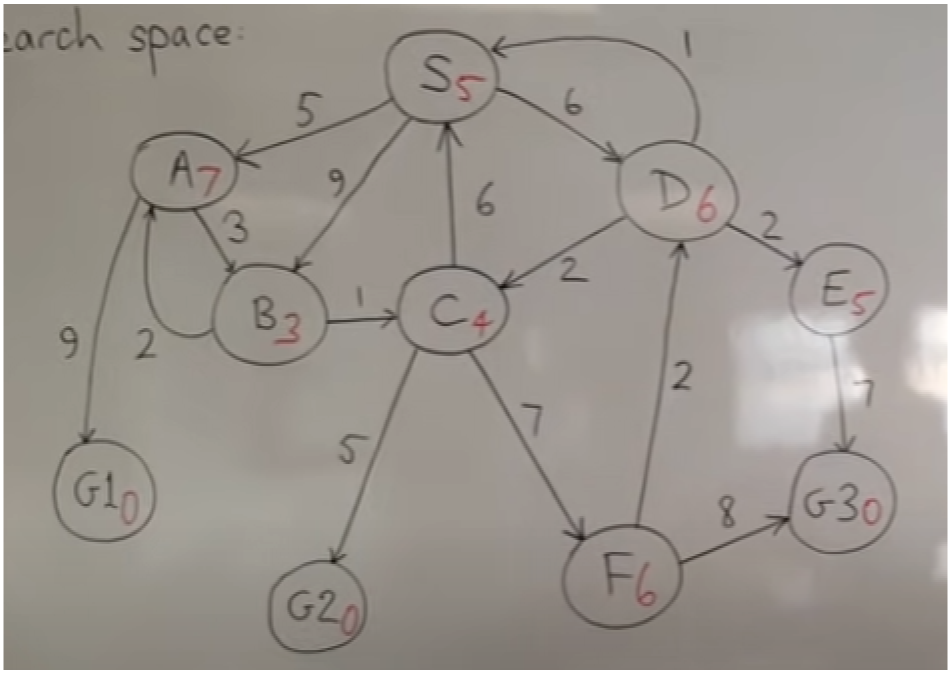
Typical search costs:

$d = 14$ IDS = 3,473,941 nodes
A*(h_1) = 539 nodes
A*(h_2) = 113 nodes
$d = 24$ IDS ≈ 54,000,000,000 nodes
A*(h_1) = 39,135 nodes
A*(h_2) = 1,641 nodes

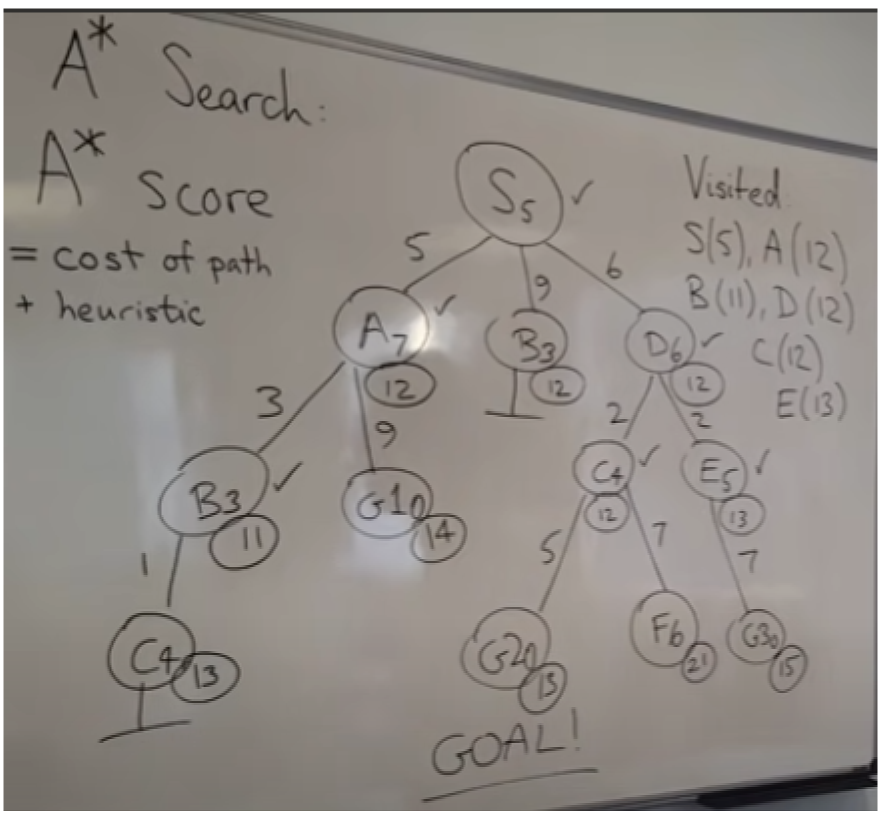
Given any admissible heuristics h_a, h_b,

$$h(n) = \max(h_a(n), h_b(n))$$

is also admissible and dominates h_a, h_b



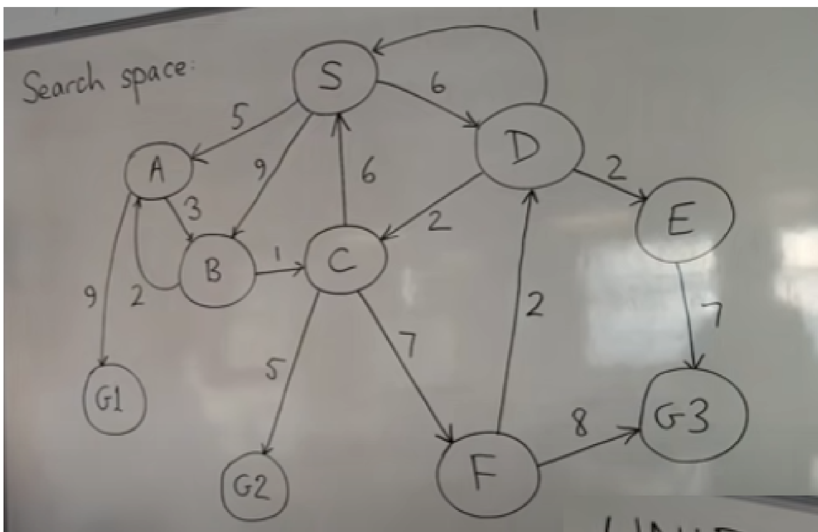
arch space:
S5
A7
B3
C4
D6
E5
F6
G10
G20
G30

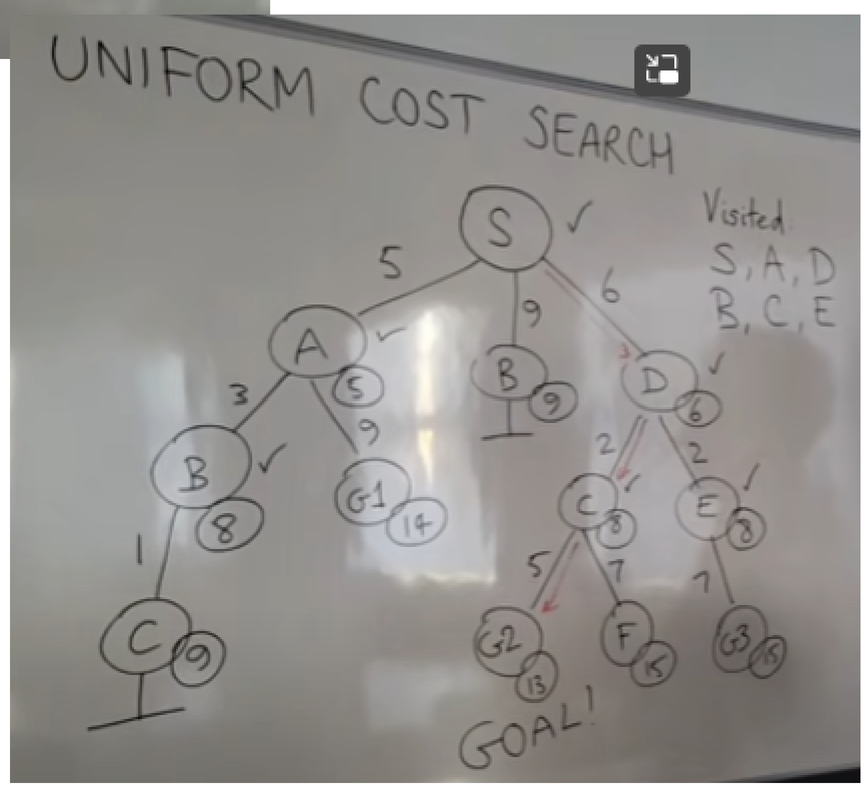


A* Search:
A* Score
= cost of path
+ heuristic
Visited
S(5), A(12)
B(11), D(12)
C(12)
E(13)
GOAL!

Uniform-Cost Search

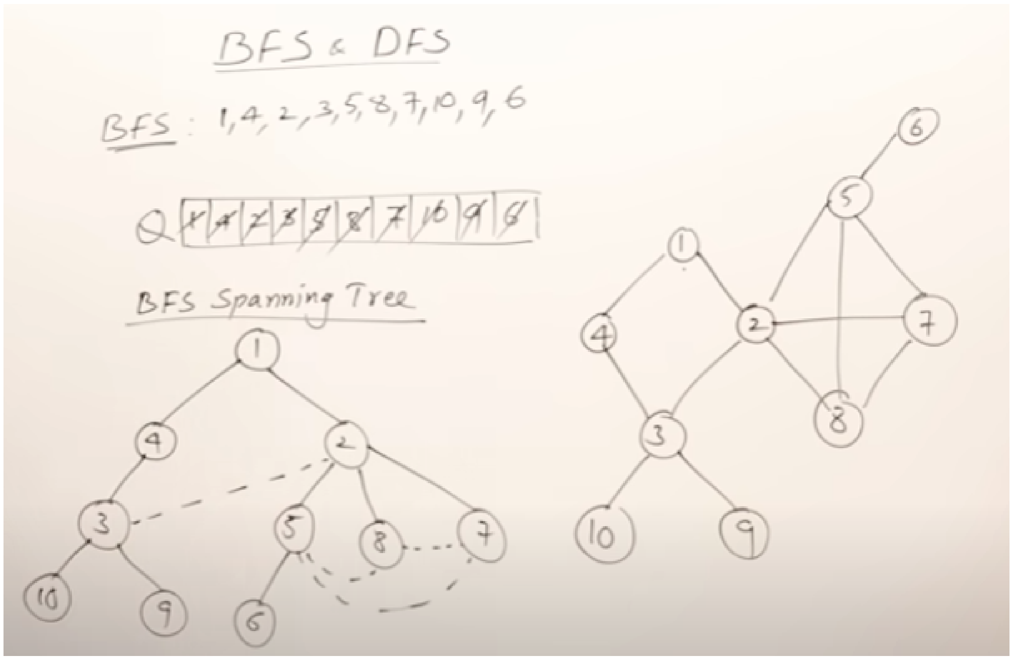
- Goal-directed agent moving around search space
- Uninformed search algorithm
- Explore everything from start
- Calculate new full paths costs
- Expand next node of cheapest path
- Don’t go back to nodes fully expanded
- Easiest way is tree with visited/expanded list





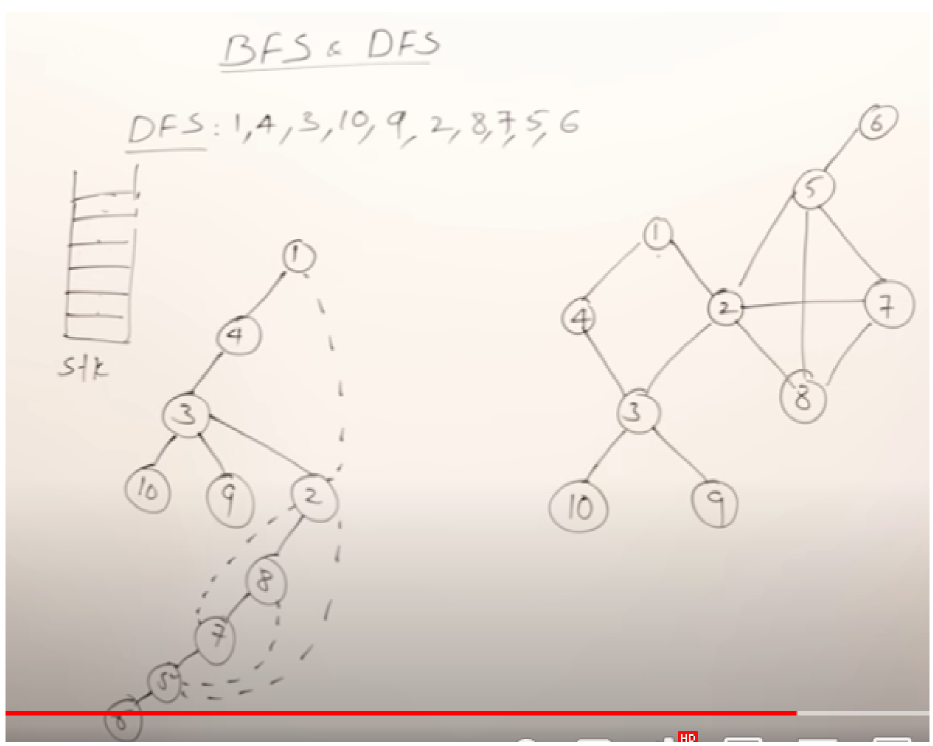
BFS

- Explore each point completely
- Best to store in queue



DFS

- Go as far as possible, then backtrack
- Best to store in stack

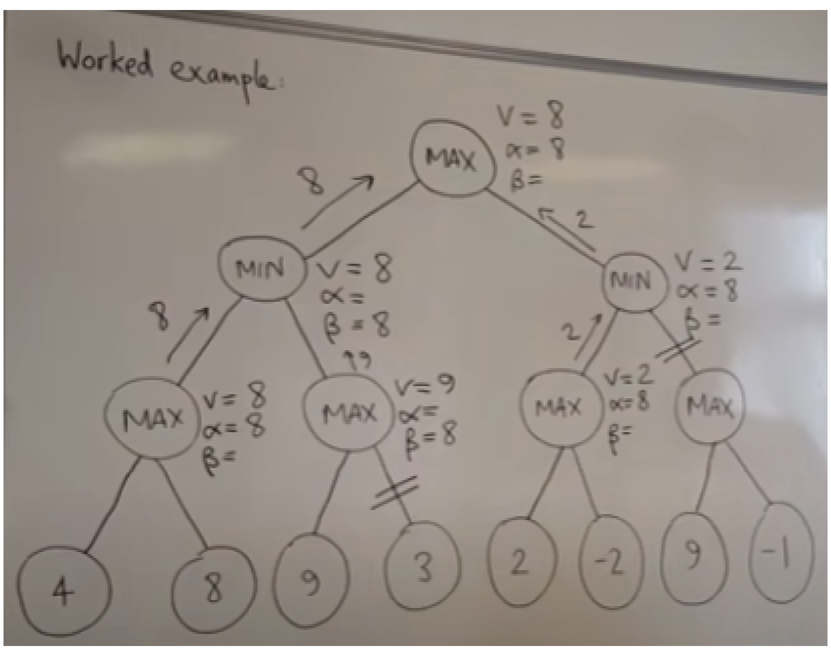


Iterative Deepening DFS

- Repeating a depth limited search but increasing the depth by one each time.
- Increase until goal found
- Start a fully new Depth Limited Search each time the depth is increased.

Alpha-Beta Pruning

- Used to shorten minimax search and "prune" off sections where it's impossible to get a number that will be advanced.
- Go left first in DFS type exploration
- Final answer should be the same whether the minimax was pruned or not.
- Alpha- Best already explored option along path to the root for the maximizer.
- Beta- Best already explored option along bath for the minimizer.



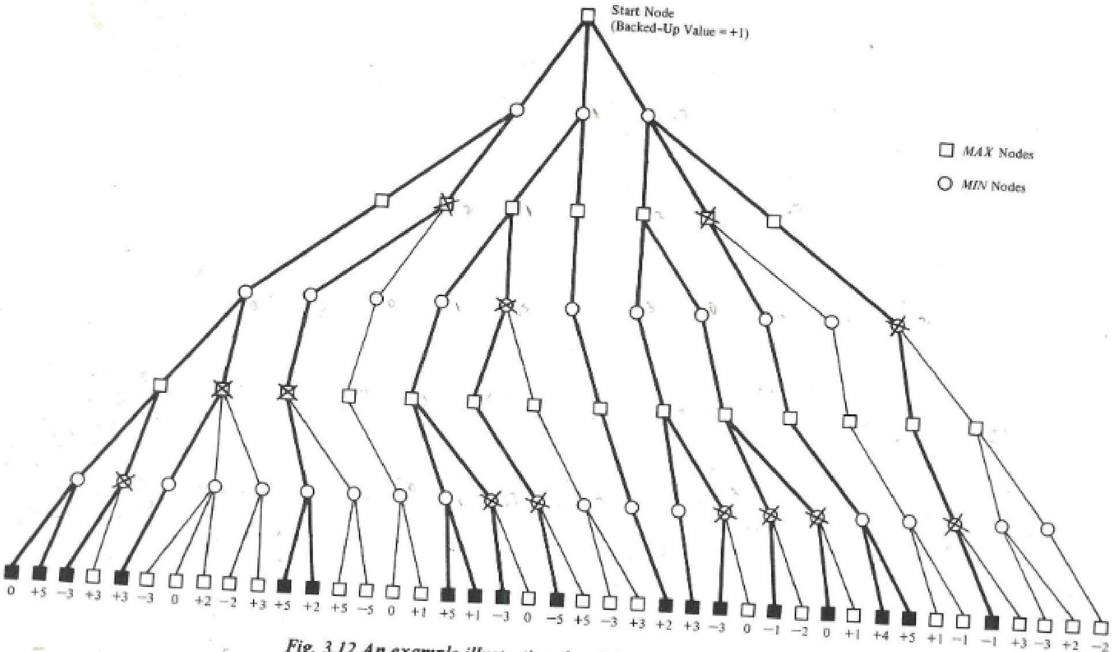
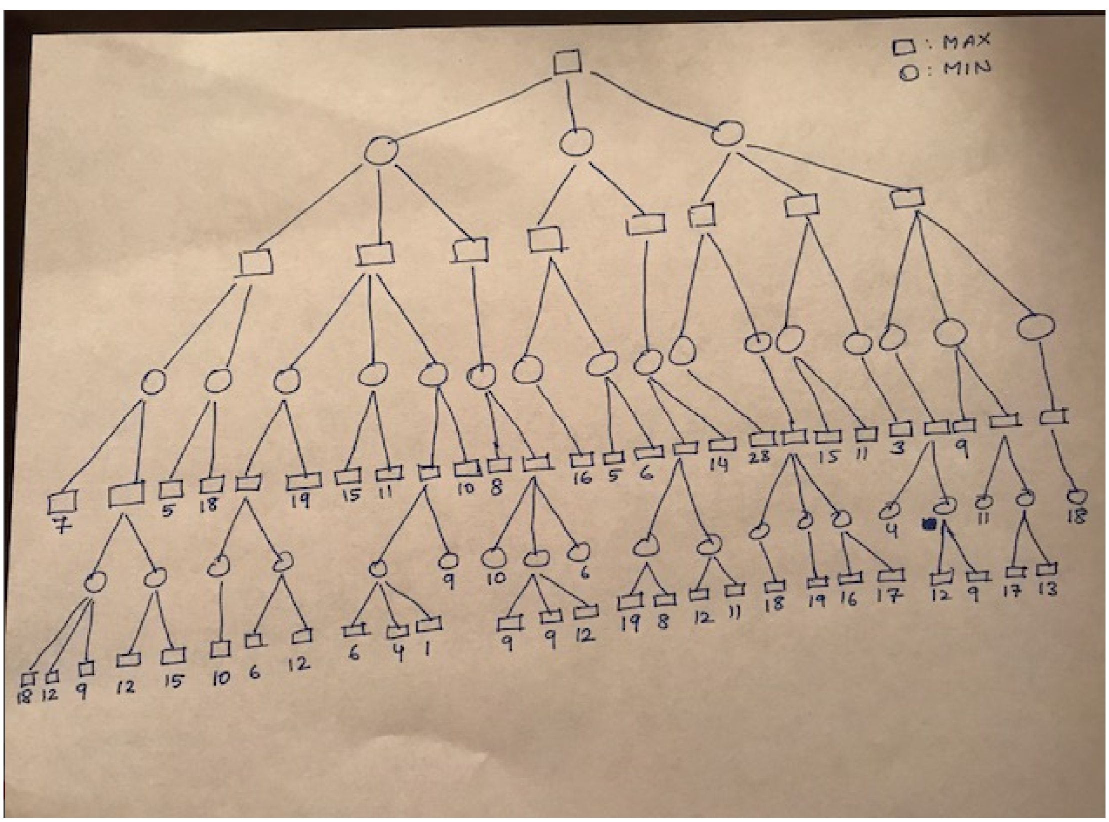


Fig. 3.12 An example illustrating the alpha-beta search procedure.



Propositional Logic

$\neg$ (not). A sentence such as $\neg W_{1,3}$ is called the **negation** of $W_{1,3}$. A **literal** is either an atomic sentence (a **positive literal**) or a negated atomic sentence (a **negative literal**).

$\wedge$ (and). A sentence whose main connective is $\wedge$, such as $W_{1,3} \wedge P_{3,1}$, is called a **conjunction**; its parts are the **conjuncts**. (The $\wedge$ looks like an "A" for "And.")

$\vee$ (or). A sentence using $\vee$, such as $(W_{1,3} \wedge P_{3,1}) \vee W_{2,2}$, is a **disjunction** of the **disjuncts** $(W_{1,3} \wedge P_{3,1})$ and $W_{2,2}$. (Historically, the $\vee$ comes from the Latin "vel," which means "or." For most people, it is easier to remember $\vee$ as an upside-down $\wedge$.)

$\Rightarrow$ (implies). A sentence such as $(W_{1,3} \wedge P_{3,1}) \Rightarrow \neg W_{2,2}$ is called an **implication** (or conditional). Its **premise** or **antecedent** is $(W_{1,3} \wedge P_{3,1})$, and its **conclusion** or **consequent** is $\neg W_{2,2}$. Implications are also known as **rules** or **if–then** statements. The implication symbol is sometimes written in other books as $\supset$ or $\rightarrow$.

$\Leftrightarrow$ (if and only if). The sentence $W_{1,3} \Leftrightarrow \neg W_{2,2}$ is a **biconditional**. Some other books write this as $\equiv$.

- $\neg P$ is true iff P is false in m.
- $P \wedge Q$ is true iff both P and Q are true in m.
- $P \vee Q$ is true iff either P or Q is true in m.
- $P \Rightarrow Q$ is true unless P is true and Q is false in m.
- $P \Leftrightarrow Q$ is true iff P and Q are both true or both false in m.

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
false	*false*	*true*	*false*	*false*	*true*	*true*
false	*true*	*true*	*false*	*true*	*true*	*false*
true	*false*	*false*	*false*	*true*	*false*	*false*
true	*true*	*false*	*true*	*true*	*true*	*true*

$$
\begin{aligned}
(\alpha \wedge \beta) &\equiv (\beta \wedge \alpha) \quad \text{commutativity of } \wedge\\
(\alpha \vee \beta) &\equiv (\beta \vee \alpha) \quad \text{commutativity of } \vee\\
((\alpha \wedge \beta) \wedge \gamma) &\equiv (\alpha \wedge (\beta \wedge \gamma)) \quad \text{associativity of } \wedge\\
((\alpha \vee \beta) \vee \gamma) &\equiv (\alpha \vee (\beta \vee \gamma)) \quad \text{associativity of } \vee\\
\neg(\neg \alpha) &\equiv \alpha \quad \text{double-negation elimination}\\
(\alpha \Rightarrow \beta) &\equiv (\neg \beta \Rightarrow \neg \alpha) \quad \text{contraposition}\\
(\alpha \Rightarrow \beta) &\equiv (\neg \alpha \vee \beta) \quad \text{implication elimination}\\
(\alpha \Leftrightarrow \beta) &\equiv ((\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)) \quad \text{biconditional elimination}\\
\neg(\alpha \wedge \beta) &\equiv (\neg \alpha \vee \neg \beta) \quad \text{De Morgan}\\
\neg(\alpha \vee \beta) &\equiv (\neg \alpha \wedge \neg \beta) \quad \text{De Morgan}\\
(\alpha \wedge (\beta \vee \gamma)) &\equiv ((\alpha \wedge \beta) \vee (\alpha \wedge \gamma)) \quad \text{distributivity of } \wedge \text{ over } \vee\\
(\alpha \vee (\beta \wedge \gamma)) &\equiv ((\alpha \vee \beta) \wedge (\alpha \vee \gamma)) \quad \text{distributivity of } \vee \text{ over } \wedge
\end{aligned}
$$

Figure 7.11 Standard logical equivalences. The symbols α, β, and γ stand for arbitrary sentences of propositional logic.

This section covers **inference rules** that can be applied to derive a **proof**—a chain of conclusions that leads to the desired goal. The best-known rule is called **Modus Ponens** (Latin for *mode that affirms*) and is written

$$\frac{\alpha \ \Rightarrow\ \beta, \qquad \alpha}{\beta}\,.$$

The notation means that, whenever any sentences of the form $\alpha \Rightarrow \beta$ and α are given, then the sentence β can be inferred. For example, if $(\mathit{WumpusAhead} \land \mathit{WumpusAlive}) \Rightarrow \mathit{Shoot}$ and $(\mathit{WumpusAhead} \land \mathit{WumpusAlive})$ are given, then Shoot can be inferred.

Another useful inference rule is **And-Elimination**, which says that, from a conjunction, any of the conjuncts can be inferred:

$$\frac{\alpha \land \beta}{\alpha}\,.$$

Now that we have a notion of truth, we are ready to talk about logical reasoning. This involves the relation of logical **entailment** between sentences—the idea that a sentence *follows logically* from another sentence. In mathematical notation, we write

$$\alpha \models \beta$$

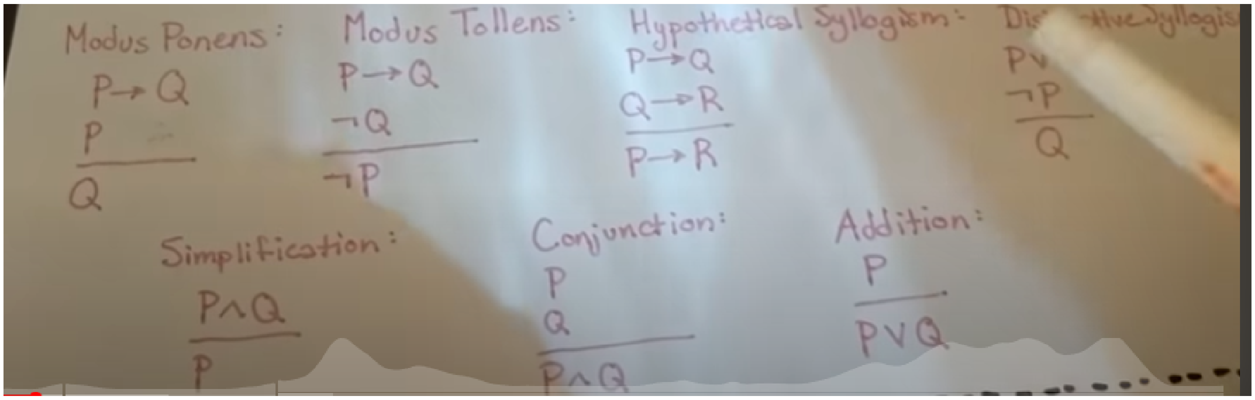
to mean that the sentence α entails the sentence β. The formal definition of entailment is this: $\alpha \models \beta$ if and only if, in every model in which α is true, β is also true. Using the notation just introduced, we can write

$$\alpha \models \beta \text{ if and only if } M(\alpha) \subseteq M(\beta)\,.$$

$$\begin{array}{rcl}
\mathit{Sentence} & \rightarrow & \mathit{AtomicSentence} \mid \mathit{ComplexSentence} \\
\mathit{AtomicSentence} & \rightarrow & \mathit{True} \mid \mathit{False} \mid P \mid Q \mid R \mid \ldots \\
\mathit{ComplexSentence} & \rightarrow & (\ \mathit{Sentence}\ ) \mid [\ \mathit{Sentence}\ ] \\
& \mid & \neg\, \mathit{Sentence} \\
& \mid & \mathit{Sentence} \land \mathit{Sentence} \\
& \mid & \mathit{Sentence} \lor \mathit{Sentence} \\
& \mid & \mathit{Sentence} \ \Rightarrow\ \mathit{Sentence} \\
& \mid & \mathit{Sentence} \ \Leftrightarrow\ \mathit{Sentence} \\
\\
\text{OPERATOR PRECEDENCE} & : & \neg, \land, \lor, \Rightarrow, \Leftrightarrow
\end{array}$$

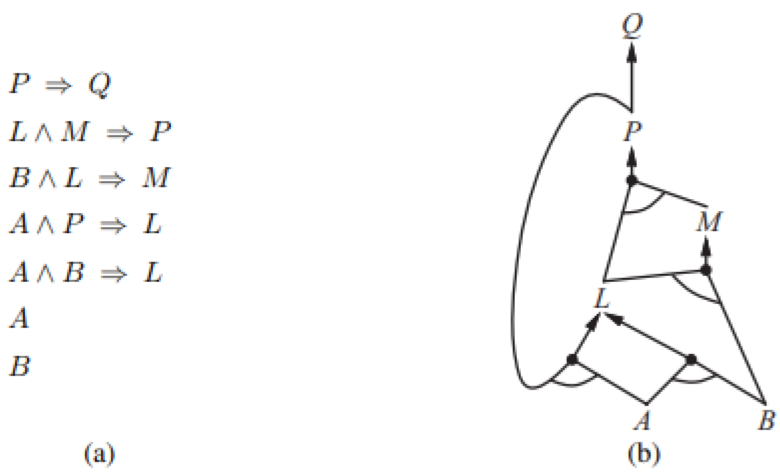
Thus, the unit resolution rule takes a **clause**—a disjunction of literals–

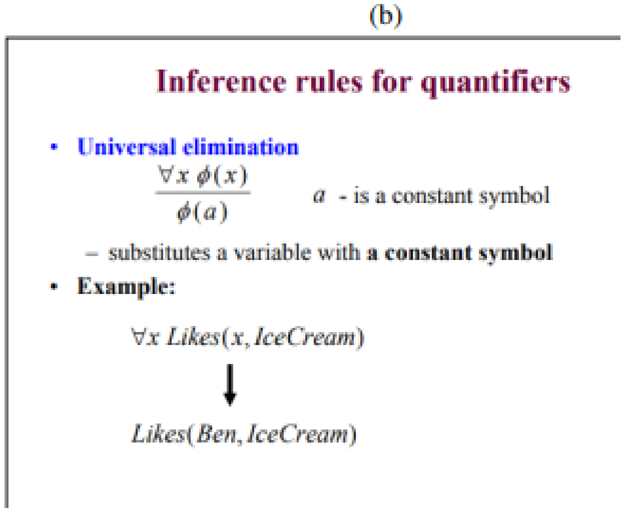
$$\begin{array}{rcl}
\mathit{CNFSentence} & \rightarrow & \mathit{Clause}_1 \land \cdots \land \mathit{Clause}_n \\
\mathit{Clause} & \rightarrow & \mathit{Literal}_1 \lor \cdots \lor \mathit{Literal}_m \\
\mathit{Literal} & \rightarrow & \mathit{Symbol} \mid \neg \mathit{Symbol} \\
\mathit{Symbol} & \rightarrow & P \mid Q \mid R \mid \ldots \\
\mathit{HornClauseForm} & \rightarrow & \mathit{DefiniteClauseForm} \mid \mathit{GoalClauseForm} \\
\mathit{DefiniteClauseForm} & \rightarrow & (\mathit{Symbol}_1 \land \cdots \land \mathit{Symbol}_l) \ \Rightarrow\ \mathit{Symbol} \\
\mathit{GoalClauseForm} & \rightarrow & (\mathit{Symbol}_1 \land \cdots \land \mathit{Symbol}_l) \ \Rightarrow\ \mathit{False}
\end{array}$$



Forward Chaining

- proving the goal starting with the facts and using the sentences without modifying them
- forward chaining just uses the implications that are given in the knowledge base whereas with resolution you break all implications into CNF and then resolve them from there as you can





Backward Chaining

- Same as forward chaining except starting at the goal
- Goal-directed reasoning

CNF

Conjunctive normal form

The resolution rule applies only to clauses (that is, disjunctions of literals), so it would seem to be relevant only to knowledge bases and queries consisting of clauses. How, then, can it lead to a complete inference procedure for all of propositional logic? The answer is that *every sentence of propositional logic is logically equivalent to a conjunction of clauses.* A sentence expressed as a conjunction of clauses is said to be in **conjunctive normal form** or **CNF** (see Figure 7.14). We now describe a procedure for converting to CNF. We illustrate the procedure by converting the sentence $B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$ into CNF. The steps are as follows:

1. Eliminate $\Leftrightarrow$, replacing $\alpha \Leftrightarrow \beta$ with $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$.

   $$(B_{1,1} \Rightarrow (P_{1,2} \vee P_{2,1})) \wedge ((P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1}) .$$

2. Eliminate $\Rightarrow$, replacing $\alpha \Rightarrow \beta$ with $\neg\alpha \vee \beta$:

   $$(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg(P_{1,2} \vee P_{2,1}) \vee B_{1,1}) .$$

3. CNF requires $\neg$ to appear only in literals, so we "move $\neg$ inwards" by repeated application of the following equivalences from Figure 7.11:

   $\neg(\neg\alpha) \equiv \alpha$ (double-negation elimination)
   $\neg(\alpha \wedge \beta) \equiv (\neg\alpha \vee \neg\beta)$ (De Morgan)
   $\neg(\alpha \vee \beta) \equiv (\neg\alpha \wedge \neg\beta)$ (De Morgan)

   In the example, we require just one application of the last rule:

   $$(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge ((\neg P_{1,2} \wedge \neg P_{2,1}) \vee B_{1,1}) .$$

4. Now we have a sentence containing nested $\wedge$ and $\vee$ operators applied to literals. We apply the distributivity law from Figure 7.11, distributing $\vee$ over $\wedge$ wherever possible.

   $$(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg P_{1,2} \vee B_{1,1}) \wedge (\neg P_{2,1} \vee B_{1,1}) .$$

The original sentence is now in CNF, as a conjunction of three clauses. It is much harder to read, but it can be used as input to a resolution procedure.

Resolution/Resolution-Refutation

- First break knowledge base into CNF clauses
- Resolution rule takes a clause- a disjunction of literals- and a literal (at least) and produces a new clause

$$\frac{P_{1,1} \lor P_{3,1}, \qquad \neg P_{1,1} \lor \neg P_{2,2}}{P_{3,1} \lor \neg P_{2,2}}$$

$(A \lor B \lor \neg C) \land (B \lor D) \land (\neg A) \land (B \lor C)$

- $(A \lor B \lor \neg C)$ is a clause, which is a disjunction of literals
- A, B, and ¬ C are literals

There is one more technical aspect of the resolution rule: the resulting clause should contain only one copy of each literal.[9] The removal of multiple copies of literals is called **factoring**. For example, if we resolve $(A \lor B)$ with $(A \lor \neg B)$, we obtain $(A \lor A)$, which is reduced to just A.

- Resolution rule:

$$\frac{\begin{array}{l}\alpha \lor \beta \\ \neg\beta \lor \gamma\end{array}}{\alpha \lor \gamma}$$

- Resolution refutation:
  - Convert all sentences to CNF
  - Negate the desired conclusion (converted to CNF)
  - Apply resolution rule until either
    - Derive false (a contradiction)
    - Can't apply any more
- Resolution refutation is sound and complete
  - If we derive a contradiction, then the conclusion follows from the axioms
  - If we can't apply any more, then the conclusion cannot be proved from the axioms.

Prove R

1	P v Q
2	P → R
3	Q → R

false v R
¬ R v false

~~false v false~~

Step	Formula	Derivation
1	P v Q	Given
2	¬ P v R	Given
3	¬ Q v R	Given
4	¬ R	Negated conclusion
5	Q v R	1,2
6	¬ P	2,4
7	¬ Q	3,4
8	R	5,7
9	•	4,8

FOPC

$$
\begin{aligned}
\textit{Quantifier} &\rightarrow \forall \mid \exists \\
\textit{Constant} &\rightarrow A \mid X_1 \mid \textit{John} \mid \cdots \\
\textit{Variable} &\rightarrow a \mid x \mid s \mid \cdots \\
\textit{Predicate} &\rightarrow \textit{True} \mid \textit{False} \mid \textit{After} \mid \textit{Loves} \mid \textit{Raining} \mid \cdots \\
\textit{Function} &\rightarrow \textit{Mother} \mid \textit{LeftLeg} \mid \cdots
\end{aligned}
$$

8.2.5 Complex sentences

We can use **logical connectives** to construct more complex sentences, with the same syntax and semantics as in propositional calculus. Here are four sentences that are true in the model of Figure 8.2 under our intended interpretation:

$$
\begin{aligned}
&\neg Brother(LeftLeg(Richard), John) \\
&Brother(Richard, John) \land Brother(John, Richard) \\
&King(Richard) \lor King(John) \\
&\neg King(Richard) \Rightarrow King(John) .
\end{aligned}
$$

Universal quantification ($\forall$)

Recall the difficulty we had in Chapter 7 with the expression of general rules in propositional logic. Rules such as "Squares neighboring the wumpus are smelly" and "All kings are persons" are the bread and butter of first-order logic. We deal with the first of these in Section 8.3. The second rule, "All kings are persons," is written in first-order logic as

$$\forall\, x \;\; King(x) \;\Rightarrow\; Person(x)\,.$$

$\forall$ is usually pronounced "For all ...". (Remember that the upside-down A stands for "all.") Thus, the sentence says, "For all x, if x is a king, then x is a person." The symbol x is called a **variable**. By convention, variables are lowercase letters. A variable is a term all by itself, and as such can also serve as the argument of a function—for example, $LeftLeg(x)$. A term with no variables is called a **ground term**.

Existential quantification ($\exists$)

Universal quantification makes statements about every object. Similarly, we can make a statement about *some* object in the universe without naming it, by using an existential quantifier. To say, for example, that King John has a crown on his head, we write

$$\exists\, x \;\; Crown(x) \wedge OnHead(x, John)\,.$$

$\exists x$ is pronounced "There exists an x such that ..." or "For some x ...".

Intuitively, the sentence $\exists\, x \;\; P$ says that P is true for at least one object x. More precisely, $\exists\, x \;\; P$ is true in a given model if P is true in *at least one* extended interpretation that assigns x to a domain element. That is, at least one of the following is true:

We will often want to express more complex sentences using multiple quantifiers. The simplest case is where the quantifiers are of the same type. For example, "Brothers are siblings" can be written as

$$\forall\, x \;\; \forall\, y \;\; Brother(x, y) \;\Rightarrow\; Sibling(x, y)\,.$$

Converting English Sentences

- Every gardener likes the sun.
  `(Ax) gardener(x) => likes(x,Sun)`
- You can fool some of the people all of the time
  `(Ex)(At) (person(x) ^ time(t)) => can-be-fooled(x,t`
- You can fool all of the people some of the time.
  `(Ax)(Et) (person(x) ^ time(t) => can-be-fooled(x,t)`

- Deb is not tall.
  `~tall(Deb)`

- All purple mushrooms are poisonous.
  `(Ax) (mushroom(x) ^ purple(x)) => poisonous(x)`
- No purple mushroom is poisonous.
  `~(Ex) purple(x) ^ mushroom(x) ^ poisonous(x)`
  or, equivalently,
  `(Ax) (mushroom(x) ^ purple(x)) => ~poisonous(x)`

All students are smart.
$\forall x\ (Student\ (x) \Rightarrow Smart\ (x))$

There exists a student.
$\exists x\ Student(x)$

There exists a smart student.
$\exists x\ (Student(x) \land Smart\ (x))$

Every student loves some student.
$\forall x\ (Student(x) \Rightarrow \exists y\ (Student(y) \land Loves(x,y)))$

Every student loves some other student.
$\forall x\ (Student(x) \Rightarrow \exists y\ (Student(y) \land \neg(x=y) \land Loves(x,y)))$

There is a student who is loved by every other student.
$\exists x\ (Student(x) \land \forall y\ (Student(y) \land \neg(x=y) \Rightarrow Loves(y,x)))$

Bill is a student.
$Student(Bill)$

Bill takes either Analysis or Geometry (but not both).
$Takes(Bill,Analysis) \Leftrightarrow \neg Takes(Bill,Geometry)$

Bill takes Analysis or Geometry (or both).
$Takes(Bill,Analysis) \lor Takes(Bill,Geometry)$

Bill takes Analysis and Geometry.
$Takes(Bill,Analysis) \land Takes(Bill,Geometry)$

Bill does not take Analysis.
$\neg Takes(Bill,Analysis)$

No student loves Bill.
$\neg\exists x\ (Student(x) \land Loves(x,Bill))$

Bill has at least one sister.
$\exists x\ SisterOf(x,Bill)$

Bill has no sister.
$\neg\exists x$ SisterOf(x,Bill)

Bill has at most one sister.
$\forall x\ \forall y$ (SisterOf(x,Bill) $\wedge$ SisterOf(y,Bill) $\Rightarrow$ x=y)

Bill has exactly one sister.
$\exists x$ (SisterOf(x,Bill) $\wedge$ $\forall y$ (SisterOf(y,Bill) $\Rightarrow$ x=y))

Bill has at least two sisters
$\exists x\ \exists y$ (SisterOf(x,Bill) $\wedge$ (SisterOf(y,Bill) $\wedge$ $\neg$(x=y))

Every student takes at least one course.
$\forall x$ (Student(x) $\Rightarrow$ $\exists y$ (Course(y) $\wedge$ Takes(x,y)))

Only one student failed History.
$\exists x$ (Student(x) $\wedge$ Failed(x,History) $\wedge$ $\forall y$ (Student(y) $\wedge$ Failed(y,History) $\Rightarrow$ x=y))

No student failed Chemistry, but at least one student failed History.
$\neg\exists x$ (Student(x) $\wedge$ Failed(x,Chemistry)) $\wedge$ $\exists x$ (Student(x) $\wedge$ Failed (x,History))

Every student who takes Analysis also takes Geometry.
$\forall x$ (Student(x) $\wedge$ Takes(x,Analysis) $\Rightarrow$ Takes(x,Geometry))

No student can fool all the other students.
$\neg\exists x$ (Student(x) $\wedge$ $\forall y$ (Student(y) $\wedge$ $\neg$(x=y) $\Rightarrow$ Fools(x,y)))

a. There is at least one student in every course who attends every class session of the course.

For s students, c courses, and cs class_sessions

∃s∀c∀cs (student(s) ^ course(c) ^ class_session(cs) => attends(s, cs))

b. Every country has at least one citizen who has visited every neighboring country of his own country.

For x countries, y citizens, and z neighbors

∀x∃y∀z (country(x) ^ citizen(y,x) ^ neighbors(z, x) => visited(y, z))

c. Every computer made by every manufacturer can host either windows operating system or Mac operating system.

For c computer and y manufacturers

∀c∀y (Computer(c) ^ Manufacturer(y,c) -> windows(c) V mac(c))

Conversion to CNF and Clause Form

First Order Logic: Conversion to CNF

1. Eliminate biconditionals and implications:
   - Eliminate $\Leftrightarrow$, replacing $\alpha \Leftrightarrow \beta$ with $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$.
   - Eliminate $\Rightarrow$, replacing $\alpha \Rightarrow \beta$ with $\neg\alpha \vee \beta$.
2. Move $\neg$ inwards:
   - $\neg(\forall x \ \ p) \equiv \exists x \ \ \neg p$,
   - $\neg(\exists x \ \ p) \equiv \forall x \ \ \neg p$,
   - $\neg(\alpha \vee \beta) \equiv \neg\alpha \wedge \neg\beta$,
   - $\neg(\alpha \wedge \beta) \equiv \neg\alpha \vee \neg\beta$,
   - $\neg\neg\alpha \equiv \alpha$.
3. Standardize variables apart by renaming them: each quantifier should use a different variable.
4. Skolemize: each existential variable is replaced by a *Skolem constant* or *Skolem function* of the enclosing universally quantified variables.
   - For instance, $\exists x\, Rich(x)$ becomes $Rich(G1)$ where $G1$ is a new Skolem constant.
   - "Everyone has a heart" $\forall x \ \ Person(x) \Rightarrow \exists y \ \ Heart(y) \wedge Has(x, y)$ becomes $\forall x \ \ Person(x) \Rightarrow Heart(H(x)) \wedge Has(x, H(x))$, where H is a new symbol (Skolem function).
5. Drop universal quantifiers
   - For instance, $\forall x \ \ Person(x)$ becomes $Person(x)$.
6. Distribute $\wedge$ over $\vee$:
   - $(\alpha \wedge \beta) \vee \gamma \equiv (\alpha \vee \gamma) \wedge (\beta \vee \gamma)$.

∃s∀c∀cs (¬student(s) V ¬course(c) V ¬class_session(cs) V attends(s, cs))

CLAUSES:

C1: ¬student(s) V ¬course(c) V ¬class_session(cs) V attends(s, cs)

CNF: eats(Tim, Cake) Λ ¬sick(Tim, Cake)

CLAUSES: C5: eats(Tim, Cake); C6: ¬sick(Tim, Cake)

Relationships of ∀ x to ∃ x

Connections between ∀ and ∃

The two quantifiers are actually intimately connected with each other, through negation. Asserting that everyone dislikes parsnips is the same as asserting there does not exist someone who likes them, and vice versa:

$$\forall x \;\; \neg Likes(x, Parsnips) \quad \text{is equivalent to} \quad \neg\exists x \;\; Likes(x, Parsnips)\,.$$

We can go one step further: "Everyone likes ice cream" means that there is no one who does not like ice cream:

$$\forall x \;\; Likes(x, IceCream) \quad \text{is equivalent to} \quad \neg\exists x \;\; \neg Likes(x, IceCream)\,.$$

Because ∀ is really a conjunction over the universe of objects and ∃ is a disjunction, it should not be surprising that they obey De Morgan's rules. The De Morgan rules for quantified and unquantified sentences are as follows:

$$\begin{array}{llllll}
\forall x \;\; \neg P & \equiv & \neg\exists x \;\; P & \qquad \neg(P \lor Q) & \equiv & \neg P \land \neg Q \\
\neg\forall x \;\; P & \equiv & \exists x \;\; \neg P & \qquad \neg(P \land Q) & \equiv & \neg P \lor \neg Q \\
\forall x \;\; P & \equiv & \neg\exists x \;\; \neg P & \qquad P \land Q & \equiv & \neg(\neg P \lor \neg Q) \\
\exists x \;\; P & \equiv & \neg\forall x \;\; \neg P & \qquad P \lor Q & \equiv & \neg(\neg P \land \neg Q)\,.
\end{array}$$

Thus, we do not really need both ∀ and ∃, just as we do not really need both ∧ and ∨. Still, readability is more important than parsimony, so we will keep both of the quantifiers.

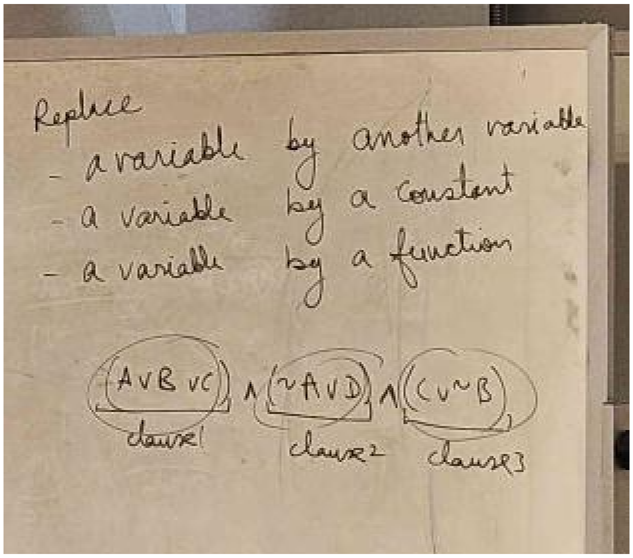
Unification

- anything lowercase is a variable and uppercase is a constant, you can exchange variables with other variables, constants, and functions but constants and functions cannot be changed
- Unify is a linear time algorithm that returns the most general unifier (mgu), i.e., a shortest length substitution list that makes the two literals match. (In general, there is not a unique minimum length substitution list, but unify returns one of those of minimum length.)
- A variable can never be replaced by a term containing that variable. For example, x/f(x) is illegal. This "occurs check" should be done in the above pseudo-code before making the recursive calls.

$$\begin{aligned}
&\textsc{Unify}(Knows(John, x),\; Knows(John, Jane)) = \{x/Jane\} \\
&\textsc{Unify}(Knows(John, x),\; Knows(y, Bill)) = \{x/Bill, y/John\} \\
&\textsc{Unify}(Knows(John, x),\; Knows(y, Mother(y))) = \{y/John, x/Mother(John)\} \\
&\textsc{Unify}(Knows(John, x),\; Knows(x, Elizabeth)) = fail\,.
\end{aligned}$$

The last unification fails because x cannot take on the values $John$ and $Elizabeth$ at the same time. Now, remember that $Knows(x, Elizabeth)$ means "Everyone knows Elizabeth," so we *should* be able to infer that John knows Elizabeth. The problem arises only because the two sentences happen to use the same variable name, x. The problem can be avoided by **standardizing apart** one of the two sentences being unified, which means renaming its variables to avoid name clashes. For example, we can rename x in $Knows(x, Elizabeth)$ to x_{17} (a new variable name) without changing its meaning. Now the unification will work:

$$\textsc{Unify}(Knows(John, x),\; Knows(x_{17}, Elizabeth)) = \{x/Elizabeth, x_{17}/John\}\,.$$



ω_1	ω_2	MGU
P(x)	P(A)	{x/A}
P(F(x), y, G(x))	P(F(x), x, G(x))	{y/x} or {x/y}
P(F(x), y, G(y))	P(F(x), z, G(x))	{y/x, z/x}
P(x, B, B)	P(A, y, z)	{x/A, y/B, z/B}
P(G(F(v)), G(u))	P(x, x)	{x/G(F(v)), u/F(v)}
P(x, F(x))	P(x, x)	No MGU!

- Examples

Literal 1	**Literal 2**	**Result of Unify**
`parents(x, father(x), mother(Bill))`	`parents(Bill, father(Bill), y)`	`{x/Bill, y/mother(Bill)}`
`parents(x, father(x), mother(Bill))`	`parents(Bill, father(y), z)`	`{x/Bill, y/Bill, z/mother(Bill)}`
`parents(x, father(x), mother(Jane))`	`parents(Bill, father(y), mother(y))`	`Failure`

Forward Chaining Proof (Tom likes Cake)

CLAUSES:

C1: ∀x(Food(x) => Likes(x,Tom))

C2: Food(Apple Pie)

C3: Food(Ice Creme)

C4: ∀x∀y((eats(y, x) ∧ ¬sick(y, x)) => Food(x))

C5: ∀x(eats(Tim, Cake)

C6: ¬sick(Tim, Cake))

C7: ∀x(eats(Tim, x) => eats(Mary, x))

PROOF

C5: ∀x(eats(Tom, Cake)

C6: ¬sick(Tom, Cake))

C5 and C6 so by C4, Food(Cake) is True Unify:{(Tom/y, Cake/x)}

Food(Cake), so by C1, Likes(Cake, Tom)

Resolution/Resolution-Refutation

- Convert to CNF and use clauses to prove
- Refutation: negate goal state

$$\frac{l_1 \vee \ldots\ldots \vee l_k, \qquad m_1 \vee \ldots\ldots \vee m_n}{\text{SUBST}(\theta, l_1 \vee \ldots\ldots \vee l_{i-1} \vee l_{i+1} \vee \ldots \vee l_k \vee m_1 \vee \ldots\ldots \vee m_{j-1} \vee m_{j+1} \vee \ldots\ldots \vee m_n)}$$

Where $\mathbf{l_i}$ and $\mathbf{m_j}$ are complementary literals.

This rule is also called the **binary resolution rule** because it only resolves exactly two literals.

Example:

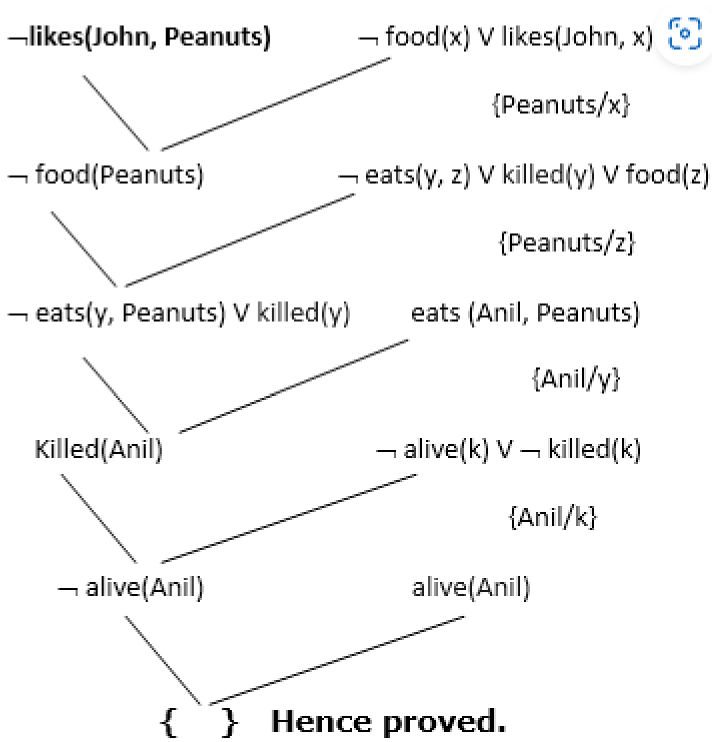
We can resolve two clauses which are given below:

[Animal (g(x) V Loves (f(x), x)] **and** **[¬ Loves(a, b) V ¬Kills(a, b)]**

Where two complimentary literals are: **Loves (f(x), x) and ¬ Loves (a, b)**

These literals can be unified with unifier **θ= [a/f(x), and b/x]** , and it will generate a resolvent clause:

[Animal (g(x) V ¬ Kills(f(x), x)].



- Tom likes Cake example for Resolution

CLAUSES:

C1: ¬Food(x) V Likes(x,Tom)

C2: Food(Apple Pie)

C3: Food(Ice Creme)

C4: ¬eats(y, x) V sick(y, x) V Food(x)

C5: eats(Tim, Cake)

C6: ¬sick(Tim, Cake)

C7: ¬eats(Tim, x) V eats(Mary, x)

PROOF

C5: eats(Tim, Cake)

C6: ¬sick(Tim, Cake)

C8: Food(Cake). (C5, C6, and C4, resolve) Unify:{(Tim/y, Cake/x)}

C9: Likes(Cake,Tom) (C8 and C1, resolve)

- Tom likes Cake example for Resolution-Refutation

  Goal: Likes(Cake,Tom)

  CLAUSES:

  C1: ¬Food(x) V Likes(x,Tom)

  C2: Food(Apple Pie)

  C3: Food(Ice Creme)

  C4: ¬eats(y, x) V sick(y, x) V Food(x)

  C5: eats(Tim, Cake)

  C6: ¬sick(Tim, Cake)

  C7: ¬eats(Tim, x) V eats(Mary, x)

  C8 [GOAL NEGATION]: ¬Likes(Cake,Tom)

  PROOF

  C5: eats(Tim, Cake)

  C6: ¬sick(Tim, Cake)

  C9: Food(Cake). (C5, C6, and C4, resolve) Unify:{(Tim/y, Cake/x)}

  C10: Likes(Cake,Tom) (C8 and C1, resolve)

  C10 Conflicts with C8, therefore the Goal (Likes(Cake,Tom)) is true.

Joint Probability Distribution and Conditional Prob.

		Color: Blue		Color: Non-Blue	
		Year: <2010	Year: >=2010	Year: <2010	Year: >=2010
Make: Toyota	GPS: Yes	0.025; 0.040	0.011; 0.10	0.015; 0.022	0.015; 0.100
	GPS: No	0.033; 0.025	0.185; 0.016	0.030; 0.009	0.126; 0.001
Make: Honda	GPS: Yes	0.070; 0.030	0.075; 0.230	0.050; 0.060	0.170; 0.353
	GPS: No	0.035; 0.010	0.040; 0.000	0.020; 0.002	0.100; 0.002

a. CINCINNATI

i.

P(GPS: No)

= 0.033 + 0.185 + 0.030 + 0.126 + 0.035 + 0.040 + 0.020 + 0.1

= **0.569**

ii.

P(Year: < 2010 | GPS: Yes)

P(GPS: Yes) = 1 - 0.569 = 0.431

P(Year: < 2010 and GPS: Yes) = 0.025 + 0.015 + 0.070 + 0.050 = 0.16

So P(Year: < 2010 | GPS: Yes) = 0.16 / 0.431 = **0.37123**

iii.

P(Make: Honda | Year: >= 2010 and GPS: No)

P(Year >= 2010 and GPS: No) = 0.185 + 0.126 + 0.040 + 0.100 = 0.451

P(Make: Honda and Year >= 2010 and GPS: No) = 0.040 + 0.100 = 0.140

So P(Make: Honda | Year: >= 2010 and GPS: No) = 0.140 / 0.451 = **0.310241**

b. FOR NEW YORK

i.

P(Make: Toyota | GPS: Yes)

P(GPS: Yes) = 0.040 + 0.1 + 0.022 + 0.1 + 0.030 + 0 .23+ 0.06 + 0.353

= 0.935

P(Make: Toyota and GPS: Yes) = 0.040 + 0.1 + 0.022 + 0.1 = 0.262

So P(Make: Toyota | GPS: Yes) = 0.262 / 0.935 = **0.28021**

ii.

P(Color: Blue | Year: <2010 and Make: Honda)

P(Year: <2010 and Make: Honda) = 0.03 + 0.06 + 0.01 + 0.002 = 0.102

P(Color: Blue and Year: <2010 and Make: Honda) = 0.03 + 0.01 = 0.04

So P(Color: Blue | Year: <2010 and Make: Honda) = 0.04 / 0.102 = **0.392157**

Iii.

P(GPS: No | Make: Honda)

P(Make: Honda) = 0.03 + 0.23 + 0.06 + 0.353 + 0.01 + 0.002 + 0.002 = 0.687

P(GPS: No and Make: Honda) = 0.01 + 0.002 + 0.002 = 0.014

So P(GPS: No | Make: Honda) = 0.014 / 0.687 = **0.020378**

c. FOR BOTH

i.

P(Make: Toyota) = (P(Toyota and Cincinnati) + P(Toyota and New York))/2

= (0.025 + 0.011 + 0.015 + 0.015 + 0.033 + 0.185 + 0.030 + 0.126

+ 0.04 + 0.1 + 0.022 + 0.1 + 0.025 + 0.016 + 0.009 + 0.001) / 2

= 0.753 / 2 = **0.3765**

ii.

P(Year: < 2010 | Color: Blue)

sum(P(Blue))/2 = (0.025 + 0.040 + 0.011 + 0.1 + 0.033 + 0.025 + 0.185 + 0.016

+ 0.07 + 0.03 + 0.075 + 0.23 + 0.035 + 0.01 + 0.04) / 2 = 0.925 / 2 = 0.4625

sum(P(Year: < 2010 and Color: Blue)) / 2= (0.025 + 0.040 + 0.033 + 0.025 + 0.07

+ 0.03 + 0.035 + 0.01) / 2 = 0.268 / 2 = 0.134

So P(Year: < 2010 | Color: Blue) = 0.134 / 0.4625 = **0.29**

iii.

P(Make:Toyota | Year: >= 2010 and GPS: Yes)

Sum(P(Year >= 2010 and GPS: Yes)) / 2

= (0.011 + 0.1 + 0.015 + 0.1 + 0.075 + 0.23 + 0.17 + 0.353) / 2 = 1.054 / 2 =

0.527

Sum(P(Make:Toyota and Year: >= 2010 and GPS: Yes) / 2 = (0.011 + 0.1 + 0.015 +

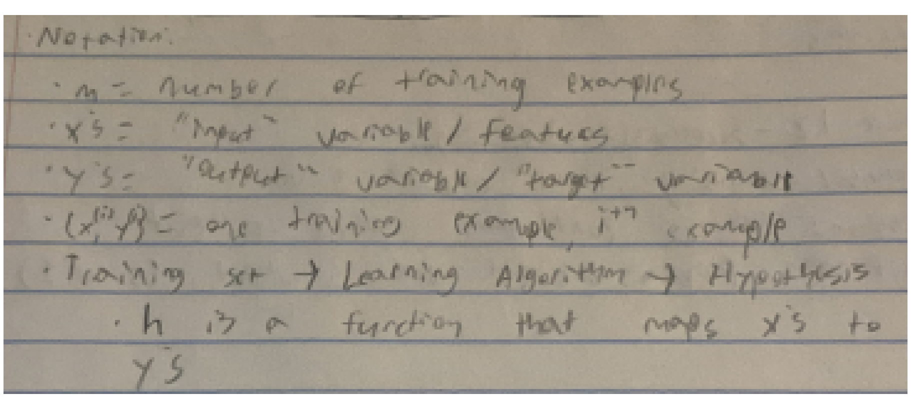
0.1) / 2 = 0.226 / 2 = 0.113

So P(Make:Toyota | Year: >= 2010 and GPS: Yes) = 0.113 / 0.527 = **0. 21442**

Intro to Machine Learning

- Tough to form a set definition for machine learning
- Arthur Samuel (1959): Machine learning is the field of study which gives computers the ability to learn without being explicitly programmed.

- Tom Mitchell (1998): Well-posed learning problem: A computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves from experience E
- Supervised Learning
    - “Right” answers given
    - Regression: Predict a continuous valued output that can be a spectrum of answers
    - Classification problem: Discrete set of possible outputs
- Unsupervised Learning
    - Data set with no labels
    - Let the algorithm create the types and structure
    - Algorithm not given the “right” answers
    - Examples:
        - Organize computing clusters
        - Social network analysis
        - Market segmentation
        - Astronomical data analysis
- Linear Regression with One Var Cost Function



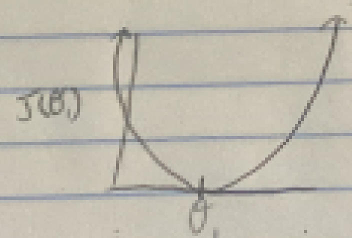
· Notation:
- m = number of training examples
- x's = "input" variable / features
- y's = "output" variable / "target" variable
- $(x^{(i)}, y^{(i)})$ = one training example, i^{th} example
- Training set → Learning Algorithm → Hypothesis
    - h is a function that maps x's to y's

- Hypothesis: $h_\theta(x) = \theta_0 + \theta_1 x$
- Idea: Chose θ_0, θ_1 so that $h_\theta(x)$ is close to y for our training examples

- minimize $\frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$
θ_0, θ_1
  - Squared error cost function

- Hypothesis: $h_\theta(x) = \theta_0 + \theta_1 x$
- Parameters: θ_0, θ_1
- Cost Function: $J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$

- Goal: minimize $J(\theta_0, \theta_1)$
- $J(\theta_1)$ (Function of the parameter θ_1)



- Chose value of θ_1 that minimizes $J(\theta_1)$

Perceptron Algo

- Some set of correct and not correct value (positive and negative in this case) is given and each is classified as positive or negative.
- X: given values with 1 added
- W: weight to be added
- Start with random weight vector, here its [0 -1 0 1].
- Take dot product to get X * W.

- If X*W is negative and the class for this entry is positive, there's an error. Similarly if X*W is positive and the class for this entry is negative, there's also an error.
- If error and X*W is negative, add X to the weight vector (W) which is the adjustment.If error and X*W is positive, subtract X from the weight vector (W) which is the adjustment.
- If no error, no adjustment
- Go back and redo the failed entries to have formula converge
- One Epoch= going through the list once