Artificial Intelligence

CMSC471/671
Section 0101
Tuesday/Thursday 5:30 – 6:45
Math-Psychology 103

Instructor: Professor Yun Peng
ECS Building Room 221
(410)455-3816
ypeng@cs.umbc.edu

Chapter 1: Introduction

- Can machines think?
- And if so, how?
- And if not, why not?
- And what does this say about human beings?
- And what does this say about the mind?

What is artificial intelligence?

- There are no clear consensus on the definition of AI
- Here's one from John McCarthy, (He coined the phrase AI in 1956) - see http:// www. formal. Stanford. EDU/ jmc/ whatisai/)

Q. What is artificial intelligence?

A. It is the science and engineering of making intelligent machines, especially intelligent computer programs. It is related to the similar task of using computers to understand human intelligence, but AI does not have to confine itself to methods that are biologically observable.

Q. Yes, but what is intelligence?

A. Intelligence is the computational part of the ability to achieve goals in the world. Varying kinds and degrees of intelligence occur in people, many animals and some machines.

Other possible AI definitions

- AI is a collection of hard problems which can be solved by humans and other living things, but for which we don't have good algorithms for solving.
  - e. g., understanding spoken natural language, medical diagnosis, circuit design, learning, self-adaptation, reasoning, chess playing, proving math theories, etc.
- Definition from R & N book: a program that
  - Acts like human (Turing test)
  - Thinks like human (human-like patterns of thinking steps)
  - Acts or thinks rationally (logically, correctly)
- Some problems used to be thought of as AI but are now considered not
  - e. g., compiling Fortran in 1955, symbolic mathematics in 1965, pattern recognition in 1970

What's easy and what's hard?

- It's been easier to mechanize many of the high level cognitive tasks we usually associate with "intelligence" in people
  - e. g., symbolic integration, proving theorems, playing chess, some aspect of medical diagnosis, etc.
- It's been very hard to mechanize tasks that animals can do easily
  - walking around without running into things
  - catching prey and avoiding predators
  - interpreting complex sensory information (visual, aural, …)
  - modeling the internal states of other animals from their behavior
  - working as a team (ants, bees)
- Is there a fundamental difference between the two categories?
- Why some complex problems (e.g., solving differential

History of AI

- AI has roots in a number of scientific disciplines
  - computer science and engineering (hardware and software)
  - philosophy (rules of reasoning)
  - mathematics (logic, algorithms, optimization)
  - cognitive science and psychology (modeling high level human/animal thinking)
  - neural science (model low level human/animal brain activity)
  - linguistics
- The birth of AI (1943 – 1956)
  - Pitts and McCulloch (1943): simplified mathematical model of neurons (resting/firing states) can realize all propositional logic primitives (can compute all Turing computable functions)
  - Allen Turing: Turing machine and Turing test (1950)
  - Claude Shannon: information theory; possibility of chess playing computers

- Early enthusiasm (1952 – 1969)
  - 1956 Dartmouth conference
    John McCarthy (Lisp);
    Marvin Minsky (first neural network machine);
    Alan Newell and Herbert Simon (GPS);
  - Emphasize on intelligent general problem solving
    GSP (means-ends analysis);
    Lisp (AI programming language);
    Resolution by John Robinson (basis for automatic theorem proving);
    heuristic search (A*, AO*, game tree search)
- Emphasis on knowledge (1966 – 1974)
  - domain specific knowledge is the key to overcome existing difficulties
  - knowledge representation (KR) paradigms
  - declarative vs. procedural representation

- Knowledge-based systems (1969 – 1999)
  - DENDRAL: the first knowledge intensive system (determining 3D structures of complex chemical compounds)
  - MYCIN: first rule-based expert system (containing 450 rules for diagnosing blood infectious diseases)
    EMYCIN: an ES shell
  - PROSPECTOR: first knowledge-based system that made significant profit (geological ES for mineral deposits)
- AI became an industry (1980 – 1989)
  - wide applications in various domains
  - commercially available tools
- Current trends (1990 – present)
  - more realistic goals
  - more practical (application oriented)
  - distributed AI and intelligent software agents
  - resurgence of neural networks and emergence of genetic algorithms

Chapter 2: Intelligent Agents

- Definition: An Intelligent Agent perceives it environment via sensors and acts rationally upon that environment with its effectors.
- Hence, an agent gets percepts one at a time, and maps this percept sequence to actions.
- Properties
  - Autonomous
  - Interacts with other agents plus the environment
  - Reactive to the environment
  - Pro-active (goal- directed)

Rationality

- An ideal **rational agent** should, for each possible percept sequence, do whatever actions that will maximize its performance measure based on

  (1) the percept sequence, and

  (2) its built-in and acquired knowledge.

- Hence it includes information gathering, not "rational ignorance."
- Rationality => Need a performance measure to say how well a task has been achieved.
- Types of performance measures: payoffs, false alarm and false dismissal rates, speed, resources required, effect on environment, etc.

Autonomy

- A system is autonomous to the extent that its own behavior is determined by its own experience and knowledge.
- To survive agents must have:
  - Enough built- in knowledge to survive.
  - Ability to learn.

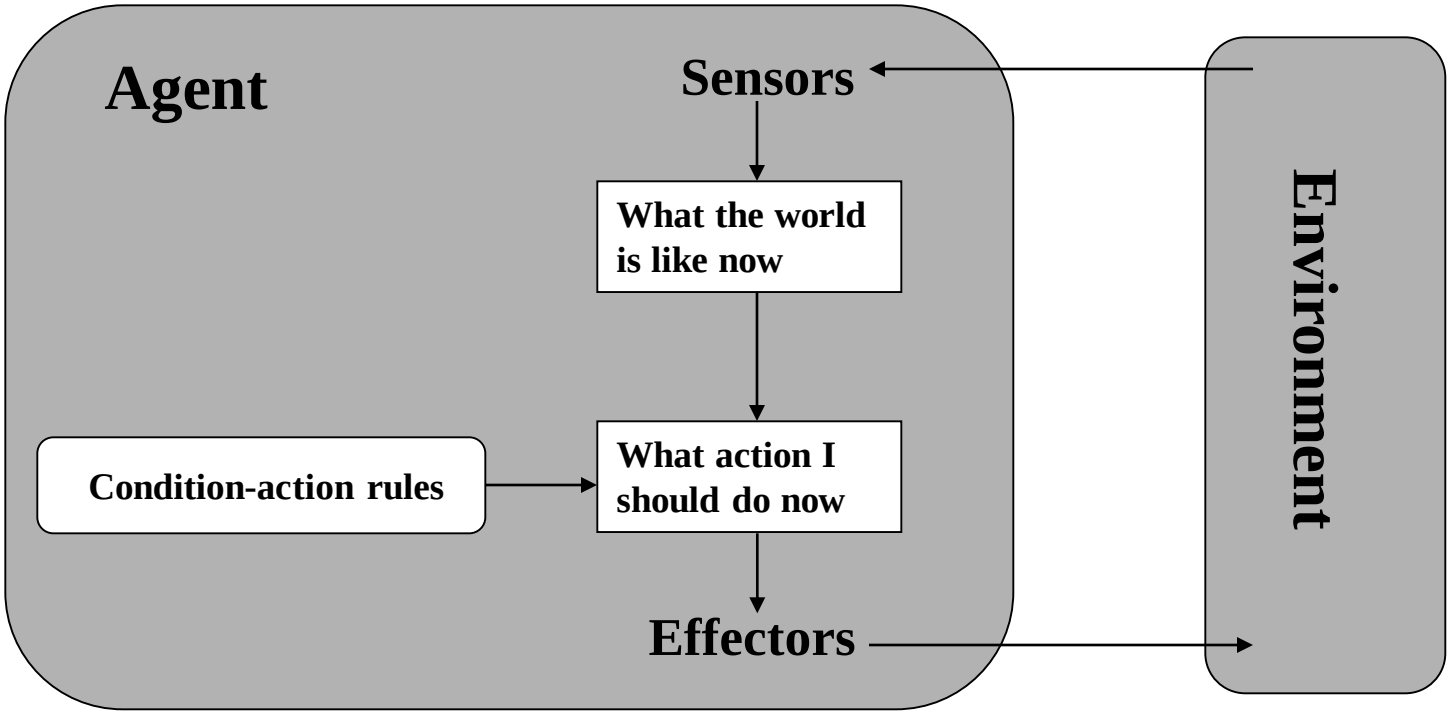
Some Agent Types

- **Table-driven agents**
  - use a percept sequence/ action table in memory to find the next action. They are implemented by a (large) lookup table.
- **Simple reflex agents**
  - are based on condition- action rules and implemented with an appropriate production (rule-based) system. They are stateless devices which do not have memory of past world states.
- **Agents with memory**
  - have internal state which is used to keep track of past states of the world.
- **Agents with goals**
  - are agents which in addition to state information have a kind of goal information which describes desirable situations. Agents of this kind take future events into consideration.
- **Utility-based agents**
  - base their decision on classic axiomatic utility-theory in order to act rationally.

Simple Reflex Agent

- **Table lookup** of percept- action pairs defining all possible condition- action rules necessary to interact in an environment
- **Problems**
  - Too big to generate and to store (Chess has about 10^120 states, for example)
  - No knowledge of non- perceptual parts of the current state
  - Not adaptive to changes in the environment; requires entire table to be updated if changes occur
- **Use *condition-action* rules to summarize portions of the table**

A Simple Reflex Agent: Schema



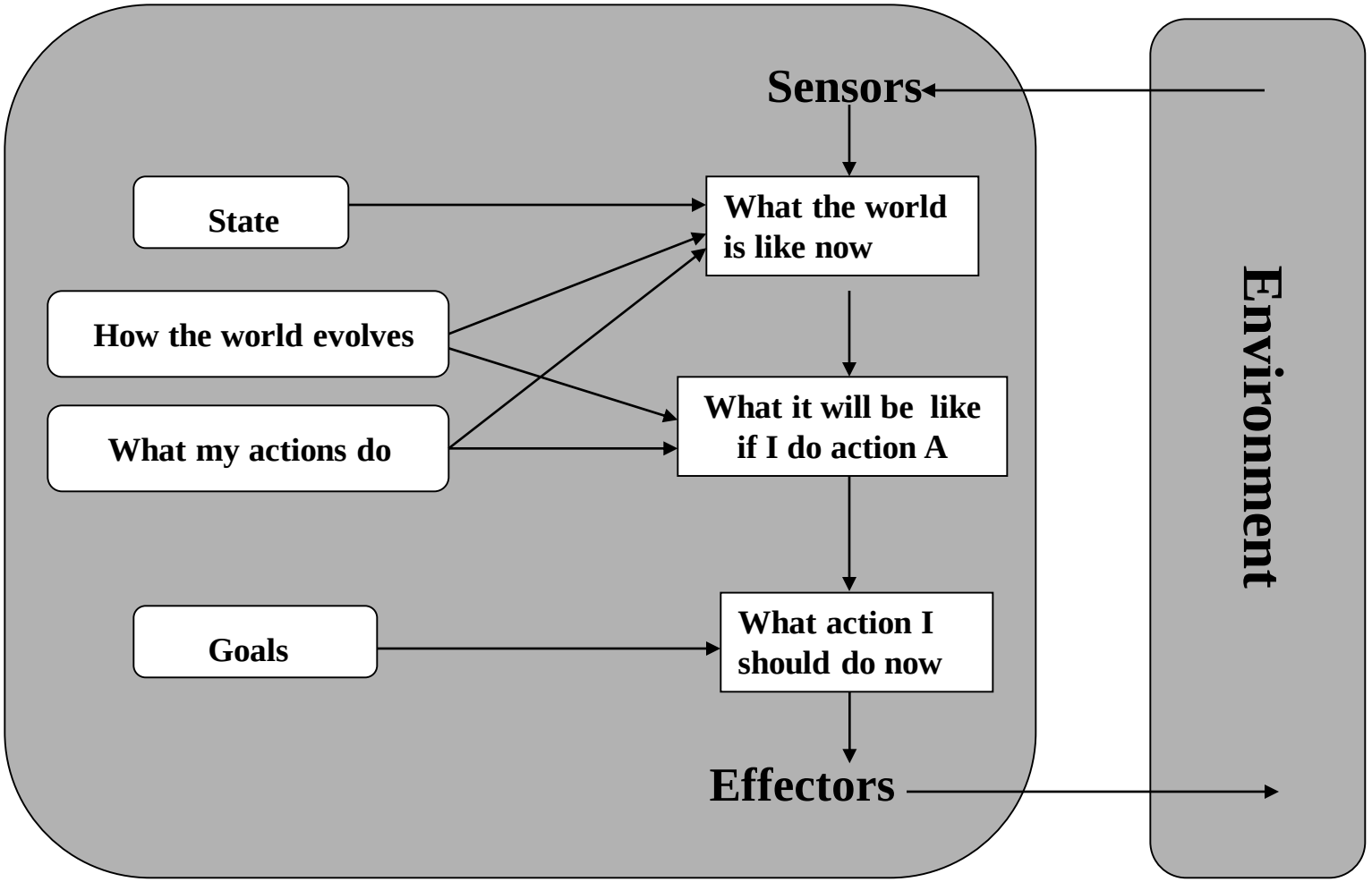
Reflex Agent with Internal State

- Encode "internal state" of the world to remember the past as contained in earlier percepts
- Needed because sensors do not usually give the entire state of the world at each input, so perception of the environment is captured over time. "State" used to encode different "world states" that generate the same immediate percept.
- Requires ability to represent change in the world; one possibility is to represent just the latest state, but then can't reason about hypothetical courses of action

Goal- Based Agent

- Choose actions so as to achieve a (given or computed) goal.
- A goal is a description of a desirable situation
- Keeping track of the current state is often not enough -- need to add goals to decide which situations are good
- Deliberative instead of reactive
- May have to consider long sequences of possible actions before deciding if goal is achieved -- involves consideration of the future, *"what will happen if I do...?"*

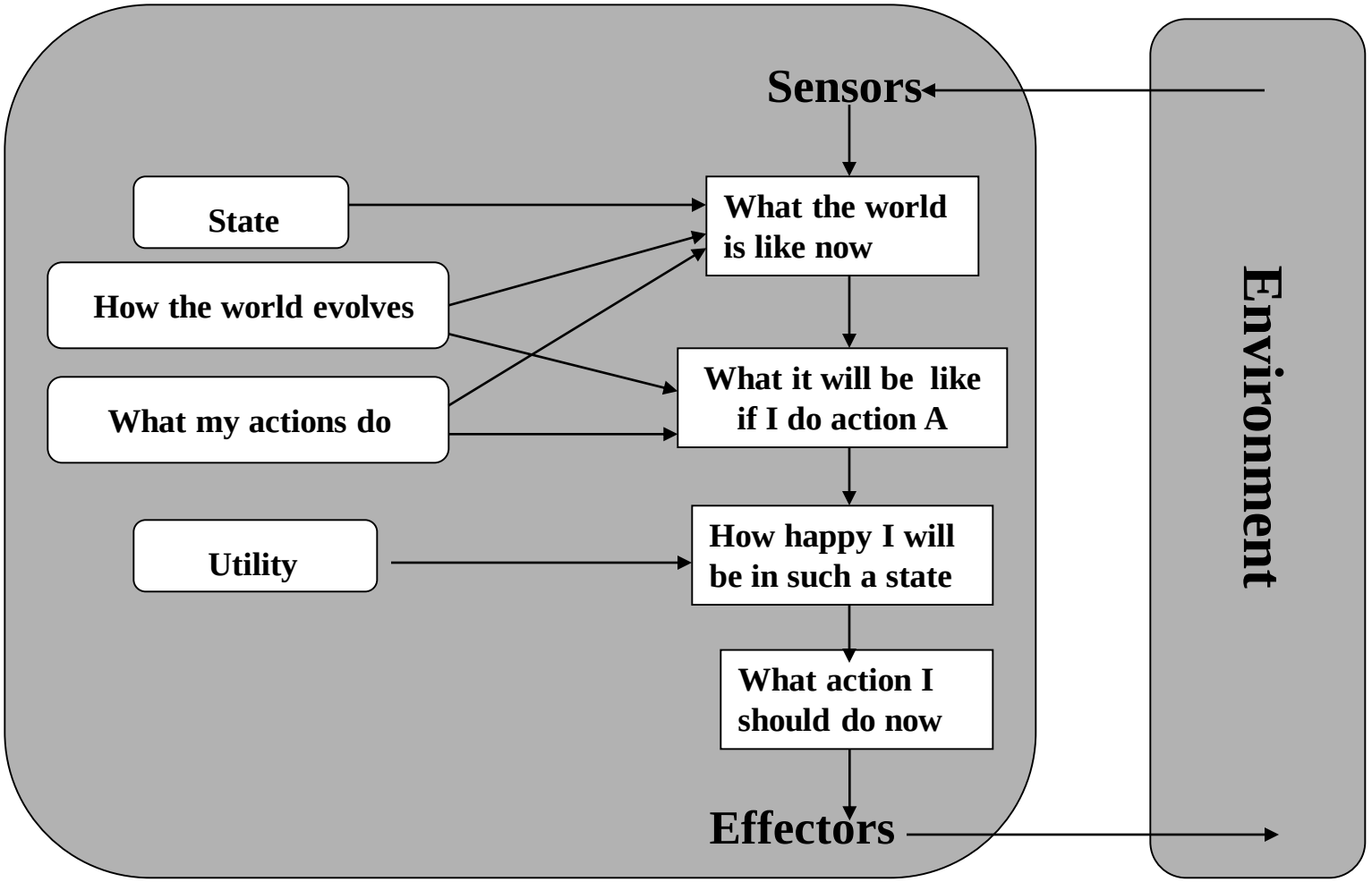
Agents with Explicit Goals



Utility- Based Agent

- When there are multiple possible alternatives, how to decide which one is best?
- A goal specifies a crude distinction between a happy and unhappy state, but often need a more general performance measure that describes "degree of happiness"
- Utility function **U: States --> Reals** indicating a measure of success or happiness when at a given state
- Allows decisions comparing choice between conflicting goals, and choice between likelihood of success and importance of goal (if achievement is uncertain)

A Complete Utility- Based Agent



Properties of Environments

- **Accessible/ Inaccessible.**
  - If an agent's sensors give it access to the complete state of the environment needed to choose an action, the environment is accessible.
  - Such environments are convenient, since the agent is freed from the task of keeping track of the changes in the environment.
- **Deterministic/ Nondeterministi**c.
  - An environment is deterministic if the next state of the environment is completely determined by the current state of the environment and the action of the agent.
  - In an accessible and deterministic environment the agent need not deal with uncertainty.
- **Episodic/ Nonepisodi**c.
  - An episodic environment means that subsequent episodes do not depend on what actions occurred in previous episodes.
  - Such environments do not require the agent to plan ahead.

Properties of Environments

- **Static/ Dynamic.**
  - An environment which does not change while the agent is thinking is static.
  - In a static environment the agent need not worry about the passage of time while he is thinking, nor does he have to observe the world while he is thinking.
  - In static environments the time it takes to compute a good strategy does not matter.
- **Discrete/ Continuous.**
  - If the number of distinct percepts and actions is limited the environment is discrete, otherwise it is continuous.
- **With/ Without rational adversaries.**
  - If an environment does not contain other rationally thinking, adversary agents, the agent need not worry about strategic, game theoretic aspects of the environment
  - Most engineering environments are without rational adversaries, whereas most social and economic systems get their complexity from the interactions of (more or less) rational agents.
  - As example for a game with a rational adversary, try the Prisoner's Dilemma

The Prisoners' Dilemma

- The two players in the game can choose between two moves, either "cooperate" or "defect".
- Each player gains when both cooperate, but if only one of them cooperates, the other one, who defects, will gain more.
- If both defect, both lose (or gain very little) but not as much as the "cheated" cooperator whose cooperation is not returned.
- If both decision- makers were purely rational, they would never cooperate. Indeed, rational decision- making means that you make the decision which is best for you whatever the other actor chooses.

	Cooperative	Defect
Cooperative	5	-10
Defect	10	0

Summary

- An **agent** perceives and acts in an environment, has an architecture and is implemented by an agent program.
- An **ideal agent** always chooses the action which maximizes its expected performance, given percept sequence received so far.
- An **autonomous agent** uses its own experience rather than built- in knowledge of the environment by the designer.
- An **agent program** maps from percept to action & updates its internal state.
  - **Reflex agents** respond immediately to percpets.
  - **Goal-based agents** act in order to achieve their goal(s).
  - **Utility-based** agents maximize their own utility function.
- **Representing knowledge** is important for successful agent design.
- Some **environments** are more difficult for agents than others. The most challenging environments are inaccessible, nondeterministic, nonepisodic, dynamic, and continuous.

Uninformed Search

Chapter 3

Some material adopted from notes by Tom Finin of CSEE, UMBC, and by Charles R. Dyer, University of Wisconsin-Madison

Building Goal-Based Agents

- We have a **goal** to reach
  - Driving from point A to point B
  - Put 8 queens on a chess board such that no one attacks another
  - Prove that John is an ancestor of Mary
- We have information about where we are now at the **beginning**
- We have a set of **actions** we can take to move around (change from where we are)
- **Objective**: find a sequence of legal actions which will bring us from the start point to a goal

What is the goal to be achieved?

- Could describe a situation we want to achieve, a set of properties that we want to hold, etc.
- Requires defining a **“goal test”** so that we know what it means to have achieved/satisfied our goal.
- This is a hard part that is rarely tackled in AI, usually assuming that the system designer or user will specify the goal to be achieved.

What are the actions?

- Quantify all of the primitive actions or events that are sufficient to describe all necessary changes in solving a task/goal.
- No uncertainty associated with what an action does to the world. That is, given an action (aka operator or move) and a description of the current state of the world, the action completely specifies
  - **Precondition:** if that action CAN be applied to the current world (i.e., is it applicable and legal), and
  - **Effect:** what the exact state of the world will be after the action is performed in the current world (i.e., no need for "history" information to be able to compute what the new world looks like).

Actions

- Note also that actions can all be considered as **discrete events** that can be thought of as occurring at an **instant of time**.
  - That is, the world is in one situation, then an action occurs and the world is now in a new situation. For example, if "Mary is in class" and then performs the action "go home," then in the next situation she is "at home." There is no representation of a point in time where she is neither in class nor at home (i.e., in the state of "going home").
- The number of operators needed depends on the **representation** used in describing a state.

Representing states

- At any moment, the relevant world is represented as a **state**
  - Initial (start) state: S
  - An action (or an operation) changes the current state to another state (if it is applied): state transition
  - An action can be taken (applicable) only if the its precondition is met by the current state
  - For a given state, there might be more than one applicable actions
  - Goal state: a state satisfies the goal description or passes the goal test
  - Dead-end state: a non-goal state to which no action is applicable

Representing states

- **Stat space**:
  - Includes the initial state S and all other states that are reachable from S by a sequence of actions
  - A state space can be organized as a graph:

    nodes: states in the space

    arcs: actions/operations
- The **size of a problem** is usually described in terms of the **number of states** (or the size of the state space) that are possible.
  - Tic-Tac-Toe has about 3^9 states.
  - Checkers has about 10^40 states.
  - Rubik's Cube has about 10^19 states.
  - Chess has about 10^120 states in a typical game.
  - GO has more states than Chess

Closed World Assumption

- We will generally use the **Closed World Assumption**.
- All necessary information about a problem domain is available in each percept so that each state is a complete description of the world.
- There is no incomplete information at any point in time.

Knowledge representation issues

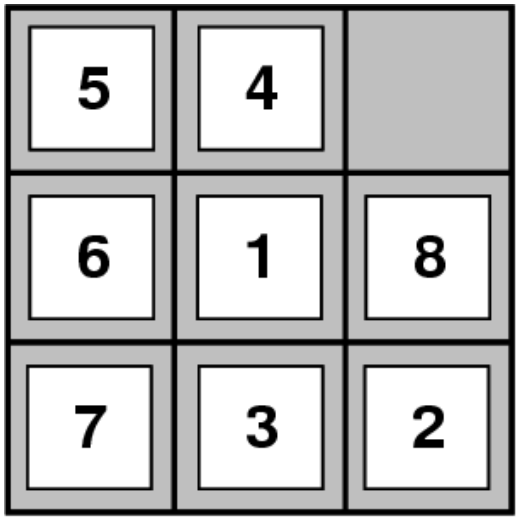
- What's in a state ?
  - Is the color of the boat relevant to solving the Missionaries and Cannibals problem? Is sunspot activity relevant to predicting the stock market? What to represent is a very hard problem that is usually left to the system designer to specify.
- What **level of abstraction** or detail to describe the world.
  - Too fine-grained and we'll "miss the forest for the trees." Too coarse-grained and we'll miss critical details for solving the problem.
- The number of states depends on the representation and level of abstraction chosen.
  - In the Remove-5-Sticks problem, if we represent the individual sticks, then there are 17-choose-5 possible ways of removing 5 sticks. On the other hand, if we represent the "squares" defined by 4 sticks, then there are 6 squares initially and we must remove 3 squares, so only 6-choose-3 ways of removing 3 squares.

Some example problems

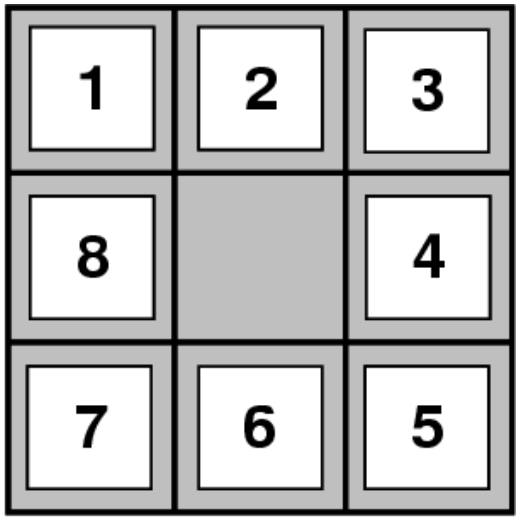
- Toy problems and micro-worlds
  - 8-Puzzle
  - Missionaries and Cannibals
  - Cryptarithmetic
  - Remove 5 Sticks
  - Traveling Salesman Problem (TSP)
- Real-world-problems

8-Puzzle

Given an initial configuration of 8 numbered tiles on a 3 x 3 board, move the tiles in such a way so as to produce a desired goal configuration of the tiles.



Start State

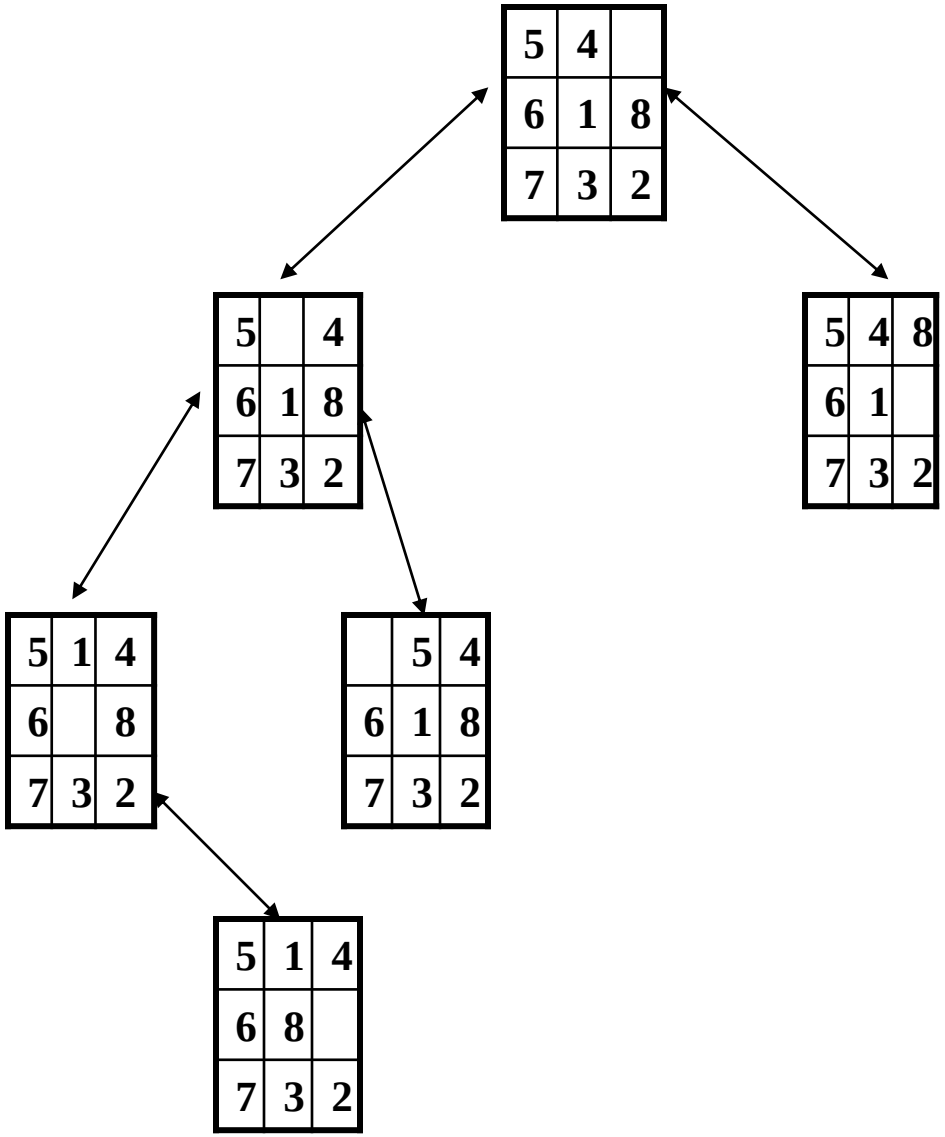


Goal State

8 puzzle

- **State:** 3 x 3 array configuration of the tiles on the board.
- **Operators:** Move Blank square Left, Right, Up or Down.
  - This is a more efficient encoding of the operators than one in which each of four possible moves for each of the 8 distinct tiles is used.
- **Initial State:** A particular configuration of the board.
- **Goal:** A particular configuration of the board.
- The state space is partitioned into two subspaces
- NP-complete problem, requiring O(2^k) steps where k is the length of the solution path.
- 15-puzzle problems (4 x 4 grid with 15 numbered tiles), and N-puzzles (N = n^2-1)

A portion of the state space of a 8-Puzzle problem



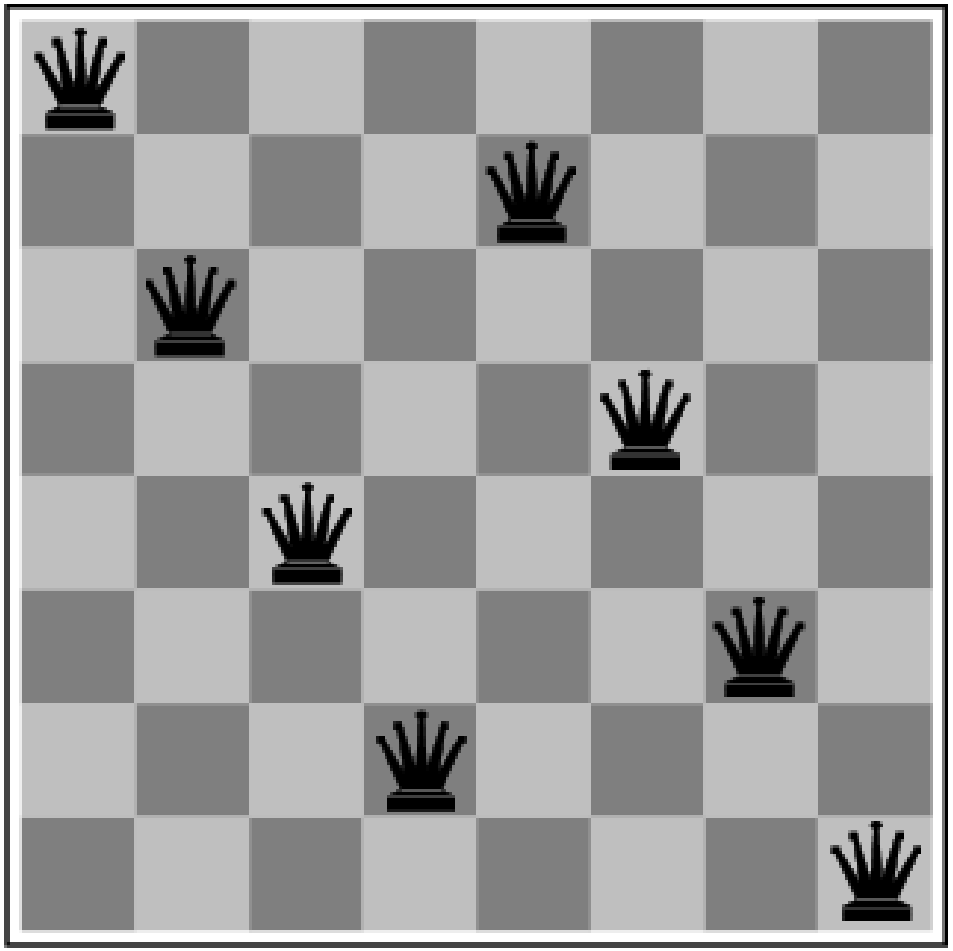
The 8-Queens Problem

Place eight queens on a chessboard such that no queen attacks any other!

Total # of states: 4.4x10^9

Total # of solutions:

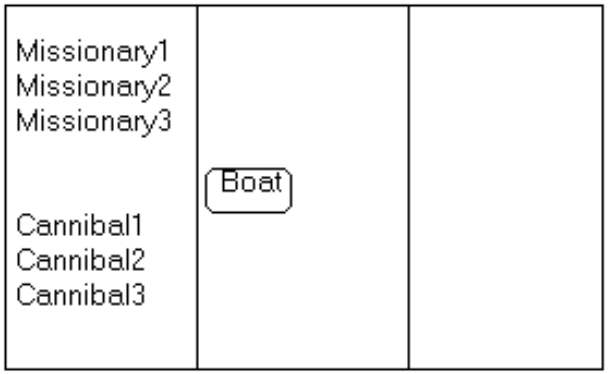
12 (or 96)



Missionaries and Cannibals

There are 3 missionaries, 3 cannibals, and 1 boat that can carry up to two people on one side of a river.

- **Goal**: Move all the missionaries and cannibals across the river.
- **Constraint:** Missionaries can never be outnumbered by cannibals on either side of river, or else the missionaries are killed.
- **State:** configuration of missionaries and cannibals and boat on each side of river.
- **Operators:** Move boat containing some set of occupants across the river (in either direction) to the other side.



3 Missionaries and 3 Cannibals wish to cross the river. They have a boat that will carry two people. Everyone can navigatethe boat. If at any time the Cannibals outnumber the missionaries on either bank of the river, they will eat the Missionaries. Find the smallest number of crossings that will allow everyone to cross the river safely.
The problem can be solved in 11 moves. But people rarely get the optimal solution, because the MC problem contains a 'tricky' state at the end, where two people move back across the river.

Missionaries and Cannibals Solution

	Near side		***Far side***
0 Initial setup:	MMMCCC	B	-
1 Two cannibals cross over:	MMMC	B	CC
2 One comes back:	MMMCC	B	C
3 Two cannibals go over again:	MMM	B	CCC
4 One comes back:	MMMC	B	CC
5 Two missionaries cross:	MC	B	MMCC
6 A missionary & cannibal return:	MMCC	B	MC
7 Two missionaries cross again:	CC	B	MMMC
8 A cannibal returns:	CCC	B	MMM
9 Two cannibals cross:	C	B	MMMCC
10 One returns:	CC	B	MMMC
11 And brings over the third:	-	B	MMMCCC

Cryptarithmetic

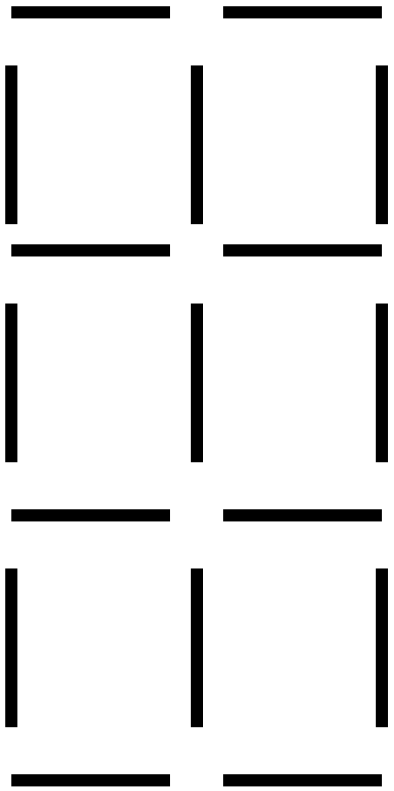
- Find an assignment of digits (0, ..., 9) to letters so that a given arithmetic expression is true. examples: SEND + MORE = MONEY and

```
 FORTY     Solution:  29786
+  TEN                 850
+  TEN                 850
 -----               -----
 SIXTY               31486
F=2, O=9, R=7, etc.
```

- Note: In this problem, the solution is NOT a sequence of actions that transforms the initial state into the goal state, but rather the solution is simply finding a goal node that includes an assignment of digits to each of the distinct letters in the given problem.

Remove 5 Sticks

- Given the following configuration of sticks, remove exactly 5 sticks in such a way that the remaining configuration forms exactly 3 squares.



Traveling Salesman Problem

- Given a road map of n cities, find the **shortest** tour which visits every city on the map exactly once and then return to the original city (*Hamiltonian circuit*)
- (Geometric version):
  - a complete graph of n vertices.
  - n!/2n legal tours
  - Find one legal tour that is shortest

Formalizing Search in a State Space

- A state space is a **graph**, (V, E) where V is a set of **nodes** and E is a set of **arcs**, where each arc is directed from a node to another node
- **node:** corresponds to a **state**
  - state description
  - plus optionally other information related to the parent of the node, operation to generate the node from that parent, and other bookkeeping data)
- **arc:** corresponds to an applicable action/operation.
  - the source and destination nodes are called as **parent (immediate predecessor)** and **child** (**immediate successor**) nodes with respect to each other
  - ancestors((predecessors) and descendents (successors)
  - each arc has a fixed, non-negative **cost** associated with it, corresponding to the cost of the action

- **node generation:** making explicit a node by applying an action to another node which has been made explicit
- **node expansion:** generate **all** children of an explicit node by applying **all** applicable operations to that node
- One or more nodes are designated as **start nodes**
- A **goal test** predicate is applied to a node to determine if its associated state is a goal state
- A **solution** is a sequence of operations that is associated with a path in a state space from a start node to a goal node
- The **cost of a solution** is the sum of the arc costs on the solution path

- **State-space search** is the process of searching through a state space for a solution by making explicit a sufficient portion of an implicit state-space graph to include a goal node.
  - Hence, initially V={S}, where S is the start node; when S is expanded, its successors are generated and those nodes are added to V and the associated arcs are added to E. This process continues until a goal node is generated (included in V) and identified (by goal test)
- During search, a node can be in one of the three categories:
  - Not generated yet (has not been made explicit yet)
  - **OPEN**: generated but not expanded
  - **CLOSED**: expanded
  - Search strategies differ mainly on how to select an OPEN node for expansion at each step of search

A General State-Space Search Algorithm

- Node n
  - state description
  - parent (may use a backpointer) (if needed)
  - Operator used to generate n (optional)
  - Depth of n (optional)
  - Path cost from S to n (if available)
- OPEN list
  - initialization: {S}
  - node insertion/removal depends on specific search strategy
- CLOSED list
  - initialization: {}
  - organized by backpointers

A General State-Space Search Algorithm

```
open := {S}; closed :={};
repeat
   n := select(open);              /* select one node from open for expansion */
   if n is a goal
        then exit with success;  /* delayed goal testing */
   expand(n)
            /* generate all children of n
               put these newly generated nodes in open (check duplicates)
               put n in closed (check duplicates) */
until open = {};
exit with failure
```

Some Issues

- Search process constructs a search tree, where
  - **root** is the initial state S, and
  - **leaf nodes** are nodes
    - not yet been expanded (i.e., they are in OPEN list) or
    - having no successors (i.e., they're "deadends")
- Search tree may be infinite because of loops even if state space is small
- Search strategies mainly differ on *select* (open)
- Each node represents a partial solution path (and cost of the partial solution path) from the start node to the given node.
  - in general, from this node there are many possible paths (and therefore solutions) that have this partial path as a prefix.

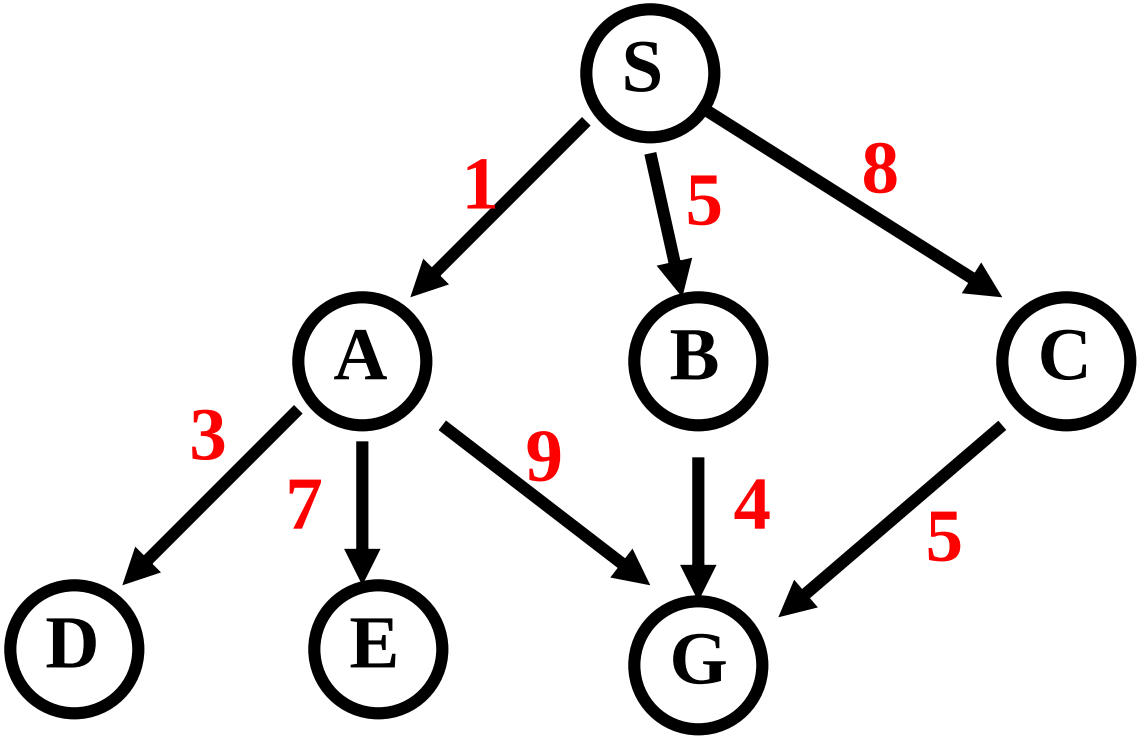
Evaluating Search Strategies

- **Completeness**
  - Guarantees finding a solution whenever one exists
- **Time Complexity**
  - How long (worst or average case) does it take to find a solution? Usually measured in terms of the **number of nodes expanded**
- **Space Complexity**
  - How much space is used by the algorithm? Usually measured in terms of the **maximum size that the "OPEN" list** becomes during the search
- **Optimality/Admissibility**
  - If a solution is found, is it guaranteed to be an optimal one? For example, is it the one with minimum cost?

Uninformed vs. Informed Search

- **Uninformed Search Strategies**
  - Breadth-First search
  - Depth-First search
  - Uniform-Cost search
  - Depth-First Iterative Deepening search
- **Informed Search Strategies**
  - Hill climbing
  - Best-first search
  - Greedy Search
  - Beam search
  - Algorithm A
  - Algorithm A*

Example Illustrating Uninformed Search Strategies



Breadth-First

- Algorithm outline:
  - Always select from the OPEN the node with the smallest depth for expansion, and put all newly generated nodes into OPEN
  - OPEN is organized as **FIFO** (first-in, first-out) list, I.e., a **queue**.
  - Terminate if a node selected for expansion is a goal
- **Properties**
  - **Complete**
  - **Optimal** (i.e., admissible) if all operators have the same cost. Otherwise, not optimal but finds solution with shortest path length **(shallowest solution)**.
  - **Exponential time and space complexity**,

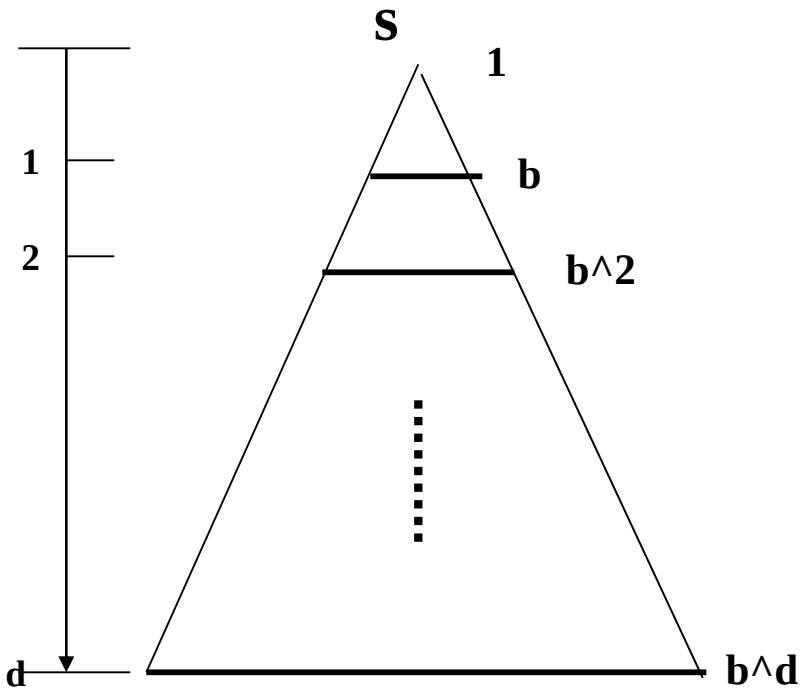
    O(b^d) nodes will be generated, where

    d is the depth of the solution and

    b is the branching factor (i.e., number of children) at each node

Breadth-First

- A complete search tree of depth d where each non-leaf node has b children, has a total of 1 + b + b^2 + ... + b^d = (b^(d+1) - 1)/(b-1) nodes
- Time complexity (# of nodes generated): O(b^d)
- Space complexity (maximum length of OPEN): O(b^d)



- For a complete search tree of depth 12, where every node at depths 0, ..., 11 has 10 children and every node at depth 12 has 0 children, there are 1 + 10 + 100 + 1000 + ... + 10^12 = (10^13 - 1)/9 = O(10^12) nodes in the complete search tree.

- BFS is suitable for problems with shallow solutions

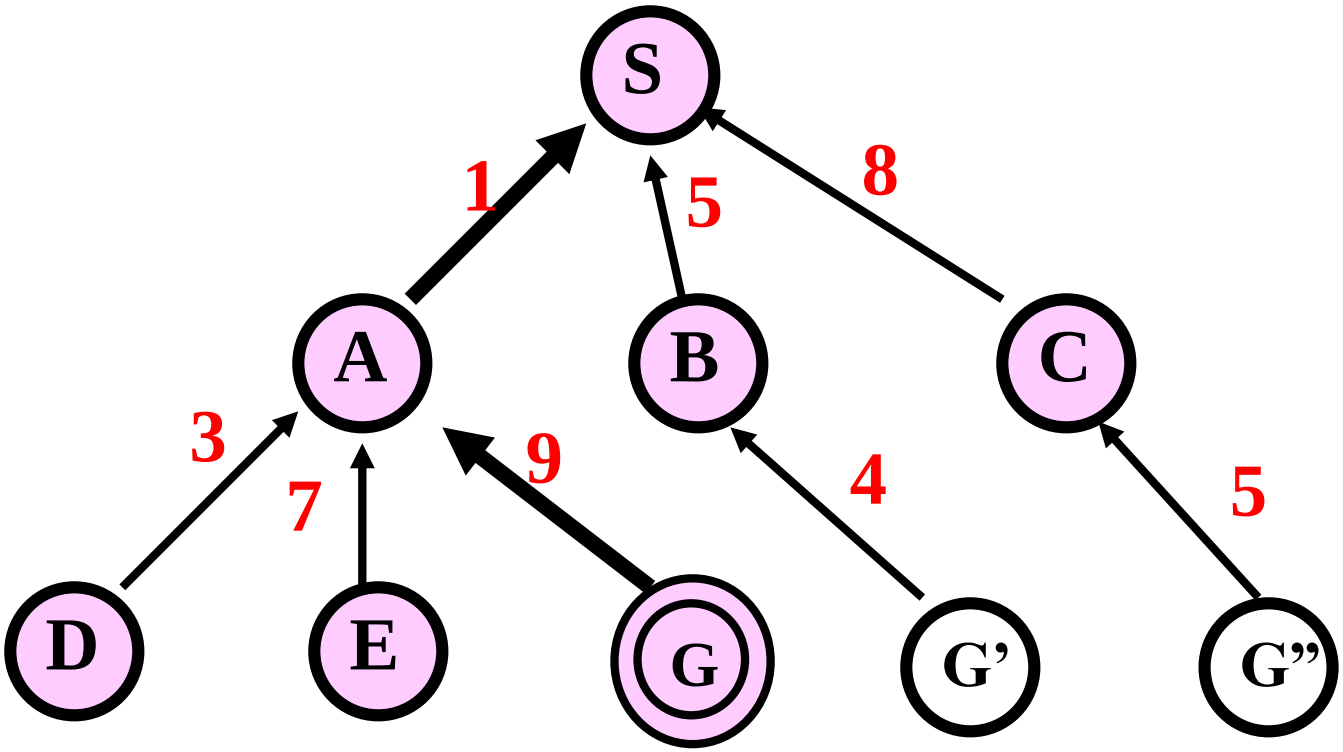
Breadth-First Search

exp. node	OPEN list	CLOSED list
	{ S }	{}
S	{ A B C }	{S}
A	{ B C D E G }	{S A}
B	{ C D E G G' }	{S A B}
C	{ D E G G' G" }	{S A B C}
D	{ E G G' G" }	{S A B C D}
E	{ G G' G" }	{S A B C D E}
G	{ G' G" }	{S A B C D E}

Solution path found is S A G <-- this G also has cost 10

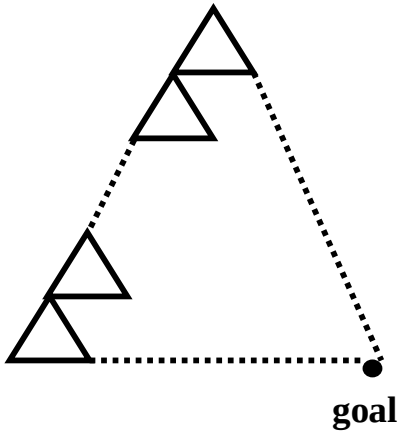
Number of nodes expanded (including goal node) = 7

CLOSED List: the search tree connected by backpointers



Depth-First (DFS)

- Algorithm outline:
  - Always select from the OPEN the node with the greatest depth for expansion, and put all newly generated nodes into OPEN
  - OPEN is organized as **LIFO** (last-in, first-out) list.
  - Terminate if a node selected for expansion is a goal

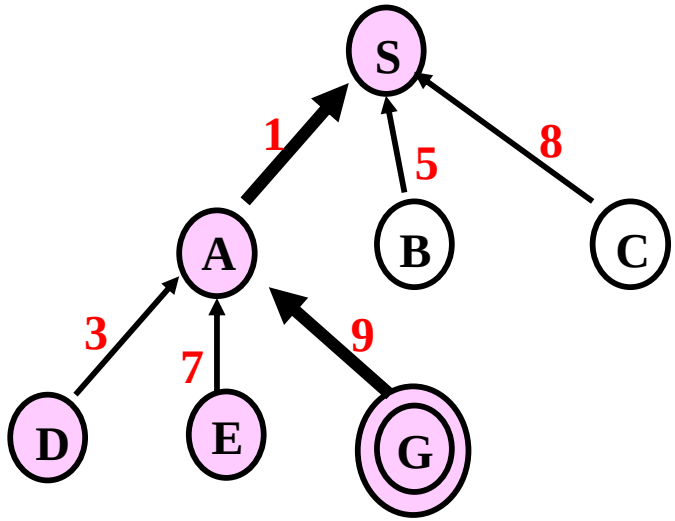


- **May not terminate** without a "depth bound," i.e., cutting off search below a fixed depth D (How to determine the depth bound?)
- **Not complete** (with or without cycle detection, and with or without a cutoff depth)
- **Exponential time**, O(b^d), but only **linear space**, O(bd), required
- Can find **deep solutions quickly** if lucky
- When search hits a deadend, can only back up one level at a time even if the "problem" occurs because of a bad operator choice near the top of the tree. Hence, only does "chronological backtracking"

Depth-First Search

return GENERAL-SEARCH(problem, ENQUEUE-AT-FRONT)

exp. node	**OPEN list**	**CLOSED list**
	{ S }	
S	{ A B C }	
A	{ D E G B C}	
D	{ E G B C }	
E	{ G B C }	
G	{ B C }	



Solution path found is S A G <-- this G has cost 10

Number of nodes expanded (including goal node) = 5

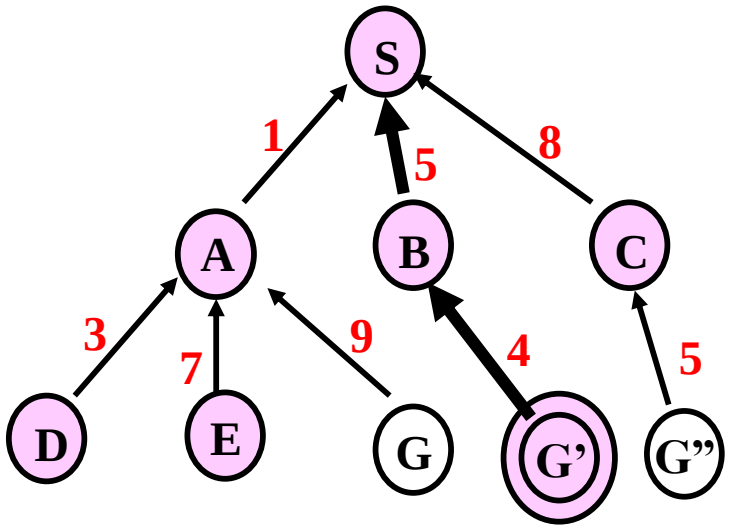
Uniform-Cost (UCS)

- Let g(n) = cost of the path from the start node to an open node n
- Algorithm outline:
  - Always select from the OPEN the node with the least g(.) value for expansion, and put all newly generated nodes into OPEN
  - Nodes in OPEN are sorted by their g(.) values (in ascending order)
  - Terminate if a node selected for expansion is a goal
- Called *"Dijkstra's Algorithm"* in the algorithms literature and similar to *"Branch and Bound Algorithm"* in operations research literature

Uniform-Cost Search

GENERAL-SEARCH(problem, ENQUEUE-BY-PATH-COST)

exp. node	**nodes list**	**CLOSED list**
	{S(0)}	
S	{A(1) B(5) C(8)}	
A	{D(4) B(5) C(8) E(8) G(10)}	
D	{B(5) C(8) E(8) G(10)}	
B	{C(8) E(8) G'(9) G(10)}	
C	{E(8) G'(9) G(10) G"(13)}	
E	{G'(9) G(10) G"(13) }	
G'	{G(10) G"(13) }	



Solution path found is S B G <-- this G has cost 9, not 10

Number of nodes expanded (including goal node) = 7

Uniform-Cost (UCS)

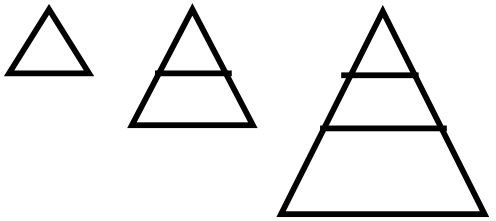
- **Complete** (if cost of each action is not infinitesimal)
  - The total # of nodes n with g(n) <= g(goal) in the state space is finite
  - If n' is a child of n, then g(n') = g(n) + c(n, n') > g(n)
  - Goal node will eventually be generated (put in OPEN) and selected for expansion (and passes the goal test)
- **Optimal/Admissible**
  - Admissibility depends on the goal test being applied when a node is removed from the OPEN list, not when it's parent node is expanded and the node is first generated (delayed goal testing)
  - Multiple solution paths (following different backpointers)
  - Each solution path that can be generated from an open node n will have its path cost >= g(n)
  - When the first goal node is selected for expansion (and passes the goal test), its path cost is less than or equal to g(n) of every OPEN node n (and solutions entailed by n)
- **Exponential time and space complexity**, O(b^d) where d is the depth of the solution path of the least cost solution

Depth-First Iterative Deepening (DFID)

- BF and DF both have exponential time complexity O(b^d)
  - BF is **complete** but has exponential space complexity
  - DF has **linear space complexity** but is incomplete
- Space is often a **harder** resource constraint than time
- Can we have an algorithm that
  - Is complete
  - Has linear space complexity, and
  - Has time complexity of O(b^d)
- DFID by Korf in 1985 (17 years after A*)

  First do DFS to depth 0 (i.e., treat start node as having no successors), then, if no solution found, do DFS to depth 1, etc.

  until solution found do
  DFS with depth bound c
  c = c+1



Depth-First Iterative Deepening (DFID)

- **Complete** (iteratively generate all nodes up to depth d)
- **Optimal/Admissible** if all operators have the same cost. Otherwise, not optimal but does guarantee finding solution of shortest length (like BF).
- **Linear space complexity:** O(bd), (like DF)
- **Time complexity** is a little worse than BFS or DFS because nodes near the top of the search tree are generated multiple times, but because almost all of the nodes are near the bottom of a tree, the worst case time complexity is still exponential, O(b^d)

Depth-First Iterative Deepening

- If branching factor is b and solution is at depth d, then nodes at depth d are generated once, nodes at depth d-1 are generated twice, etc., and node at depth 1 is generated d times.

  Hence

  total(d) = b^d + 2b^(d-1) + ... + db

  <= b^d / (1 - 1/b)^2 = O(b^d).

  – If b=4, then worst case is 1.78 * 4^d, I.e., 78% more nodes searched than exist at depth d (in the worst case).

$$tota(d) = 1 \cdot b^d + 2 \cdot b^{d-1} + \cdots + (d-1) \cdot b^2 + d \cdot b$$

$$= b^d (1 + 2 \cdot b^{-1} + \cdots + (d-1) \cdot b^{2-d} + d \cdot b^{1-d})$$

Let $x = b^{-1}$, then

$$tota(d) = b^d (1 + 2 \cdot x^1 + \cdots + (d-1) \cdot x^{d-2} + d \cdot x^{d-1})$$

$$= b^d \frac{d}{dx}(x + x^2 + \cdots + x^{d-1} + x^d)$$

$$= b^d \frac{d}{dx} \frac{(x - x^{d+1})}{1-x}$$

$$\leq b^d \frac{d}{dx} \frac{x}{1-x} \qquad /* \, x^{d+1} << 1 \text{ when } d \text{ is large since } 1/b < 1 */$$

$$= b^d \frac{1 \cdot (1-x) - x \cdot (-1)}{(1-x)^2}. \text{ Therefore}$$

$$tota(d) \leq b^d /(1-x)^2 = b^d /(1-b^{-1})^2$$

Comparing Search Strategies

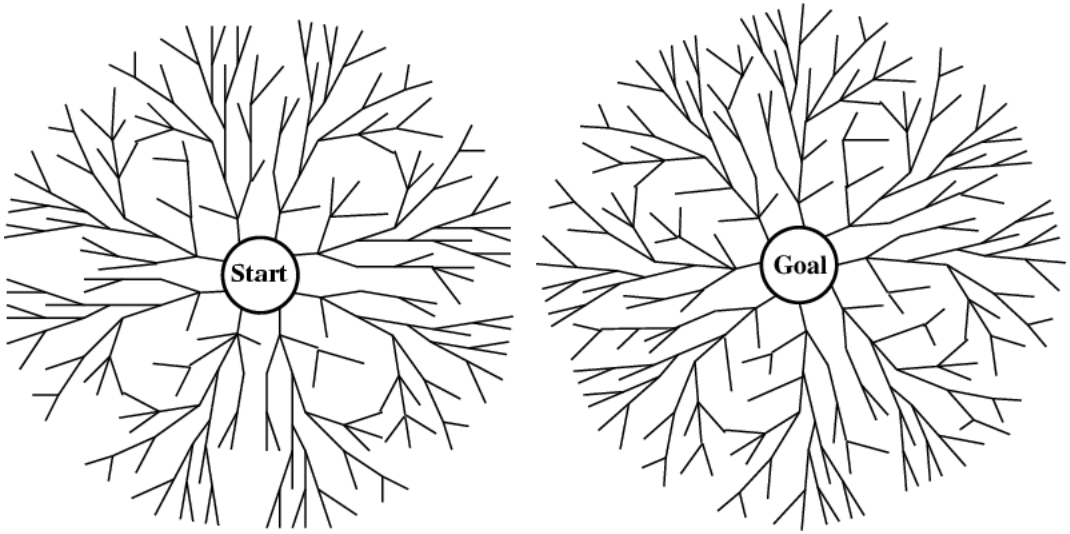
Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Time	b^d	b^d	b^m	b^l	b^d	$b^{d/2}$
Space	b^d	b^d	bm	bl	bd	$b^{d/2}$
Optimal?	Yes	Yes	No	No	Yes	Yes
Complete?	Yes	Yes	No	Yes, if $l \geq d$	Yes	Yes

When to use what

- **Depth-First Search:**
  - Many solutions exist
  - Know (or have a good estimate of) the depth of solution
- **Breadth-First Search**:
  - Some solutions are known to be shallow
- **Uniform-Cost Search**:
  - Actions have varying costs
  - Least cost solution is the required

  This is the only uninformed search that worries about costs.
- **Iterative-Deepening Search**:
  - Space is limited and the shortest solution path is required

Bi-directional search

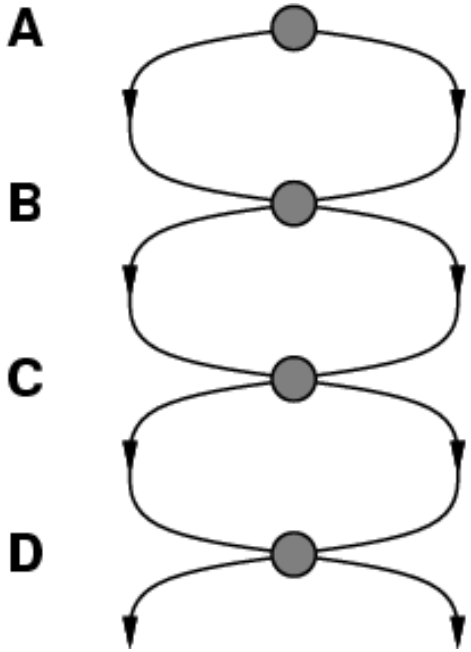


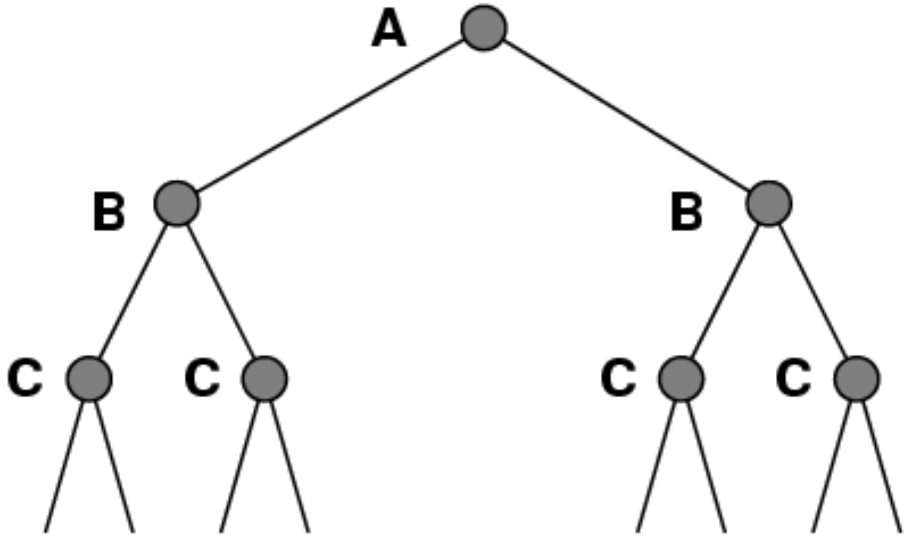
- Alternate searching from the start state toward the goal and from the goal state toward the start.
- Stop when the frontiers intersect.
- Works well only when there are unique start and goal states and when actions are reversible
- Can lead to finding a solution more quickly (but watch out for pathological situations.

Avoiding Repeated States

- In increasing order of effectiveness in reducing size of state space (and with increasing computational costs.)
  1. Do not return to the state you just came from.
  2. Do not create paths with cycles in them.
  3. Do not generate any state that was ever created before.
- Net effect depends on ``loops'' in state-space.

A State Space that Generates an Exponentially Growing Search Space





Informed Search

chapter 4

Informed Methods Add Domain-Specific Information

- Add domain-specific information to select what is the best path to continue searching along
- Define a heuristic function, h(n), that estimates the "goodness" of a node n with respect to reaching a goal.
- Specifically, h(n) = estimated cost (or distance) of minimal cost path from n to a goal state.
- **h(n) is about cost of the future search, g(n) past search**
- h(n) is an estimate (rule of thumb), based on domain-specific information that is computable from the current state description. Heuristics do not guarantee feasible solutions and are often without theoretical basis.

Heuristics

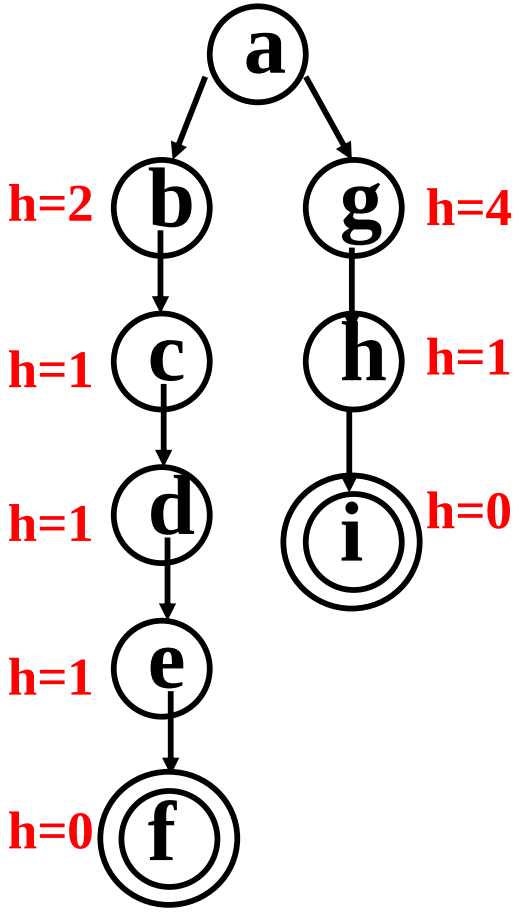
- Examples:
  - Missionaries and Cannibals: Number of people on starting river bank
  - 8-puzzle: Number of tiles out of place (i.e., not in their goal positions)
  - 8-puzzle: Sum of Manhattan distances each tile is from its goal position
  - 8-queen: # of un-attacked positions – un-positioned queens
- In general:
  - h(n) >= 0 for all nodes n
  - h(n) = 0 implies that n is a goal node
  - h(n) = infinity implies that n is a deadend from which a goal cannot be reached

Best First Search

- Order nodes on the OPEN list by increasing value of an evaluation function, f(n) , that incorporates domain-specific information in some way.
- Example of f(n):
  - f(n) = g(n) (uniform-cost)
  - f(n) = h(n) (greedy algorithm)
  - f(n) = g(n) + h(n) (algorithm A)
- This is a generic way of referring to the class of informed methods.

Greedy Search

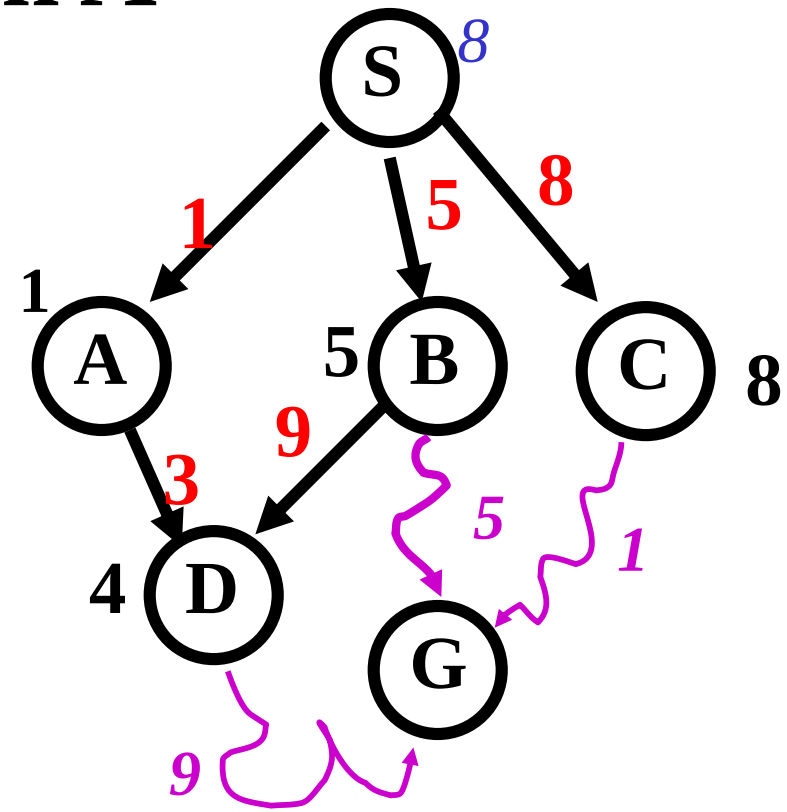
- Evaluation function f(n) = h(n), sorting open nodes by increasing values of f.
- Selects node to expand believed to be closest (hence it's "greedy") to a goal node (i.e., smallest f = h value)
- Not admissible, as in the example. Assuming all arc costs are 1, then Greedy search will find goal f, which has a solution cost of 5, while the optimal solution is the path to goal i with cost 3.
- Not complete (if no duplicate check)



Algorithm A

- Use as an evaluation function

  f(n) = g(n) + h(n)
- The h(n) term represents a “depth-first“ factor in f(n)
- g(n) = minimal cost path from the start state to state n generated so far
- The g(n) term adds a "breadth-first" component to f(n).
- Ranks nodes on OPEN list by estimated cost of solution from start node through the given node to goal.
- Not complete if h(n) can equal infinity.
- Not admissible



(h*(D)=2, h*(C)=2)

f(D)=4+9=13
f(B)=5+5=10
f(C)=8+1=9

C is chosen next to expand

Algorithm A

OPEN := {S}; CLOSED := {};

repeat

Select node n from OPEN with minimal f(n) and place n on CLOSED;

if n is a goal node **exit** with success;

Expand(n);

For each child n' of n do

if n' is not already on OPEN or CLOSED **then**

put n′ on OPEN; set backpointer from n' to n

compute h(n'), g(n')=g(n)+ c(n,n'), f(n')=g(n')+h(n');

else if n' is already on OPEN or CLOSED and **if** g(n') is lower for the new version of n' **then**

discard the old version of n‘;

Put n' on OPEN; set backpointer from n' to n

until OPEN = {};

exit with failure

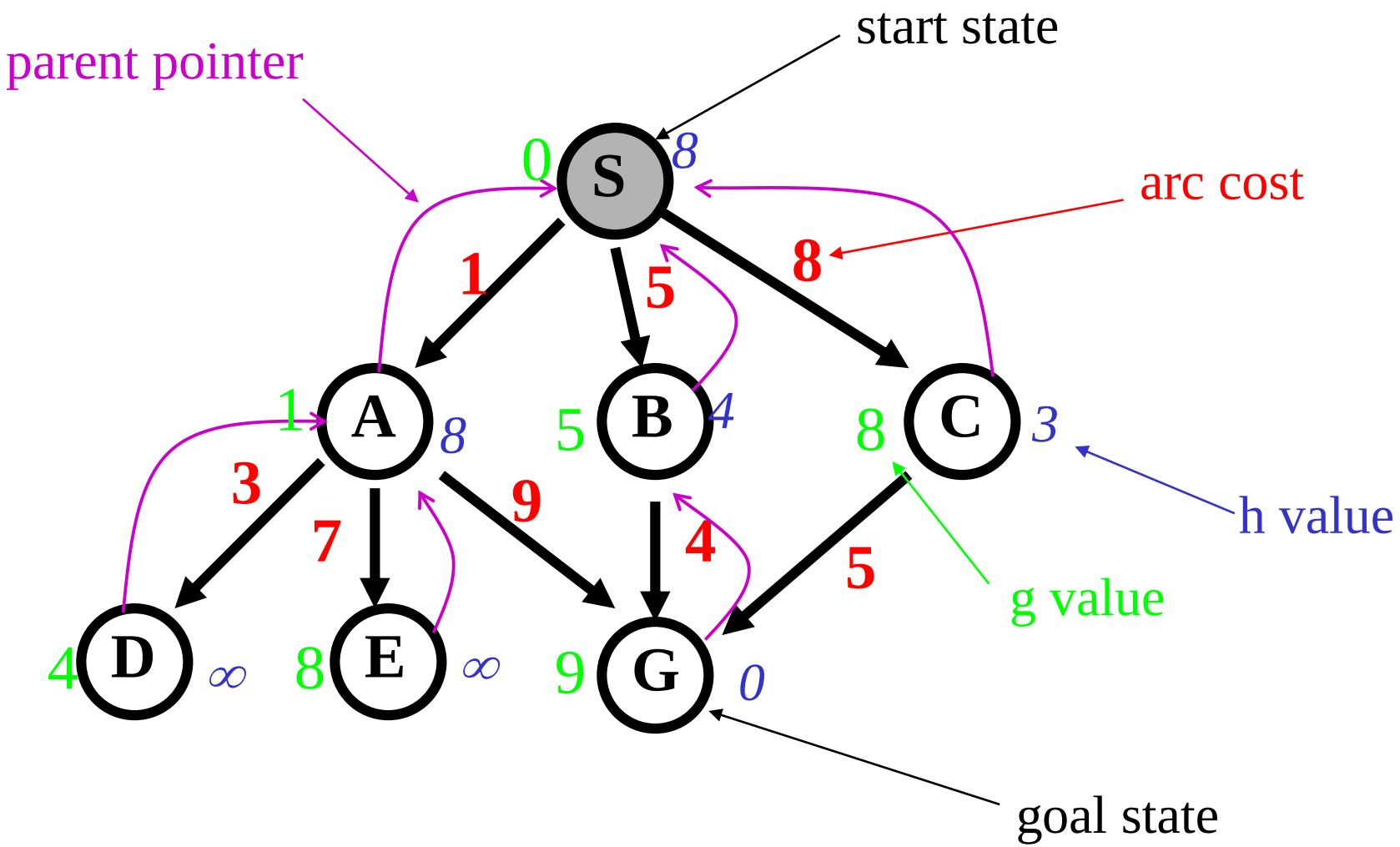
Algorithm A*

- Algorithm A with constraint that **h(n) <= h*(n)**
- **h*(n)** = true cost of the minimal cost path from n to any goal.
- **g*(n)** = true cost of the minimal cost path from S to n.
- **f*(n) =** h*(n)+g*(n) = true cost of the minimal cost solution path from S to to any goal going through n.
- h is **admissible** when h(n) <= h*(n) holds.
- Using an admissible heuristic guarantees that the first solution found will be an optimal one.
- A* is **complete** whenever the branching factor is finite, and every operator has a fixed positive cost (total # of nodes with f(.) <= f*(goal) is finite)
- A* is **admissible**

Some Observations on A*

- **Null heuristic:** If h(n) = 0 for all n, then this is an admissible heuristic and A* acts like Uniform-Cost Search.
- **Better heuristic:** If h1(n) <= h2(n) <= h*(n) for all non-goal nodes, then h2 is a better heuristic than h1
  - If A1* uses h1, and A2* uses h2, then every node expanded by A2* is also expanded by A1*.
  - In other words, A1 expands at least as many nodes as A2*.
  - We say that A2* is better informed than A1*.
- The closer h is to h*, the fewer extra nodes that will be expanded
- **Perfect heuristic:** If h(n) = h*(n) for all n, then only the nodes on the optimal solution path will be expanded. So, no extra work will be performed.

Example search space



Example

n	g(n)	h(n)	f(n)	h*(n)
S	0	8	8	9
A	1	8	9	9
B	5	4	9	4
C	8	3	11	5
D	4	inf	inf	inf
E	8	inf	inf	inf
G	9	0	9	0

- h*(n) is the (hypothetical) perfect heuristic.
- Since h(n) <= h*(n) for all n, h is admissible
- Optimal path = S B G with cost 9.

Greedy Algorithm

f(n) = h(n)

```
node exp.       OPEN list
                {S(8)}
    S           {C(3) B(4) A(8)}
    C           {G(0) B(4) A(8)}
    G           {B(4) A(8)}
```

- Solution path found is S C G with cost 13.
- 3 nodes expanded.
- Fast, but not optimal.

A* Search

f(n) = g(n) + h(n)

```
node exp.      OPEN list
               {S(8)}
    S          {A(9) B(9) C(11)}
    A          {B(9) G(10) C(11) D(inf) E(inf)}
    B          {G(9) G(10) C(11) D(inf) E(inf)}
    G          {C(11) D(inf) E(inf)}
```

- Solution path found is S B G with cost 9
- 4 nodes expanded.
- Still pretty fast. And optimal, too.

Proof of the Optimality of A*

- Let l* be the **optimal** solution path (from S to G), let f* be its cost
- At any time during the search, one or more node on l* are in OPEN
- We assume that A* has selected G2, a goal state with a suboptimal solution (g(G2) > f*).
- We show that this is impossible.
  - Let node n be the shallowest OPEN node on l*
  - Because all ancestors of n along l* are expanded, g(n)=g*(n)
  - Because h(n) is admissible, h*(n)>=h(n). Then

    f* =g*(n)+h*(n) >= g*(n)+h(n) = g(n)+h(n)= f(n).
  - If we choose G2 instead of n for expansion, f(n)>=f(G2).
  - This implies f*>=f(G2).
  - G2 is a goal state: h(G2) = 0, f(G2) = g(G2).
  - Therefore f* >= g(G2)
  - Contradiction.

Iterative Deepening A* (IDA*)

- **Idea:**
  - Similar to IDDF except now at each iteration the **DF search** is not bound by the current depth_limit but by the current **f_limit**
  - At each iteration, all nodes with **f(n) <= f_limit** will be expanded (in DF fashion).
  - If no solution is found at the end of an iteration, **increase f_limit** and start the next iteration
- **f_limit:**
  - Initialization: f_limit := h(s)
  - Increment: at the end of each (unsuccessful) iteration,

    f_limit := max{f(n)|n is a cut-off node}
- **Goal testing**: test all cut-off nodes until a solution is found
- **Admissible** if h is admissible

Automatic generation of h functions

- Original problem **P** — Relaxed problem **P'**

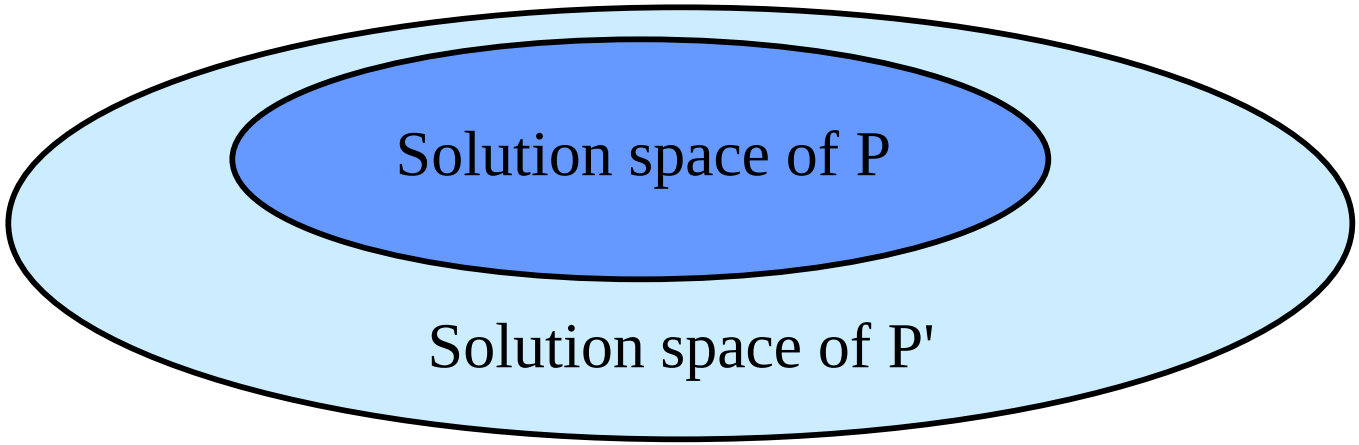
  A set of constraints — removing one or more constraints

  P is complex — **P'** becomes simpler

- Use cost of a best solution path from n in **P'** as h(n) for **P**
- **Admissibility:**

  h^* $\quad$ h

  cost of best solution in **P** >= cost of best solution in **P'**



Automatic generation of h functions

- Example: 8-puzzle
  - Constraints: to move from cell A to cell B
    cond1: there is a tile on A
    cond2: cell B is empty
    cond3: A and B are adjacent (horizontally or vertically)
  - Removing cond2:
    h2 (sum of Manhattan distances of all misplaced tiles)
  - Removing cond2 and cond3:
    h1 (# of misplaced tiles)
  - Removing cond3:
    h3, a new heuristic function

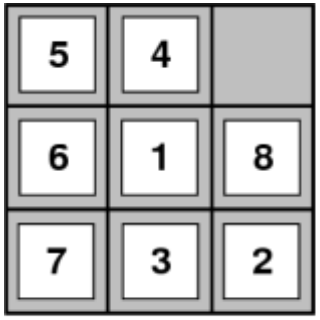
h3:

repeat

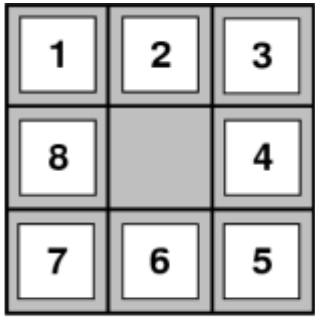
if the current empty cell A is to be occupied by tile x in the goal, move x to A. Otherwise, move into A any arbitrary misplaced tile.

until the goal is reached

- h2>= h3 >= h1



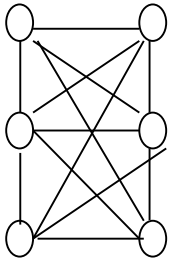
Start State



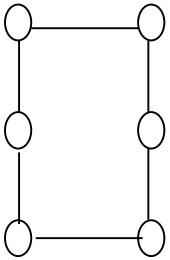
Goal State

h1(start) = 7
h2(start) = 18
h3(start) = 7

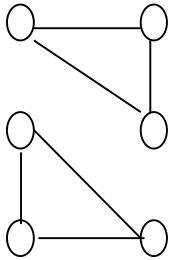
- Example: TSP. A legal tour is a (Hamiltonian) circuit
  - It is a **connected** second degree graph (each node has exactly two adjacent edges)
    Removing the connectivity constraint leads to h1:
    find the cheapest second degree graph from the given graph
    (with o(n^3) complexity)

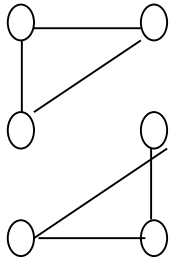


The given complete graph



A legal tour



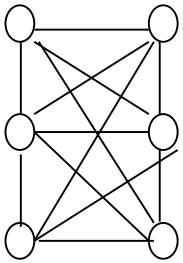


Other second degree graphs

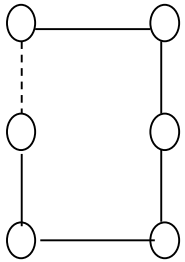
– It is a spanning tree (when an edge is removed) with the constraint that each node has at most 2 adjacent edges)
Removing the constraint leads to h2:

find the cheapest minimum spanning tree from the given graph

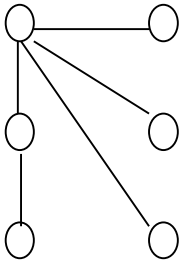
(with O(n^2/log n)

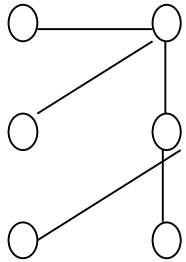


The given graph



A legal tour





Other MST

Complexity of A* search

- In general, exponential time and space complexity
- For subexponential growth of # of nodes expanded, need

  $$|h(n)-h^*(n)| <= O(\log h^*(n)) \text{ for all } n$$

  For most problem we have $|h(n)-h^*(n)| <= O(h^*(n)$
- Relaxing optimality
  - Weighted evaluation function

    $F(n)=(1-w)*g(n)+w*h(n)$

    w=0: uniformed-cost search

    w=1: greedy algorithm

    w=1/2: A* algorithm

– Dynamic weighting

f(n)=g(n)+h(n)+ε [1- d(n)/N]*h(n)

d(n): depth of node n

N: anticipated depth of an optimal goal

at beginning of search: $d(n) << N$

$f(n) \approx g(n) + (1+\varepsilon)h(n)$ encourages DF search at beginning of search: $d(n) \approx N$

$f(n) \approx g(n) + h(n)$ back to A*

It is ε-admissible (solution cost found is <= (1+ ε) solution found by A*)

- A^*_ε : another ε-admissible algorithm

  do not put a new node n on OPEN unless

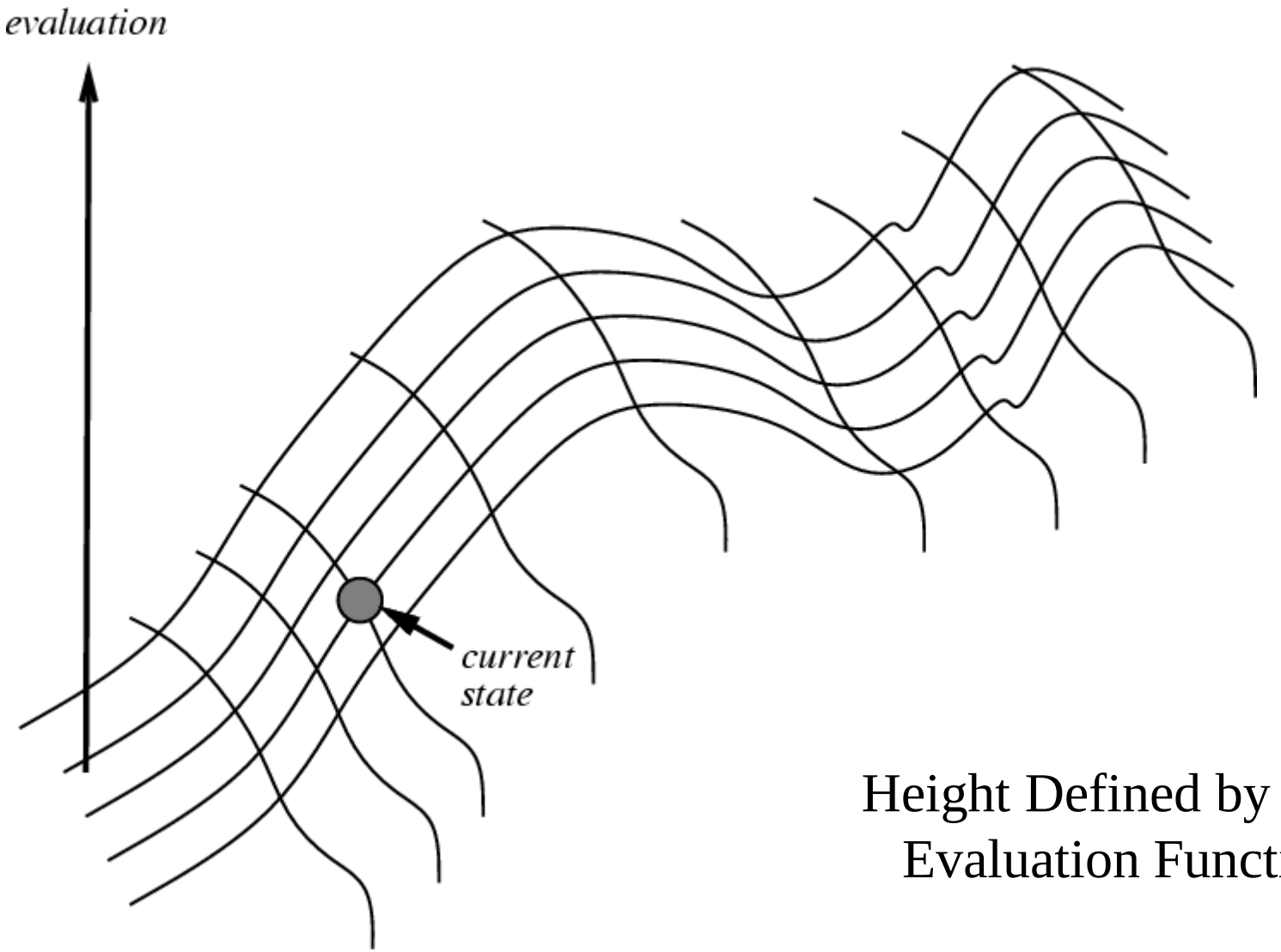
  f(n) <= smallest f value among all nodes already in OPE
- Pruning OPEN list
  - Find a solution using some quick but non-admissible method (e.g., greedy algorithm, hill-climbing, neural networks) with cost f+
  - Do not put a new node n on OPEN unless f(n)<=f+
  - Admissible: suppose f(n)>f+>=f*, the least cost solution sharing the current path from s to n would have cost

    g(n)+h*(n)>=g(n)+h(n)=f(n)>f*

Iterative Improvement Search

- Another approach to search involves starting with an initial guess at a solution and gradually improving it until it is one.
- Some examples:
  - Hill Climbing
  - Simulated Annealing
  - Genetic algorithm

Hill Climbing on a Surface of States

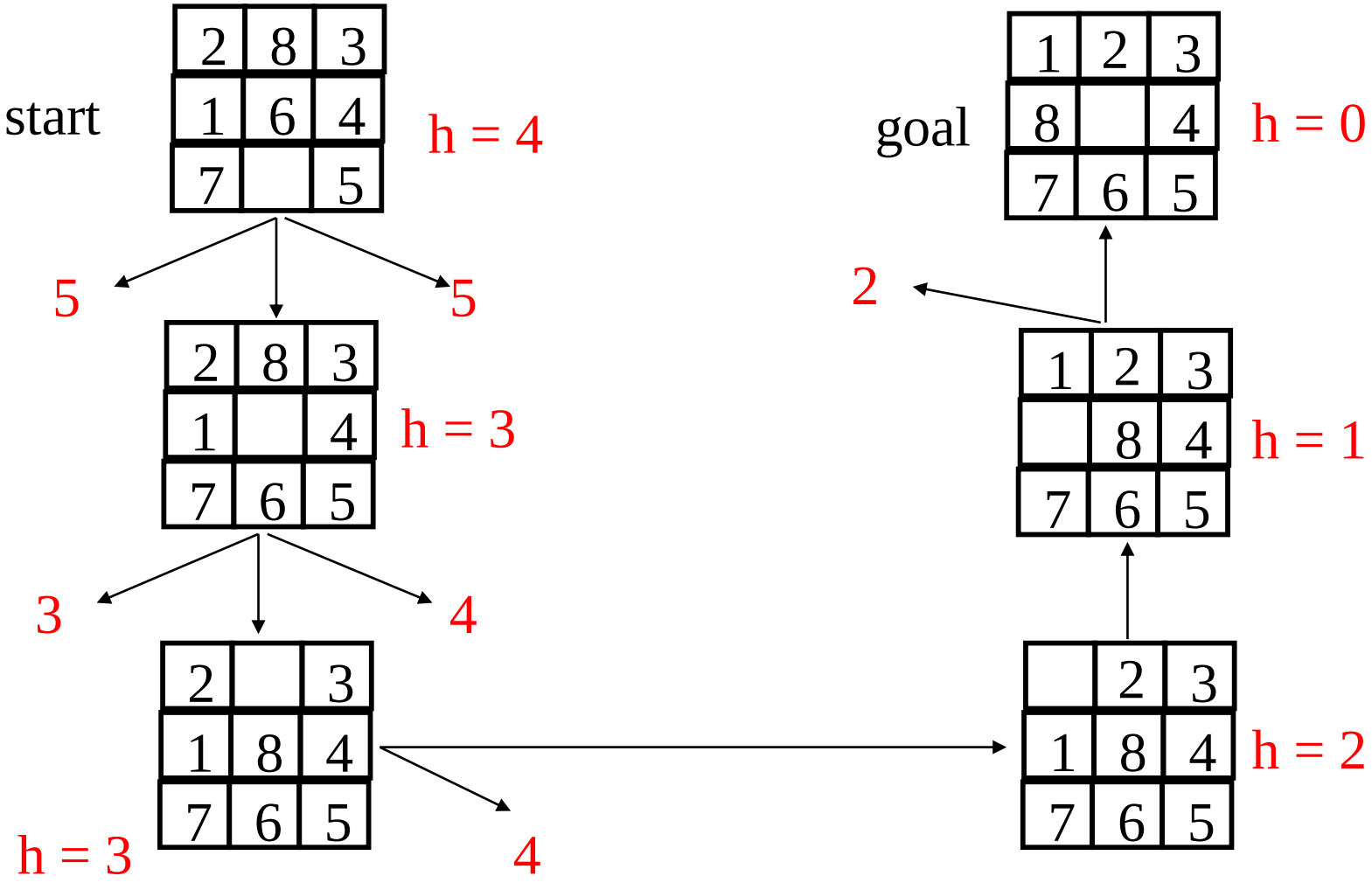


Height Defined by
Evaluation Function

Hill Climbing Search

- If there exists a successor n’ for the current state n such that
  - h(n’) < h(n)
  - h(n’) <= h(t) for all the successors t of n,
- then move from n to n’. Otherwise, halt at n.
- Looks one step ahead to determine if any successor is better than the current state; if there is, move to the best successor.
- Similar to Greedy search in that it uses h, but does not allow backtracking or jumping to an alternative path since it doesn’t “remember” where it has been.
- OPEN = {current-node}
- Not complete since the search will terminate at "local minima," "plateaus," and "ridges."

Hill climbing example



f(n) = (number of tiles out of place)

Drawbacks of Hill-Climbing

- Problems:
  - **Local Maxima**:
  - **Plateaus:** the space has a broad flat plateau with a singularity as it's maximum
  - **Ridges:** steps to the North, East, South and West may go down, but a step to the NW may go up.
- Remedy:
  - Random Restart.
  - Multiple HC searches from different start states
- Some problems spaces are great for Hill Climbing and others horrible.

Simulated Annealing

- Simulated Annealing (SA) exploits an analogy between the way in which a metal cools and freezes into a minimum energy crystalline structure (the annealing process) and the search for a minimum in a more general system.
- Each state n has an energy value f(n), where f is an evaluation function
- SA can avoid becoming trapped at local minimum energy state by introducing **randomness** into search so that it not only accepts changes that decreases state energy, but also some that increase it.
- SAs use a a control parameter T, which by analogy with the original application is known as the system "**temperature**" irrespective of the objective function involved.
- T starts out high and gradually (very slowly) decreases toward 0.

Algorithm outline for SA

current := a randomly generated state;
T := T_0 /* initial temperature T0 >>0 */
forever do
    if T <= T_end **then return** current;
    next := a randomly generated new state; /* next != current */
    ΔE = f(next) – f(current);
    current := next with probability $p_{\Delta E} = \frac{1}{1+e^{\Delta E / T}}$;
    T := schedule(T); /* reduce T by a cooling schedule */

Commonly used cooling schedule

- T := T * alpha where 0 < alpha < 1 is a cooling constant
- T_k := T_0 / log (1+k)

Observations with SA

- T ΔE $\dfrac{1}{1+e^{\Delta E/T}}$ E

$>>1$	positive	≈ 0.5	increase
	negative	≈ 0.5	decrease
≈ 0	positive	≈ 0.0	no change
	negative	≈ 1.0	decrease

- Probability of the system is at any particular state depends on the state's energy (Boltzmann distribution)

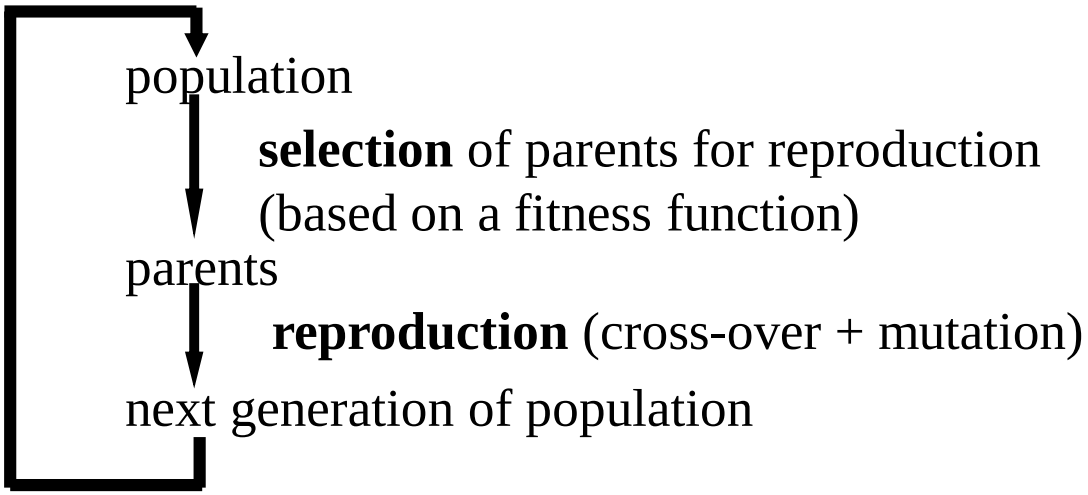
$$P_\alpha = \frac{1}{Z} e^{-E_\alpha / T}, \text{ where } Z = \sum_\alpha e^{-E_\alpha / T}$$

- If time taken to cool is infinite then

$$\frac{P(\text{stays at minimum energy state g*})}{P(\text{stays at any other state n})} = \infty$$

Genetic Algorithms

- Emulating biological evolution (survival of the fittest by natural selection process)



- Population of individuals (each individual is represented as a string of symbols: genes and chromosomes)
- Fitness function: estimates the goodness of individuals
- Selection: only those with high fitness function values are allowed to reproduce
- Reproduction:
  crossover allows offsprings to inherit good features from their parents
  mutation (randomly altering genes) may produce new (hopefully good) features
  bad individuals are throw away when the limit of population size is reached
- To ensure good results, the population size must be large
- starts out high and gradually (very slowly) decreases toward 0.

Informed Search Summary

- **Best-first search** is general search where the minimum cost nodes (according to some measure) are expanded first.
- **Greedy search** uses minimal estimated cost h(n) to the goal state as measure. This reduces the search time, but the algorithm is neither complete nor optimal.
- **A* search** combines uniform-cost search and greedy search: f(n) = g(n) + h(n) and handles state repetitions and h(n) never overestimates.
  - A* is complete, optimal and optimally efficient, but its space complexity is still bad.
  - The time complexity depends on the quality of the heuristic function.
  - IDA* and SMA* reduce the memory requirements of A*.
- **Hill-climbing algorithms** keep only a single state in memory, but can get stuck on local optima.
- **Simulated annealing** escapes local optima, and is complete and optimal given a long enough cooling schedule (in probability).
- **Genetic algorithms** escapes local optima, and is complete and optimal given a long enough cooling schedule (in probability).

Game Playing

Chapter 5

Why study games

- Fun
- Clear criteria for success
- Offer an opportunity to study problems involving {hostile, adversarial, competing} agents.
- Interesting, hard problems which require minimal “initial structure”
- Games often define very large search spaces
  - chess 10*120 nodes
- Historical reasons
- Different from games studied in game theory

Typical Case (perfect games)

- 2-person game
- Players alternate moves
- Zero-sum-- one players loss is the other's gain.
- Perfect information -- both players have access to complete information about the state of the game. No information is hidden from either player.
- No chance (e.g., using dice) involved
- Clear rules for legal moves (no uncertain position transition involved)
- Well-defined outcomes (W/L/D)
- Examples: Tic-Tac-Toe, Checkers, Chess, Go, Nim, Othello
- Not: Bridge, Solitaire, Backgammon, ...

How to play a game

- A way to play such a game is to:
  - Consider all the legal moves you can make.
  - Each move leads to a new board configuration (position).
  - Evaluate each resulting position and determine which is best
  - Make that move.
  - Wait for your opponent to move and repeat?
- Key problems are:
  - Representing the “board”
  - Generating all legal next boards
  - Evaluating a position
  - Look ahead

Game Trees

- Problem spaces for typical games represented as trees.
- Root node represents the “board” configuration at which a decision must be made as to what is the **best single move to make next**. (not necessarily the initial configuration)
- **Evaluator function** rates a board position. f(board) (a real number).
- Arcs represent the possible legal moves for a player (no costs associates to arcs
- Terminal nodes represent end-game configurations (the result must be one of “win”, “lose”, and “draw”, possibly with numerical payoff)

- If it is my turn to move, then the root is labeled a "MAX" node; otherwise it is labeled a "MIN" node indicating my opponent's turn.
- Each level of the tree has nodes that are all MAX or all MIN; nodes at level i are of the opposite kind from those at level i+1
- Complete game tree: includes all configurations that can be generated from the root by legal moves (all leaves are terminal nodes)
- Incomplete game tree: includes all configurations that can be generated from the root by legal moves to a given depth (looking ahead to a given steps)

Evaluation Function

- **Evaluation function** or **static evaluator** is used to evaluate the "goodness" of a game position.
  - Contrast with heuristic search where the evaluation function was a non-negative estimate of the cost from the start node to a goal and passing through the given node.
- The zero-sum assumption allows us to use a single evaluation function to describe the goodness of a board with respect to both players.
  - **f(n) > 0**: position n good for me and bad for you.
  - **f(n) < 0**: position n bad for me and good for you
  - **f(n) near 0**: position n is a neutral position.
  - **f(n) >> 0**: win for me.
  - **f(n) << 0**: win for you..

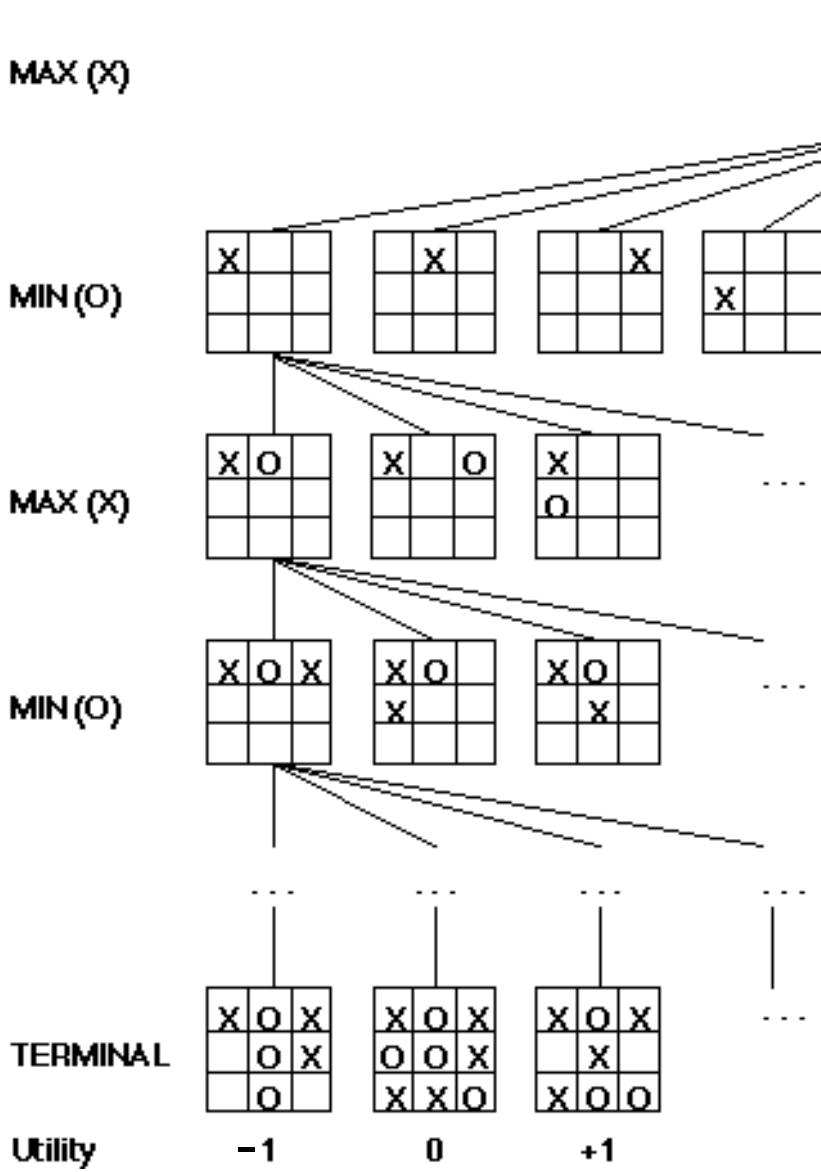
- Evaluation function is a heuristic function, and it is where the domain experts' knowledge resides.
- Example of an Evaluation Function for Tic-Tac-Toe:

  f(n) = [# of 3-lengths open for me] - [# of 3-lengths open for you]

  where a 3-length is a complete row, column, or diagonal.
- Alan Turing's function for chess
  - **f(n) = w(n)/b(n)** where w(n) = sum of the point value of white's pieces and b(n) is sum for black.
- Most evaluation functions are specified as a weighted sum of position features:

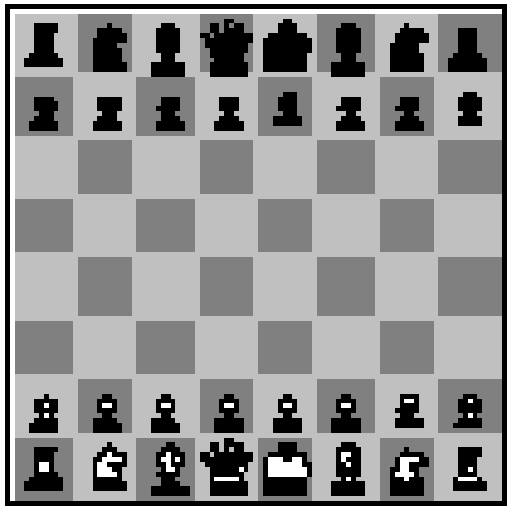
  f(n) = w1*feat1(n) + w2*feat2(n) + ... + wn*featk(n)
- Example features for chess are piece count, piece placement, squares controlled, etc.
- Deep Blue has about 6,000 features in its evaluation function.

An example (partial) game tree for Tic-Tac-Toe

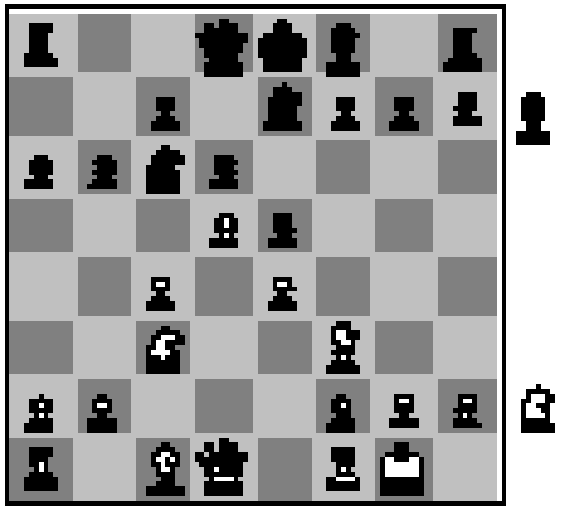


- f(n) = +1 if the position is a win for X.
- f(n) = -1 if the position is a win for O.
- f(n) = 0 if the position is a draw.

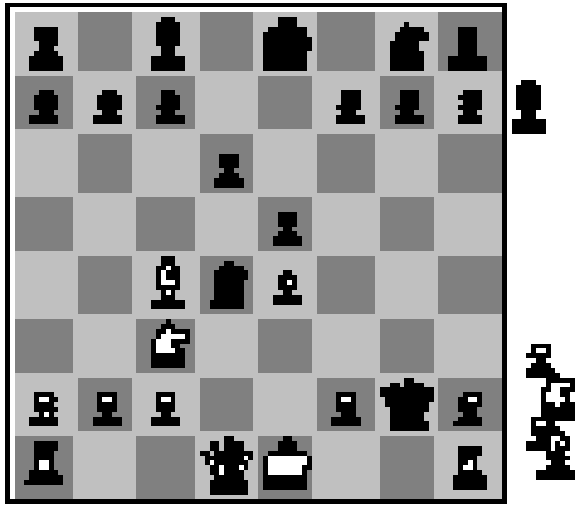
Some Chess Positions and their Evaluations



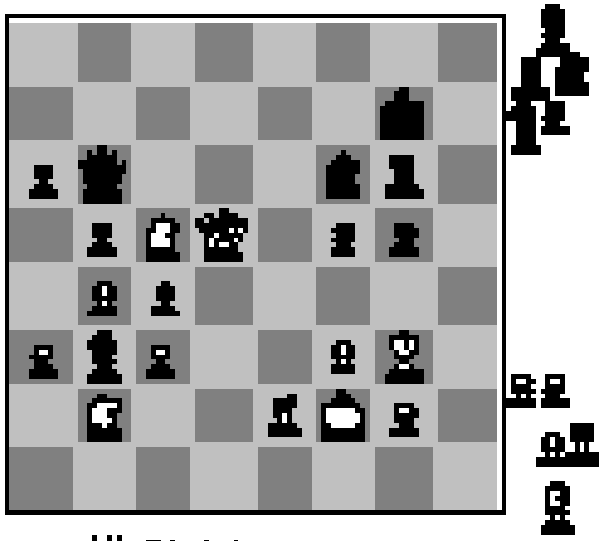
(a)



(b)



(c)



(d)

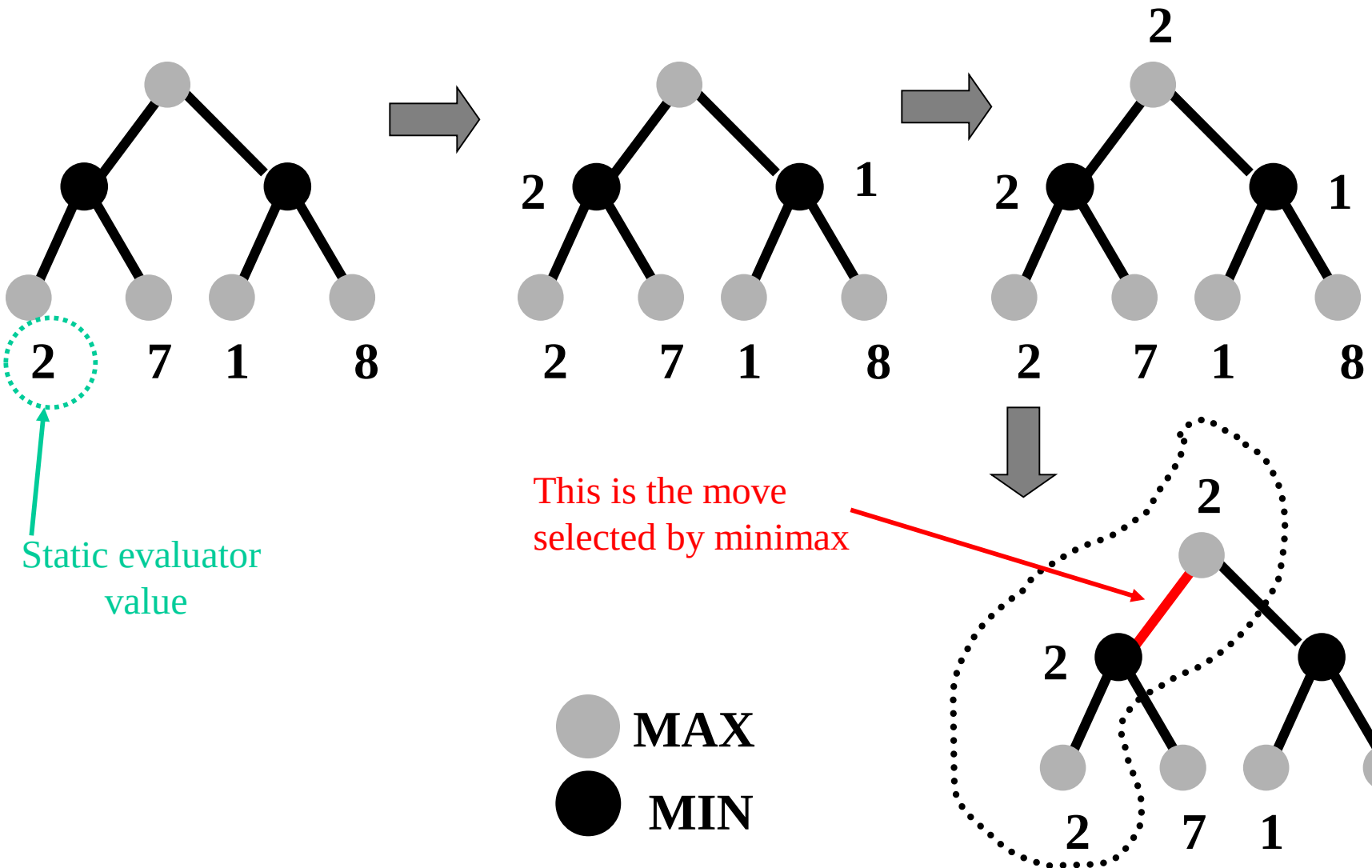
Minimax Rule

- Goal of game tree search: to determine **one move** for Max player that **maximizes** the **guaranteed payoff** for a given game tree for MAX
  - Regardless of the moves the MIN will take
- The value of each node (Max and MIN) is determined by (back up from) the values of its children
- MAX plays the worst case scenario:
  - Always assume MIN to take moves to maximize his pay-off (i.e., to minimize the pay-off of MAX)
- For a MAX node, the backed up value is the **maximum** of the values associated with its children
- For a MIN node, the backed up value is the **minimum** of the values associated with its children

Minimax procedure

- Create start node as a MAX node with current board configuration
- Expand nodes down to some depth (i.e., ply) of lookahead in the game.
- Apply the evaluation function at each of the leaf nodes
- Obtain the “back up" values for each of the non-leaf nodes from its children by Minimax rule until a value is computed for the root node.
- Pick the operator associated with the child node whose backed up value determined the value at the root as the move for MAX

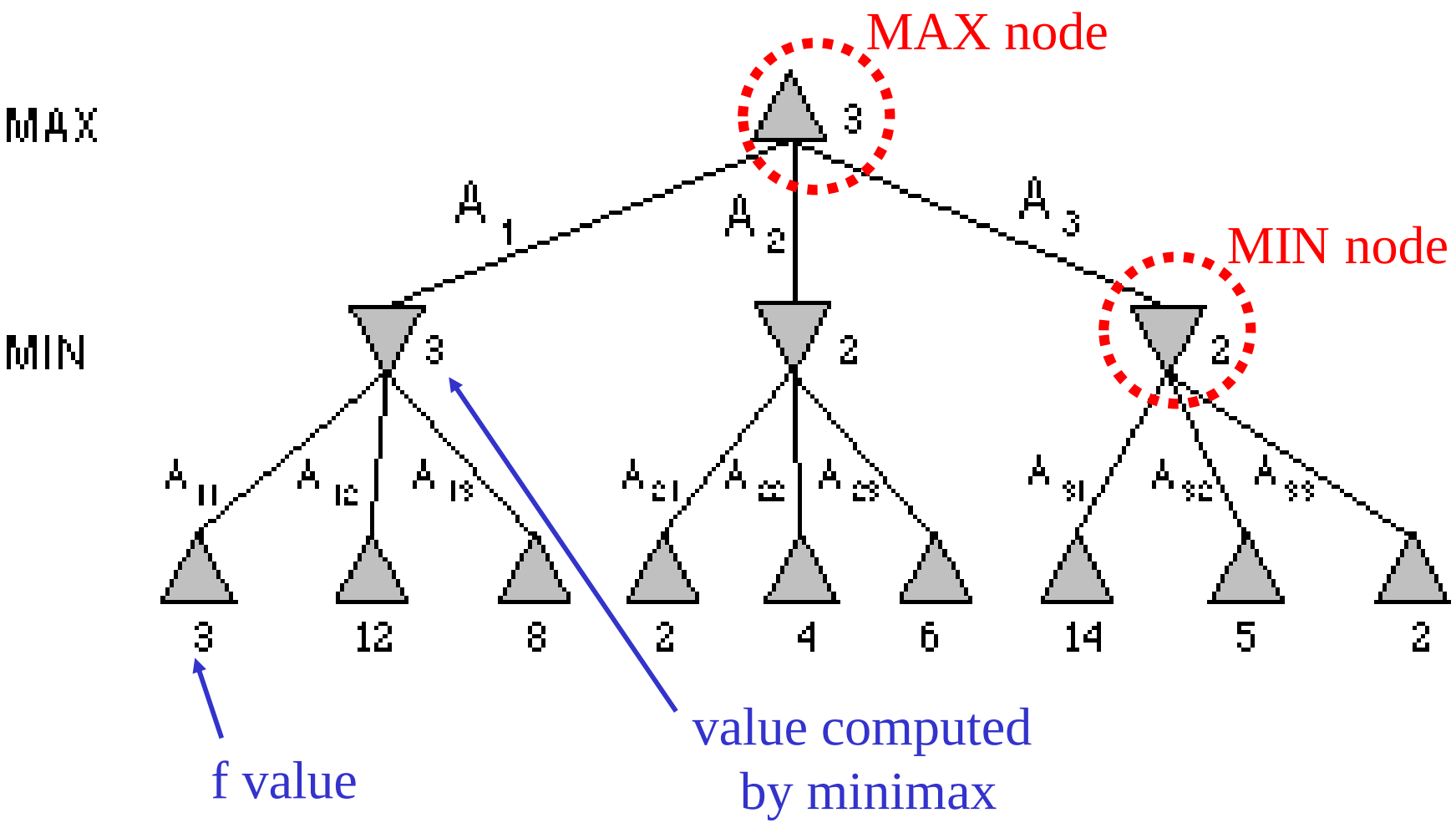
Minimax Search



Comments on Minimax search

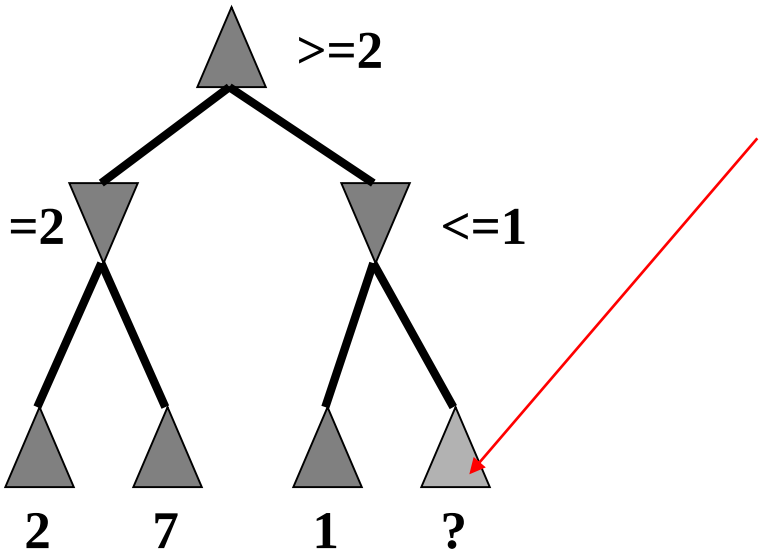
- The search is **depth-first** with the given depth (ply) as the limit
  - Time complexity: O(b^d)
  - Linear space complexity
- Performance depends on
  - Quality of evaluation functions (domain knowledge)
  - Depth of the search (computer power and search algorithm)
- Different from ordinary state space search
  - Not to search for a solution but for one move only
  - No cost is associated with each arc
  - MAX does not know how MIN is going to counter each of his moves
- Minimax rule is a basis for other game tree search algorithms

Minimax Tree



Alpha-beta pruning

- We can improve on the performance of the minimax algorithm through alpha-beta pruning.
- Basic idea: *"If you have an idea that is surely bad, don't take the time to see how truly awful it is."* -- Pat Winston



- We don't need to compute the value at this node.
- No matter what it is it can't effect the value of the root node.

Alpha-beta pruning

- Traverse the search tree in depth-first order
- At each **Max** node n, **alpha(n)** = maximum value found so far
  - Start with **-**infinity and only increase
  - Increases if a child of n returns a value greater than the current alpha
  - Serve as a tentative lower bound of the final pay-off
- At each **Min** node n, **beta(n)** = minimum value found so far
  - Start with infinity and only decrease
  - Decreases if a child of n returns a value less than the current beta
  - Serve as a tentative upper bound of the final pay-off

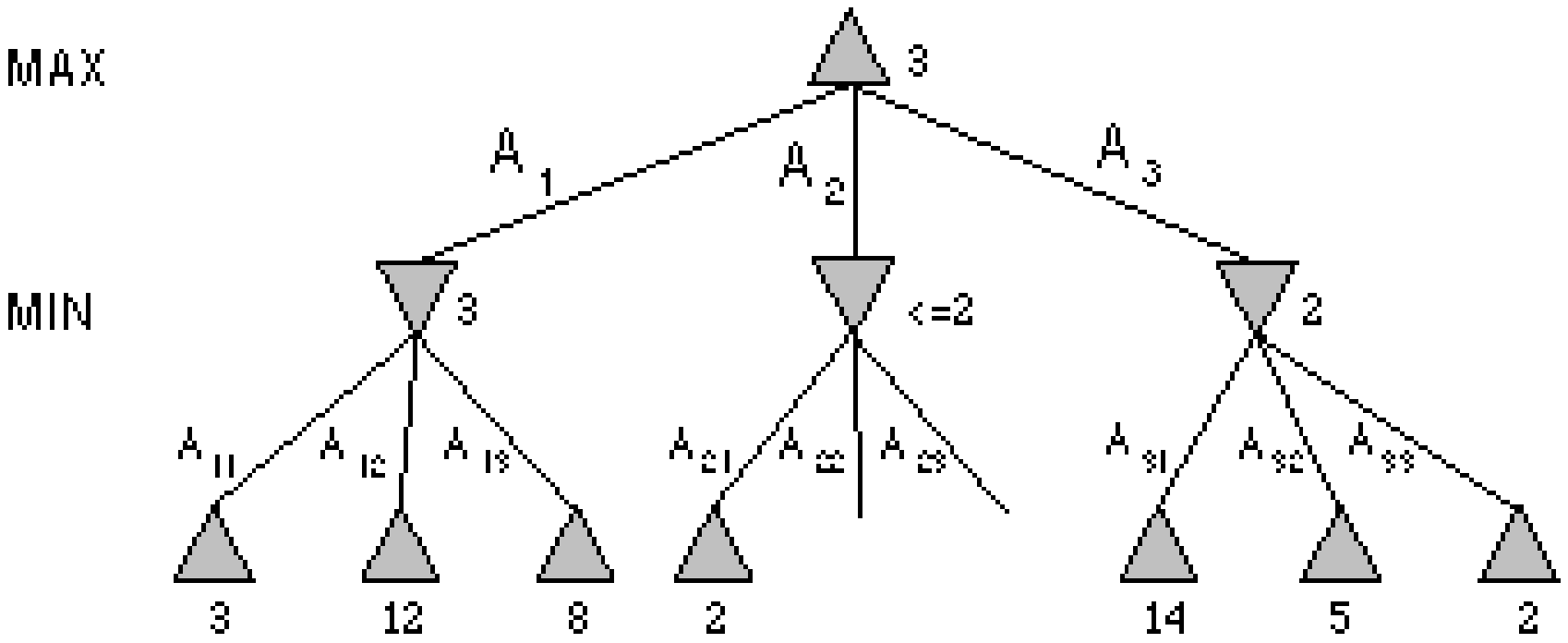
Alpha-beta pruning

- **Alpha cutoff**: Given a Max node n, cutoff the search below n (i.e., don't generate or examine any more of n's children) if alpha(n) >= beta(n)
  (alpha increases and passes beta from below)

- **Beta cutoff.:** Given a Min node n, cutoff the search below n (i.e., don't generate or examine any more of n's children) if beta(n) <= alpha(n)
  (beta decreases and passes alpha from above)

- Carry alpha and beta values down during search
  Pruning occurs whenever alpha >= beta

Alpha-beta search



Alpha-beta algorithm

initiation :
$\alpha := \infty$
$\beta := \infty$

- Two functions recursively call each other

```
function MAX-value (n, alpha, beta)
  if n is a leaf node then return f(n);
  for each child n' of n do
      alpha :=max{alpha, MIN-value(n', alpha, beta)};
      if alpha >= beta then return beta   /* pruning */
  end{do}
  return alpha

function MIN-value (n, alpha, beta)
  if n is a leaf node then return f(n);
  for each child n' of n do
      beta :=min{beta, MAX-value(n', alpha, beta)};
      if beta <= alpha then return alpha   /* pruning */
  end{do}
  return beta
```

Effectiveness of Alpha-beta pruning

- Alpha-Beta is guaranteed to compute the same value for the root node as computed by Minimax.
- **Worst case:** NO pruning, examining O(b^d) leaf nodes, where each node has b children and a d-ply search is performed
- **Best case:** examine only O(b^(d/2)) leaf nodes.
  - You can search twice as deep as Minimax! Or the branch factor is b^(1/2) rather than b.
- **Best case** is when each player's best move is the leftmost alternative, i.e. at MAX nodes the child with the largest value generated first, and at MIN nodes the child with the smallest value generated first.
- In Deep Blue, they found empirically that Alpha-Beta pruning meant that the average branching factor at each node was about 6 instead of about 35-40

Games of Chance

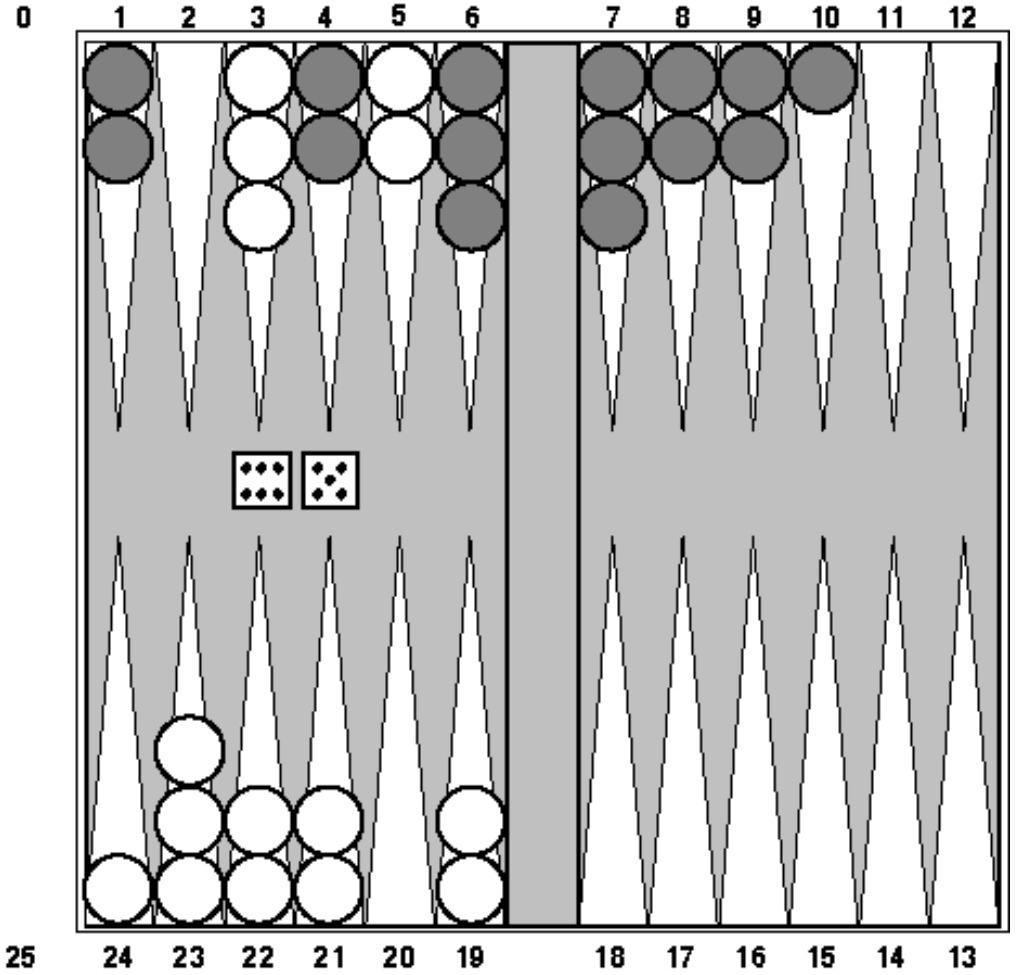
• Backgammon is a two player game with **uncertainty**.

•Players roll dice to determine what moves to make.

•White has just rolled *5 and 6* and had four legal moves:

- 5-10, 5-11
- 5-11, 19-24
- 5-10, 10-16
- 5-11, 11-16

•Such games are good for exploring decision making in adversarial problems involving skill and luck.



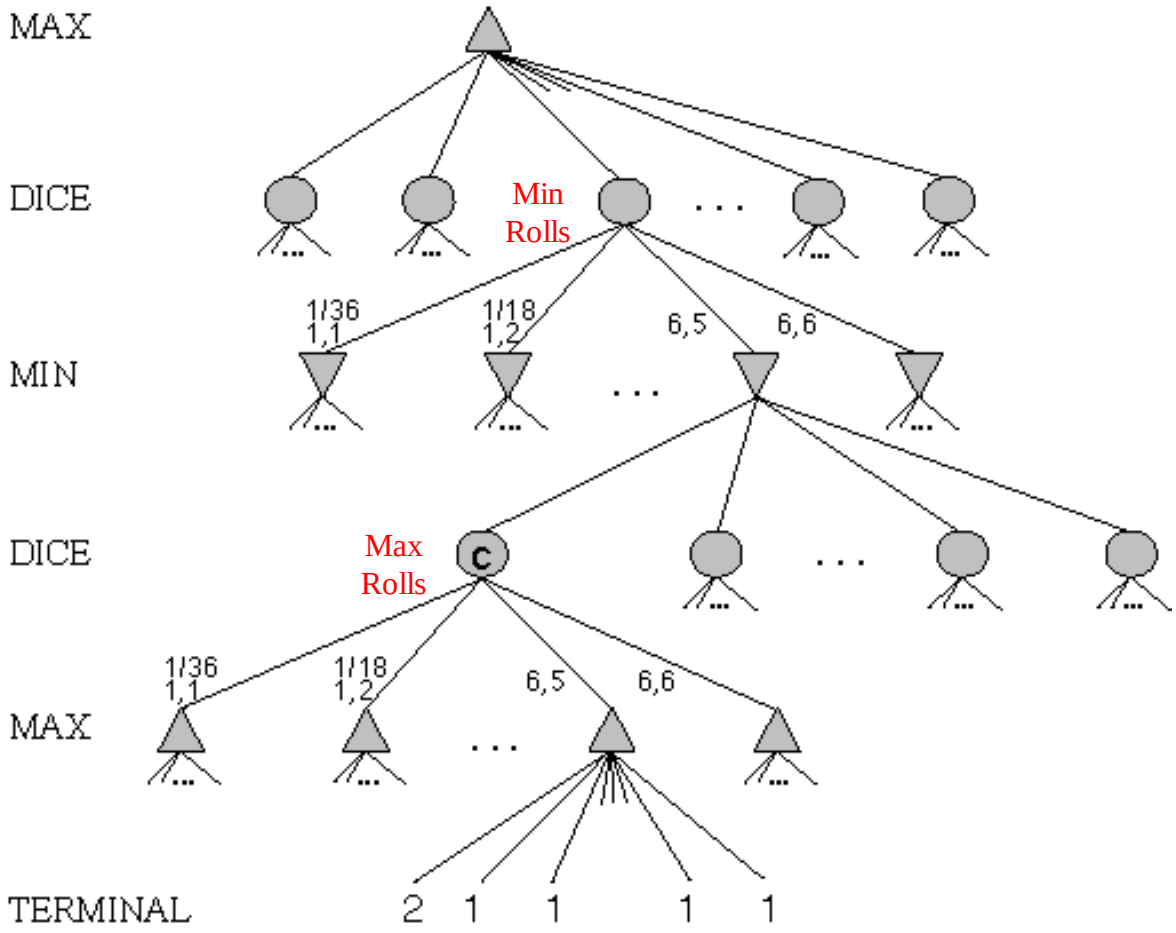
Game Trees with Chance Nodes

- **Chance nodes** (shown as circles) represent the dice rolls.
- Each chance node has 21 distinct children with a probability associated with each.
- We can use minimax to compute the values for the MAX and MIN nodes.
- Use **expected values** for chance nodes.
- For chance nodes over a max node, as in C, we compute:

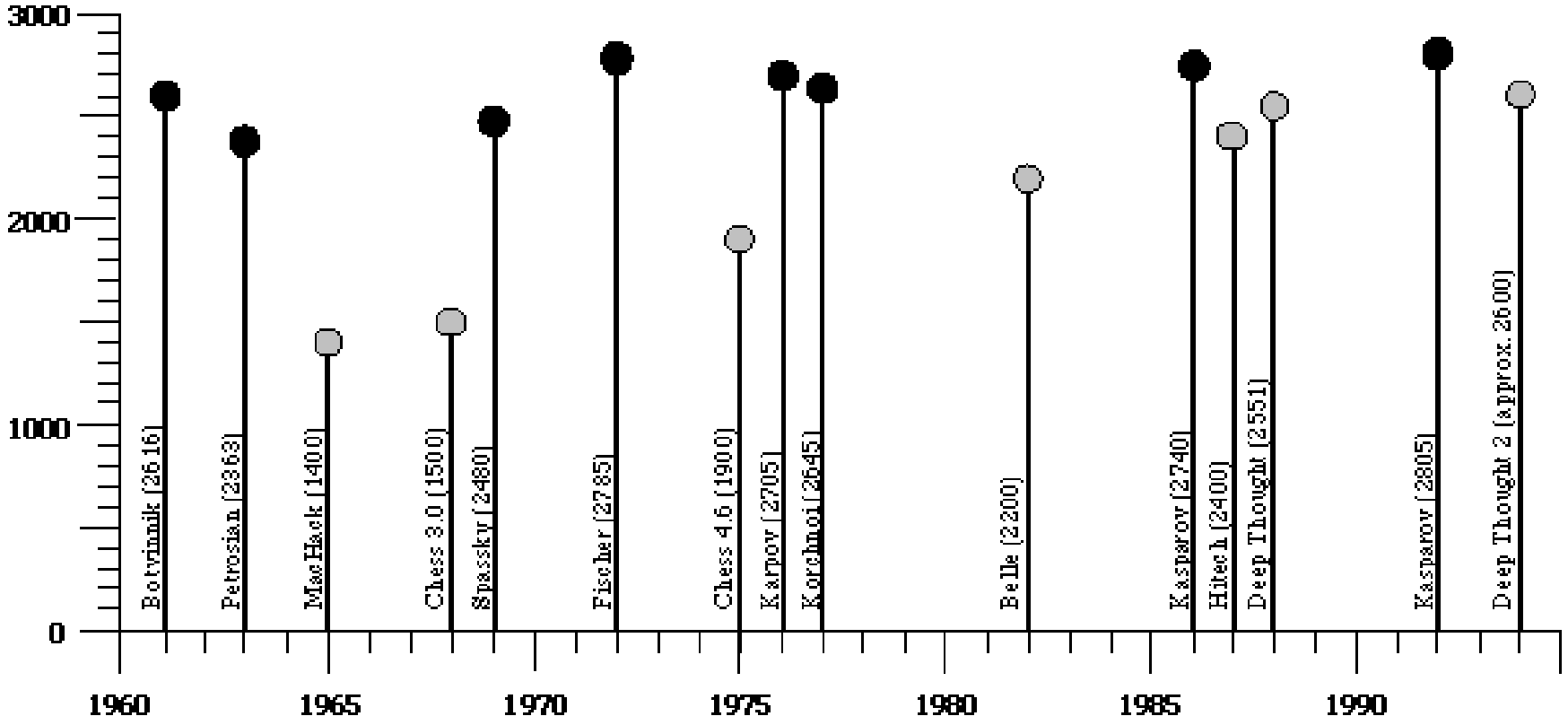
epectimax(C) = $\text{Sum}_i(P(d_i) * \text{maxvalue}(i))$

- For chance nodes over a min node compute:

epectimin(C) = $\text{Sum}_i(P(d_i) * \text{minvalue}(i))$

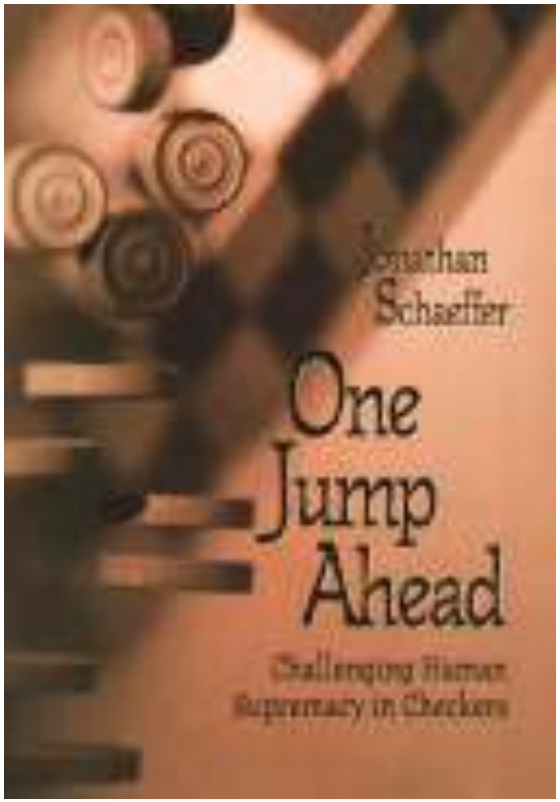


Ratings of Human and Computer Chess Champions

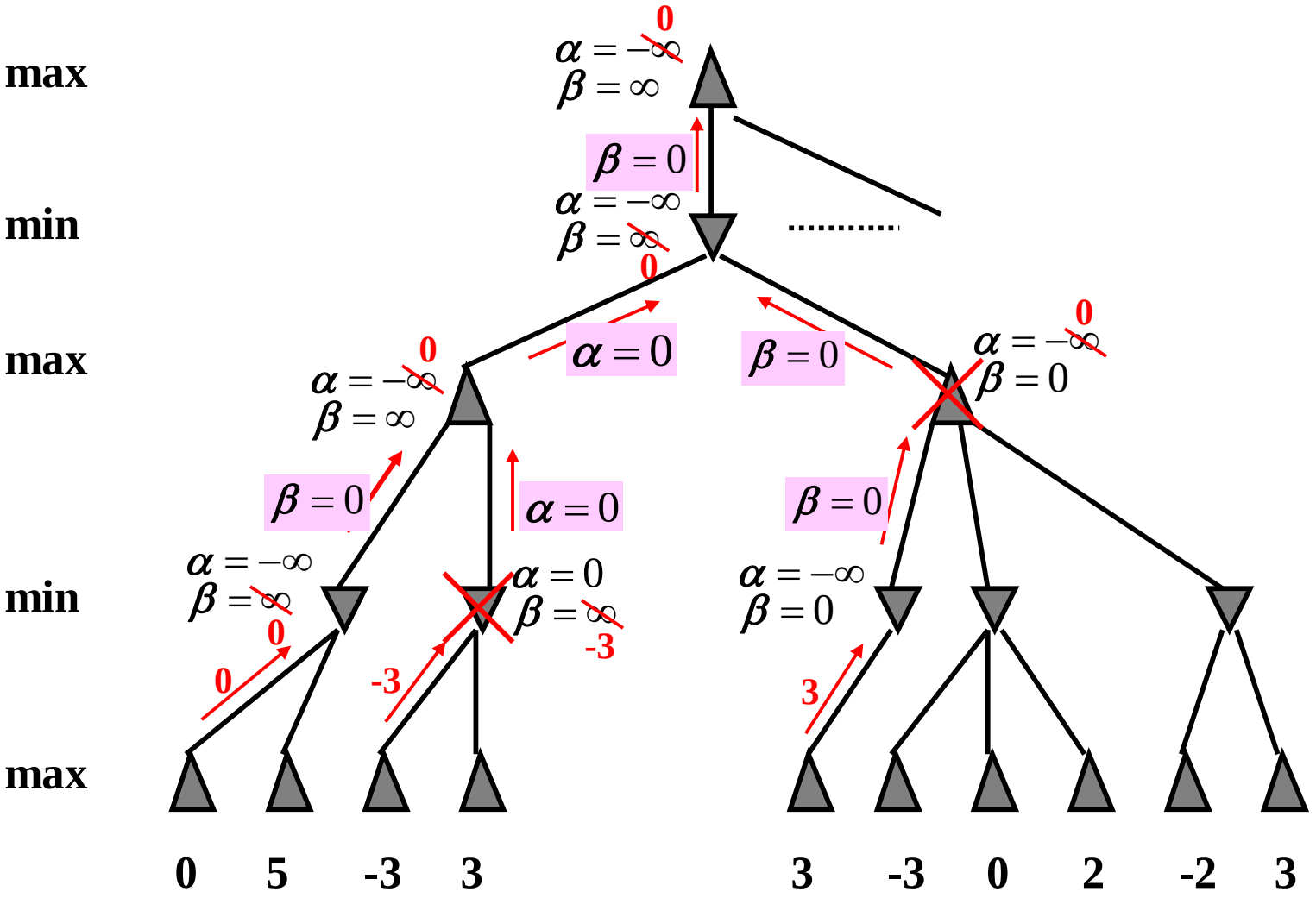


Chinook

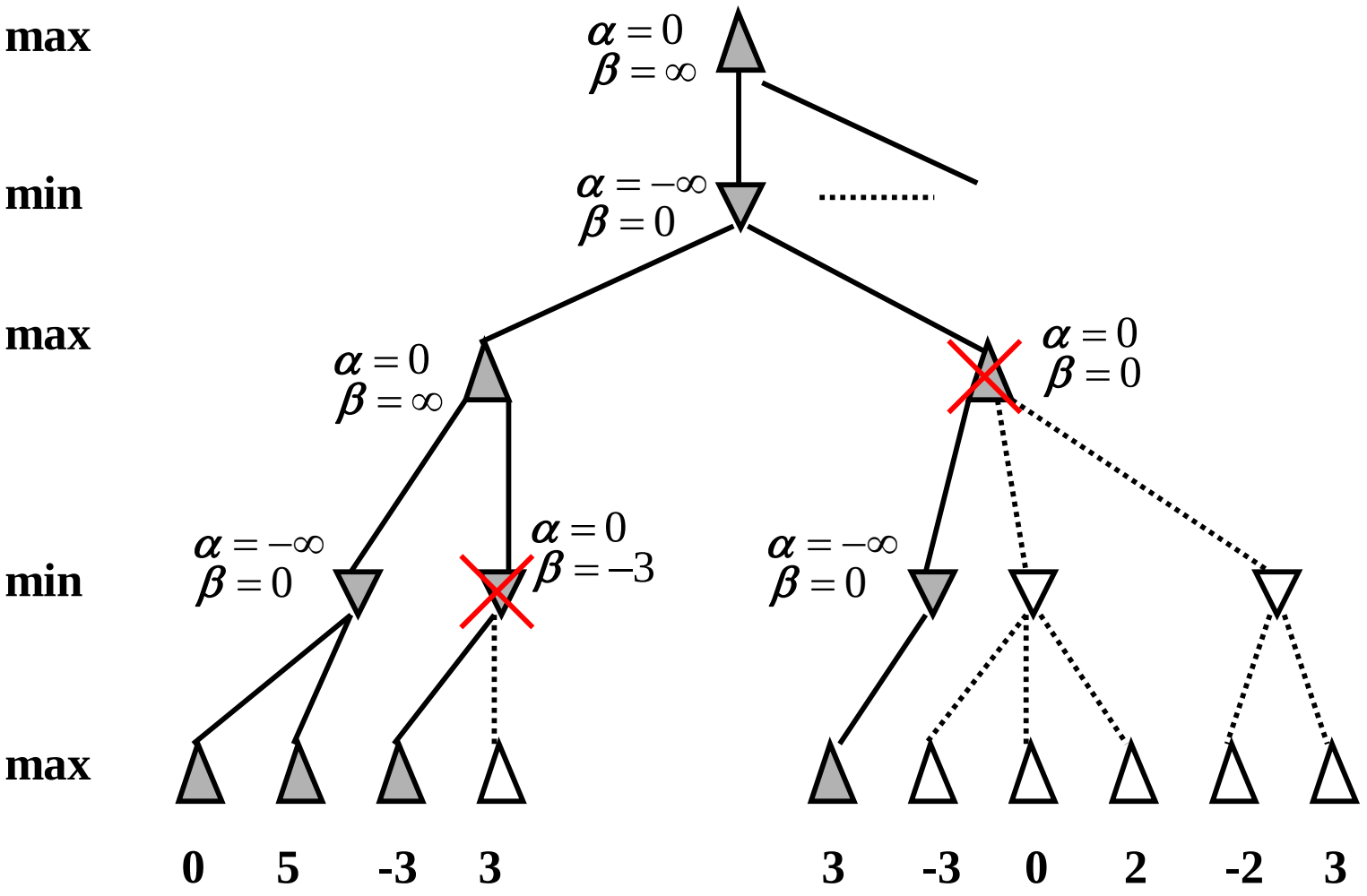
- Chinook is the World Man-Machine Checkers Champion developed by researchers at the University of Alberta.
- It earned this title by competing in human tournaments, winning the right to play for the (human) world championship, and eventually defeating the best players in the world.
- Visit <http://www.cs.ualberta.ca/~chinook/> to play Chinook over the Internet.
- Read “One Jump Ahead: Challenging Human Supremacy in Checkers” Jonathan Schaeffer, University of Alberta (496 pages, Springer. $34.95, 1998).



An example of Alpha-beta pruning



Final tree



Agents that Reason Logically

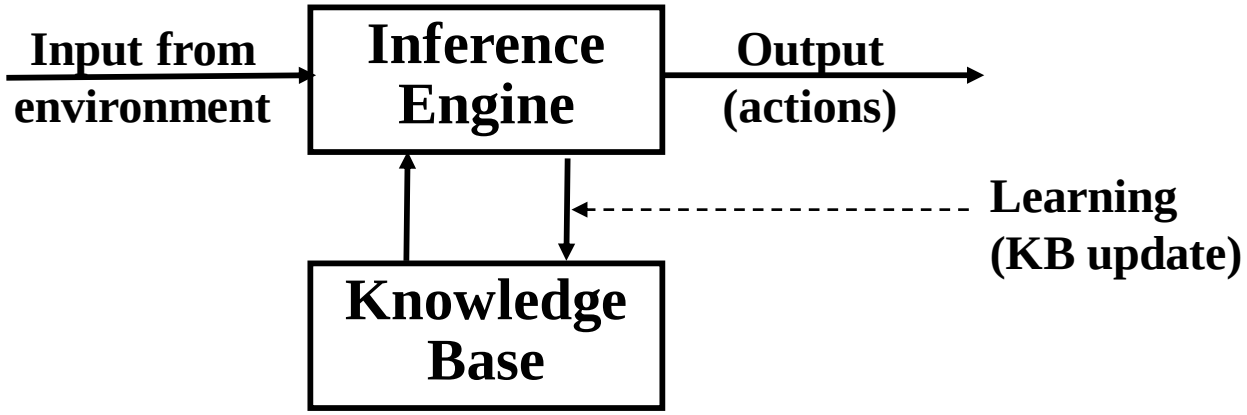
Chapter 6

Some material adopted from notes by Tim Finin, Andreas Geyer-Schulz and Chuck Dyer

A Knowledge-Based Agent

- A knowledge-based agent consists of a knowledge base (KB) and an inference engine (IE).
- A knowledge-base is a set of representations of what one knows about the world (objects and classes of objects, the fact about objects, relationships among objects, etc.)
- Each individual representation is called a **sentence**.
- The sentences are expressed in a **knowledge representation language**.
- Examples of sentences
  - The moon is made of green cheese
  - If A is true then B is true
  - A is false
  - All humans are mortal
  - Confucius is a human

- The Inference engine derives new sentences from the input and KB
- The inference mechanism depends on representation in KB
- The agent operates as follows:
  1. It receives percepts from environment
  2. It computes what action it should perform (by IE and KB)
  3. It performs the chosen action (some actions are simply inserting inferred new facts into KB).



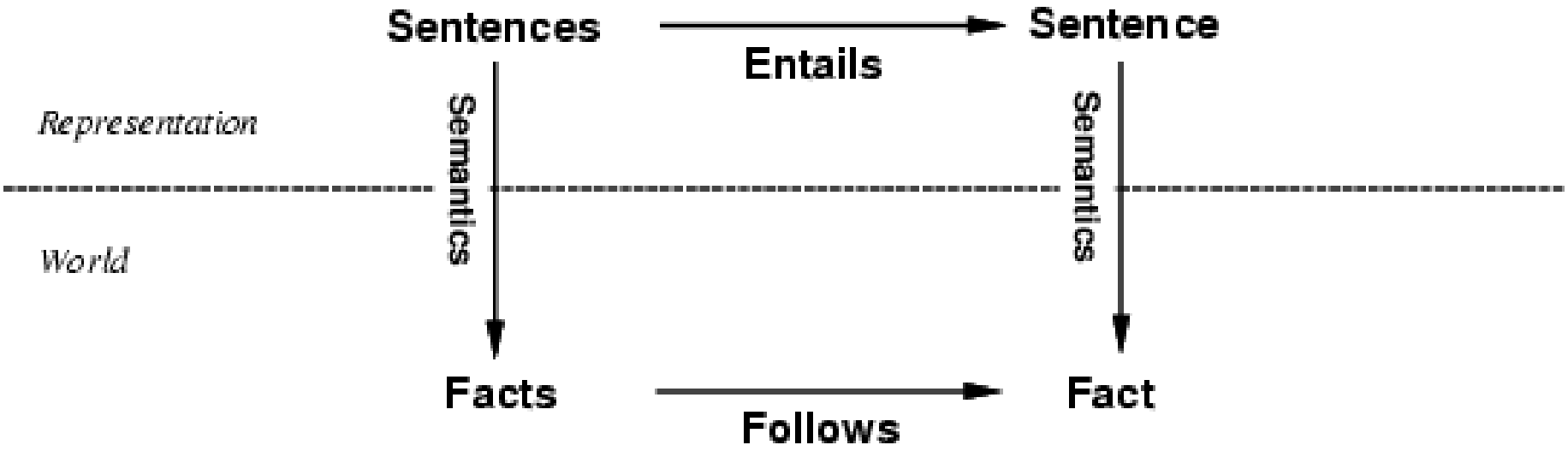
KB can be viewed at different levels

- **Knowledge Level.**
  - The most abstract level -- describe agent by saying what it knows.
  - Example: A taxi agent might know that the Golden Gate Bridge connects San Francisco with the Marin County.
- **Logical Level.**
  - The level at which the knowledge is encoded into sentences.
  - Example: Links(GoldenGateBridge, SanFrancisco, MarinCounty).
- **Implementation Level.**
  - The physical representation of the sentences in the logical level.
  - Example: “(Links GoldenGateBridge, SanFrancisco, MarinCounty)”

Representation, Reasoning, and Logic

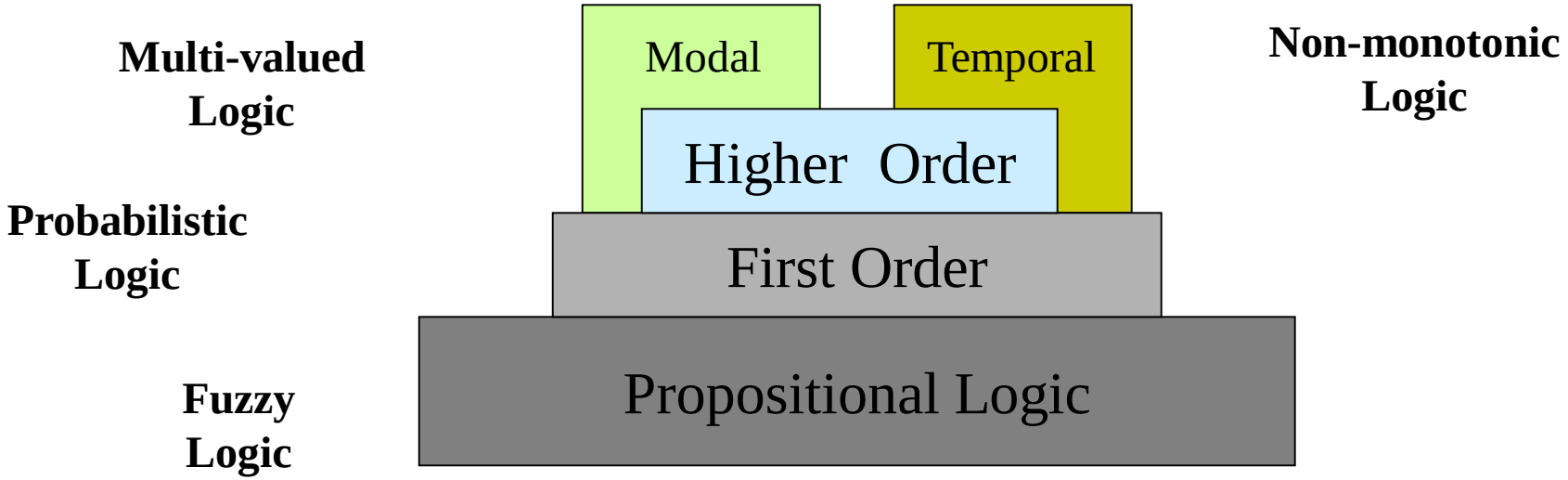
- The objective of knowledge representation is to express knowledge in a computer-tractable form, so that agents can perform well.
- A knowledge representation language is defined by:
  - Its **syntax** which defines all possible sequences of symbols that constitute sentences of the language (**grammar** to form sentences)
  - Its **semantics** determines the facts in the world to which the sentences refer (**meaning** of sentences)
    - Each sentence makes a claim about the world.
  - Its **proof theory** (inference rules and proof procedures)

The Connection between Sentences and Facts



Semantics maps sentences in logic to facts in the world.
The property of one fact following from another is mirrored
by the property of one sentence being entailed by (inferred from) another.

Logic as a KR language



Propositional Logic: Syntax

- **Symbols:**

  Logical constants: true (T), false (F)

  Propositional symbols: P, Q, S, ...

  logical connectives:

  ∧ ...conjunction (and) ∨ ...disjunction (or)
  ~ ...negation (not) => ...implication (if)
  <=> ...logical equivalence (if and only if)

  Wrapping **parentheses**: (…)

- A **proposition** (denoted by a proposition symbol) is a declarative statement which can be either true or false but not both or neither.
  - The moon is made of green cheese (F)
  - UMBC is closer to Baltimore than to Washington, DC (T)
  - *P = NP* (truth unknown)

- **Sentence**
  1. T or F itself is a sentence
  2. Individual proposition symbols P, Q, ... are sentences
  3. If S is a sentence, so is (S)
  4. If S1 and S2 are sentences, so are

     S1 ∧ S2, S1∨ S2, S1 => S2, S1 <=> S2, ~ S1
  5. Nothing else is a sentence
- **Order of precedence** of logical connectors

  ~ , ∧, ∨, => , <=>
- Minimum set of logical connectors

  (~ , ∧), or (~ , ∨)
- Atomic sentences: T, F, P, Q, ...
- Literals: atomic sentences and their negations

A BNF Grammar of Sentences in Propositional Logic

```
S := <Sentence> ;
<Sentence> := <AtomicSentence> | <ComplexSentence> ;
<AtomicSentence> := "T" | "F" |
                    "P" | "Q" | "S" ;
<ComplexSentence> := "(" <Sentence> ")" |
          <Sentence> <Connective> <Sentence> |
          ~<Sentence> ;
<Connective> := ∧ | ∨ | => | <=> ;
<literal> := <AtomicSentence> | ~ <AtomicSentence>
```

Examples of PL sentences

- P means "It is hot"
- Q means "It is humid"
- R means "It is raining"
- P ^ Q => R

  "If it is hot and humid, then it is raining"
- Q => P

  "If it is humid, then it is hot"
- Q

  "It is humid."

Propositional Logic (PL): Semantics

- Need an **interpretation** of symbols for a given set of sentences
  - Proposition symbols do not have meaning by themselves
  - An interpretation connects proposition symbols to a statement about the world (which may be true or false in that world)
  - An interpretation in PL can be defined as an **assignment of truth values** to all proposition symbols involved
  - There are many interpretations for a given set of sentences (2^n if they involve n distinct proposition symbols)
  - Example:

    I_1: P: it is humid (T) Q: it is hot (T) P ^ Q is true

    I_2: P: moon is made of green cheese (F)

    Q: I am happy (T) P ^ Q is false

- Give a meaning to each logical connectives
  - A connective is a function (boolean),
  - It does not depend on any particular interpretations (universal to all PL sentences
  - It can best be defined by a truth table
- Truth value of each sentence can then calculated

Truth tables for logical connectives

P	Q	$\sim P$	$P \wedge Q$	$P \vee Q$	$P => Q$	$P <=> Q$
T	T	F	T	T	T	T
T	F	F	F	T	F	F
F	T	T	F	T	T	F
F	F	T	F	F	T	T

- Implication
  - Approximating causal relationships
  - Many logic theoreticians are not happy about the current definition for P = F

- Equivalence laws in PL (both sides have the same truth table)
  - $P \Rightarrow Q \equiv \sim P \vee Q$
  - $P \Leftrightarrow Q \equiv (P \Rightarrow Q) \wedge (Q \Rightarrow P)$
  - Distribution law

    $P \wedge (Q \vee R) \equiv P \wedge Q \vee P \wedge R$

    $P \vee (Q \wedge R) \equiv (P \vee Q) \wedge (P \vee R)$
  - De Morgan's law

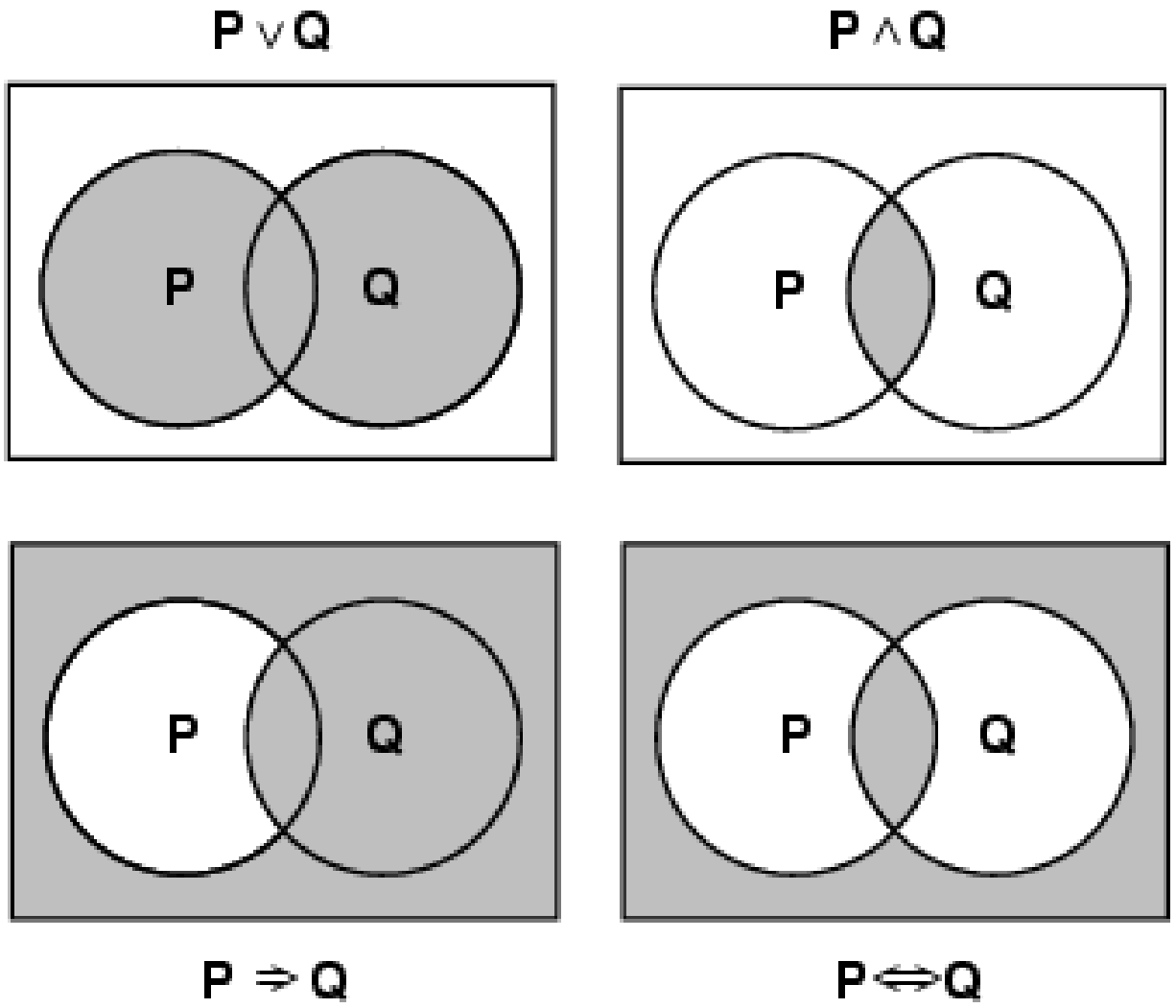
    $\sim(P \vee Q \vee R) \equiv \sim P \wedge \sim Q \wedge \sim R$

    $\sim(P \wedge Q \wedge R) \equiv \sim P \vee \sim Q \vee \sim R$

Some terms

- A **model** is an interpretation of a set of sentences such that every sentence is *True*. A model is just a formal mathematical structure that "stands in" for the world.
- A sentence is **satisfiable** if it is T*rue* under *some* interpretation(s)
- A **valid sentence** or **tautology** is a sentence that is *True* under *all* possible interpretations, no matter what the world is actually like or what the semantics is, e.g.,
- An **inconsistent sentence** or **contradictio**n is a sentence that is *False* under *all* interpretations. The world is never like what it describes, e.g.,
- **P entails Q** (Q is a **logical consequence** of P, Q **logically follows** P), written P |= Q, means that whenever P is True, so is Q. In other words, all models of P are also models of Q.

Models of Complex Sentences (Venn diagrams)



Inference Rule

- **Logical Inference** is used to create new sentences X that logically follow from a given set of sentences S (in KB and from input), i.e., S |= X
- This kind of inference is also known as deduction
- **Use truth table**: Whether S |= X holds can be determined by whether S => X is a tautology
  - S |= X holds iff every interpretation I that makes S true also makes X true

    S => X is a tautology iff every interpretation I that makes S true also makes X true
  - Example: P, P => Q |= Q because (P, P => Q) => Q is true under any interpretation
  - Huge truth tables for inference involving large number of sentences

- **Use inference rules**: generate new sentences X from a one or more existing sentences S. S is called the premise and X the conclusion of the rule.
- **Proof procedure**: a set of inference rules and a procedure of how to use these rules
- If X can be generated from S by proof procedure *i*, we say X is derived from S by *i*, denoted $S \vdash_i X$, or $S \vdash X$.
- **Soundness.** An inference procedure is sound if every sentence X it produces from a set of sentences S logically follows from S. (No contradiction is created).

$$\text{if } S \vdash X \text{ then } S \models X$$

- **Completeness.** A inference procedure is complete, if it is able to produce every sentence that logically follows from any give S.

$$\text{if } S \models X \text{ then } S \vdash X$$

Sound Inference Rules (deductive rules)

- Here are some examples of sound rules of inference.
- Each can be shown to be sound using a truth table -- a rule is sound if it's conclusion is true whenever the premise is true.

RULE	PREMISE	CONCLUSION
Modus Ponens	A, A => B	B
Modus Tollens	~B, A => B	~A
And Introduction	A, B	A ^ B
And Elimination	A ^ B	A
Or Introduction	A	A v B
Double Negation	~~A	A
Chaining	A => B, B => C	A => C

- Resolution rule

  Unit Resolution $A \vee B, \sim A \vdash B$

  Resolution $A \vee B, \sim B \vee C \vdash A \vee C$

  Let $\alpha_1 ... \alpha_i, \beta, \gamma_1 ... \gamma_k$ be literals. Then

  $$\alpha_1 \vee ... \vee \alpha_i \vee \beta, \ \sim \beta, \gamma_1 \vee ... \gamma_k \vdash \alpha_1 \vee ... \vee \alpha_i \vee \gamma_1 \vee ... \gamma_k$$

  – Operates on two disjunctions of literals

  – The pair of two opposite literals (β and $\sim \beta$) cancel each other, all other literals from the two disjuncts are combined to form a new disjunct as the inferred sentence

  – Resolution rule can replace all other inference rules

  Modus Ponens $A, \sim A \vee B \vdash B$

  Modus Tollens $\sim B, \sim A \vee B \vdash \sim A$

  Chaining $\sim A \vee B, \sim B \vee C \vdash \sim A \vee C$

Soundness of the Resolution Inference Rule

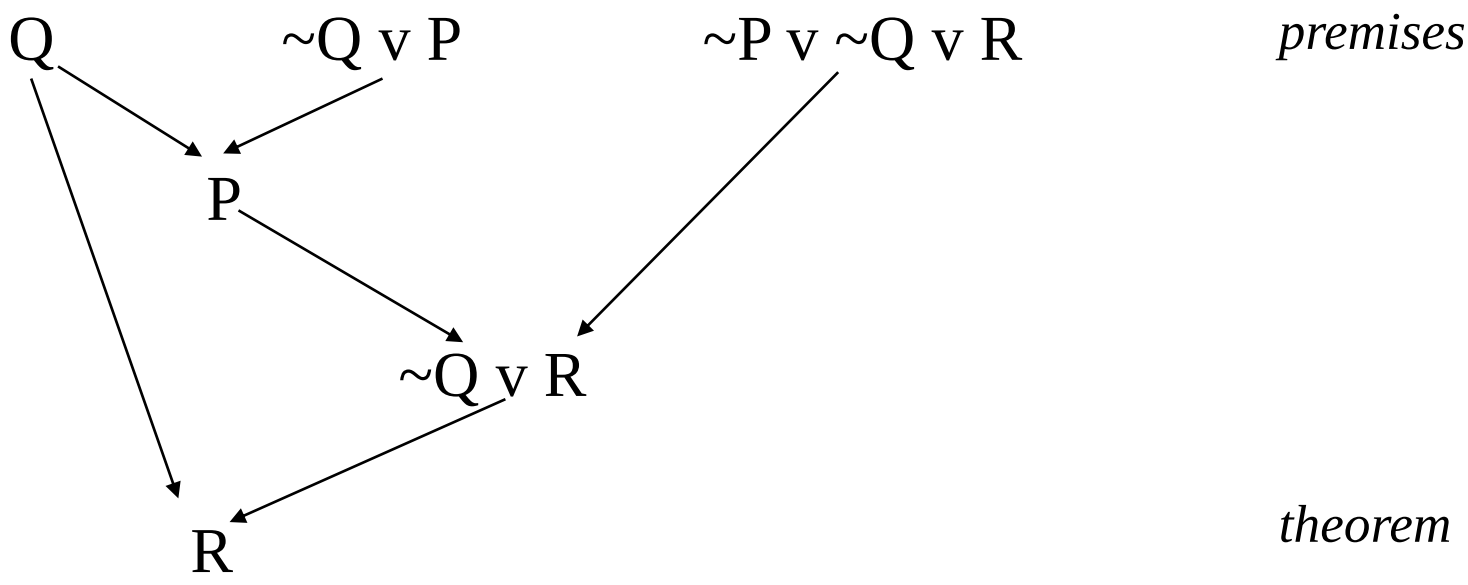
α	β	γ	$\alpha \vee \beta$	$\neg\beta \vee \gamma$	$\alpha \vee \gamma$
False	False	False	False	True	False
False	False	True	False	True	True
False	True	False	True	False	False
False	True	True	True	True	True
True	False	False	True	True	True
True	False	True	True	True	True
True	True	False	True	False	True
True	True	True	True	True	True

Proving things

- A **proof** is a sequence of sentences, where each sentence is either a premise (also called an axiom) or a sentence derived from earlier sentences in the proof by one of the rules of inference.
- The last sentence is the **theorem** (also called goal or query) that we want to prove.
- Complete proof procedure for PL has time complexity exponential to the number of premises
- Example for the "weather problem" given before.

1 Q	Premise	“It is humid”
2 Q=>P	Premise	“if it is humid, it is hot”
3 P	Modus Ponens(1,2)	“It is hot”
4 (P^Q)=>R	Premise	“If it’s hot & humid, it’s raining”
5 P^Q	And Introduction(1)	“It is hot and humid”
6 R	Modus Ponens(4,5)	“It is raining”

- Proof by resolution



- Theorem proving as search
  - Start node: the set of given premises/axioms (KB + Input)
  - Operator: inference rule (add a new sentence into parent node)
  - Goal: a state that contains the theorem asked to prove
  - Solution: a path from start node to a goal

Normal forms of PL sentences

- Disjunctive normal form (DNF)
  - Any sentence can be written as a disjunction of conjunctions of literals.
  - Examples: P ^ Q ^ ~R; A^B v C^D v P^Q^R; P
  - Widely used in logical circuit design (simplification)
- Conjunctive normal form (CNF)
  - Any sentence can be written as a conjunction of disjunctions of literals.
  - Examples: P v Q v ~R; (A v B) ^ (C v D) ^ (P v Q v R); P
- Normal forms can be obtained by applying equivalence laws

[(A v B) => (C v D)] => P

$\equiv$ ~[~(A v B) v (C v D)] v P

$\equiv$ [~~(A v B) ^ ~(C v D)] v P

$\equiv$ [(A v B)^(~C ^ ~D)] v P

$\equiv$ (A v B v P)^(~C^~D v P)

$\equiv$ **(A v B v P)^(~C v P)^(~D v P)** a CNF

Horn Sentences

- A Horn sentences is a **disjunction** of literals with at most one of these literals being non-negated (positive).

  ~P1 v ~P2 v ~P3 ... v ~Pn v Q; or alternatively

  P1 ^ P2 ^ P3 ... ^ Pn => Q; (an implication or if-then rule)

- Some properties of Horn sentences
  - P => Q ^ R ≡ (P => Q) ^ (P => R)
  - P v Q => R ≡ (P => R) ^ (Q => R)
  - P => Q v R cannot be represented as Horn sentences
- We will expand Horn sentences to Horn clauses in first order predicate logic later and give it more discussion then
- As we will see later, Horn clauses make automating logical inference easier.

PL is Too Weak a Representational Language

- Consider the problem of representing the following information:
  - Every person is mortal. (S1)
  - Confucius is a person. (S2)
  - Confucius is mortal. (S3)
- S3 is clearly a logical consequence of S1 and S2. But how can these sentences be represented using PL so that we can infer the third sentence from the first two?
- We can use symbols P, Q, and R to denote the three propositions, but this leads us to nowhere because knowledge important to infer R from P and Q (i.e., relationship between being a human and mortality, and the membership relation between Confucius and human class) is not expressed in a way that can be used by inference rules

- Alternatively, we can use symbols for parts of each sentence
  - P = "person"; M = "mortal"; C = "Confucius"
  - The above 3 sentences can be roughly represented as:

    S2: C => P; S1: P => M; S3: C => M.
  - Then S3 is entailed by S1 and S2 by the chaining rule.
- Bad semantics
  - "Confucius" (and "person" and "mortal") are not PL sentences (**not a declarative statement**) and cannot have a truth value.
  - What does P => M mean?
- We need infinite distinct symbols X for individual persons, and infinite implications to connect these X with P (person) and M (mortal) because we need a unique symbol for each individual.

  Person_1 => P; person_1 => M;
  Person_2 => P; person_2 => M;
  ... ...
  Person_n => P; person_n => M

Weakness of PL

- Hard to identify "**individuals**." E.g., Mary, 3
  - Individuals cannot be PL sentences themselves.
- Difficult to directly and clearly talk about **properties** of individuals or **relations** between individuals (hard to connect individuals to class properties).
  - E.g., property of being a human implies property of being mortal
- Generalizations, patterns, regularities can't easily be represented.
  - All members of a class have this property
  - Some member of a class have this property
- A better representation is needed to capture the relationship (and distinction) between objects and classes, including properties belonging to classes and individuals
- First-Order Logic (abbreviated FOL or FOPC) is expressive enough to concisely represent this kind of situation by separating classes and individuals
  - Explicit representation of individuals and classes, x, Mary, 3, persons.
  - Adds relations, variables, and quantifiers, e.g.,
    - *"Every person is mortal"* Forall X: person(X) => mortal(X)
    - *"There is a white alligator"* There exists some X: Alligator(X) ^ white(X)

Basic Knowledge Representation in First Order Logic

Chapter 7

Some material adopted from notes
by **Tim Finin**
And Andreas Geyer-Schulz

First Order (Predicate) Logic (FOL)

- First-order logic is used to model the world in terms of
  - **objects** which are things with individual identities

    e.g., individual students, lecturers, companies, cars ...
  - **properties** of objects that distinguish them from other objects

    e.g., mortal, blue, oval, even, large, ...
  - **classes** of objects (often defined by properties)

    e.g., human, mammal, machine, ...
  - **relations** that hold among objects

    e.g., brother of, bigger than, outside, part of, has color, occurs after, owns, a member of, ...
  - **functions** which are a subset of the relations in which there is only one ``value'' for any given ``input''.

    e.g., father of, best friend, second half, one more than ...

Syntax of FOL

- **Predicates:** P(x[1], ..., x[n])
  - P: **predicate name**; (x[1], ..., x[n]): **argument list**
  - A special function with range = {T, F};
  - Examples: human(x), /* x is a human */
    father(x, y) /* x is the father of y */
  - When all arguments of a predicate is assigned values (said to be **instantiated**), the predicate becomes either true or false, i.e., it becomes a proposition. Ex. Father(Fred, Joe)
  - A predicate, like a membership function, defines a set (or a class) of objects
- **Terms** (arguments of predicates must be terms)
  - **Constants** are terms (e.g., Fred, a, Z, "red", etc.)
  - **Variables** are terms (e.g., x, y, z, etc.), a variable is **instantiated** when it is assigned a constant as its value
  - **Functions** of terms are terms (e.g., f(x, y, z), f(x, g(a)), etc.)
  - A term is called a **ground** term if it does not involve variables
  - Predicates, though special functions, are not terms in FOL

- **Quantifiers**

  Universal quantification ∀ (or ***forall***)

  – (∀x)P(x) means that P holds for **all** values of x in the domain associated with that variable.

  – E.g., (∀x) dolphin(x) => mammal(x)

  (∀x) human(x) => mortal(x)

  – Universal quantifiers often used with "implication (=>)" to form "rules" about properties of a class

  (∀x) student(x) => smart(x) (All students are smart)

  – Often associated with English words “all”, “everyone”, “always”, etc.

  – You rarely use universal quantification to make blanket statements about every individual in the world (because such statement is hardly true)

  (∀x)student(x)^smart(x) means everyone in the world is a student and is smart.

Existential quantification ∃

– (∃x)P(x) means that P holds for **some** value(s) of x in the domain associated with that variable.

– E.g., (∃x) mammal(x) ^ lays-eggs(x)

(∃x) taller(x, Fred)

(∃x) UMBC-Student (x) ^ taller(x, Fred)

– Existential quantifiers usually used with "^ (and)" to specify a list of properties about an individual.

(∃x) student(x) ^ smart(x) (there is a student who is smart.)

– A common mistake is to represent this English sentence as the FOL sentence:

(∃x) student(x) => smart(x)

It also holds if there no student exists in the domain because

student(x) => smart(x) holds for any individual who is not a student.

– Often associated with English words "someone", "sometimes", etc.

Scopes of quantifiers

- Each quantified variable has its scope
  - (∀x)[human(x) => (∃y) [human(y) ^ father(y, x)]
  - All occurrences of x within the scope of the quantified x refer to the same thing.
  - Use different variables for different things
- Switching the order of universal quantifiers does not change the meaning:
  - (∀x)(∀y)P(x,y) <=> (∀**y**)(∀x)P(x,y), can write as (∀x,y)P(x,y)
- Similarly, you can switch the order of existential quantifiers.
  - (∃**x**)(∃y)P(x,y) <=> (∃y)(∃**x**)P(x,y)
- Switching the order of universals and existential does change meaning:
  - Everyone likes someone: (∀x)(∃y)likes(x,y)
  - Someone is liked by everyone: (∃y)(∀x) likes(x,y)

Sentences are built from terms and atoms

- A **term** (denoting a individual in the world) is a constant symbol, a variable symbol, or a function of terms.
- An **atom** (atomic sentence) is a predicate P(x[1], ..., x[n])
  - Ground atom: all terms in its arguments are ground terms (does not involve variables)
  - A ground atom has value true or false (like a proposition in PL)
- A **literal** is either an atom or a negation of an atom
- A **sentence** is an atom, or,
  - ~P, P v Q, P ^ Q, P => Q, P <=> Q, (P) where P and Q are sentences
  - If P is a sentence and x is a variable, then (∀x)P and (∃x)P are sentences
- A **well-formed formula** (**wff**) is a sentence containing no "free" variables. i.e., all variables are "bound" by universal or existential quantifiers.

  (∀x)P(x,y) has x bound as a universally quantified variable, but y is free.

A BNF for FOL Sentences

```
S := <Sentence> ;
<Sentence> := <AtomicSentence> |
          <Sentence> <Connective> <Sentence> |
          <Quantifier> <Variable>,... <Sentence> |
          ~ <Sentence> |
          "(" <Sentence> ")";
<AtomicSentence> := <Predicate> "(" <Term>, ... ")" |
                    <Term> "=" <Term>;
<Term> := <Function> "(" <Term>, ... ")" |
          <Constant> |
          <Variable>;
<Connective> := ^ | v | => | <=>;
<Quantifier> := ∃ | ∀;
<Constant> := "A" | "X1" | "John" | ... ;
<Variable> := "a" | "x" | "s" | ... ;
<Predicate> := "Before" | "HasColor" | "Raining" | ... ;
<Function> := "Mother" | "LeftLegOf" | ... ;
<Literal> := <AutomicSetence> | ~ <AutomicSetence>
```

Translating English to FOL

- **Every gardener likes the sun.**
  (∀x) gardener(x) => likes(x,Sun)
- **Not Every gardener likes the sun.**
  ~((∀x) gardener(x) => likes(x,Sun))
- **You can fool some of the people all of the time.**
  (∃x)(∀t) person(x) ^ time(t) => can-be-fooled(x,t)
- **You can fool all of the people some of the time.**
  (∀x)(∃t) person(x) ^ time(t) => can-be-fooled(x,t)
  (the time people are fooled may be different)
- **You can fool all of the people at some time.**
  (∃t)(∀x) person(x) ^ time(t) => can-be-fooled(x,t)
  (all people are fooled at the same time)
- **You can not fool all of the people all of the time.**
  ~((∀x)(∀t) person(x) ^ time(t) => can-be-fooled(x,t))
- **Everyone is younger than his father**
  (∀x) person(x) => younger(x, father(x))

- **All purple mushrooms are poisonous**.

  (∀x) (mushroom(x) ^ purple(x)) => poisonous(x)
- **No purple mushroom is poisonous.**

  ~(∃x) purple(x) ^ mushroom(x) ^ poisonous(x)

  (∀x) (mushroom(x) ^ purple(x)) => ~poisonous(x)
- **There are exactly two purple mushrooms**.

  (∃x)(Ey) mushroom(x) ^ purple(x) ^ mushroom(y) ^ purple(y) ^ ~(x=y) ^

  (∀z) (mushroom(z) ^ purple(z)) => ((x=z) v (y=z))
- **Clinton is not tall.**

  ~tall(Clinton)
- **X is above Y if X is directly on top of Y or there is a pile of one or more other objects directly on top of one another starting with X and ending with Y.**

  (∀x)(∀y) above(x,y) <=> (on(x,y) v (∃z) (on(x,z) ^ above(z,y)))

Example: A simple genealogy KB by FOL

- **Build a small genealogy knowledge base by FOL that**
  - contains facts of immediate family relations (spouses, parents, etc.)
  - contains definitions of more complex relations (ancestors, relatives)
  - is able to answer queries about relationships between people
- **Predicates:**
  - parent(x, y), child (x, y), father(x, y), daughter(x, y), etc.
  - spouse(x, y), husband(x, y), wife(x,y)
  - ancestor(x, y), descendent(x, y)
  - relative(x, y)
- **Facts:**
  - husband(Joe, Mary), son(Fred, Joe)
  - spouse(John, Nancy), male(John), son(Mark, Nancy)
  - father(Jack, Nancy), daughter(Linda, Jack)
  - daughter(Liz, Linda)
  - etc.

- **Rules for genealogical relations**
  - (∀x,y) parent(x, y) <=> child (y, x)
    (∀x,y) father(x, y) <=> parent(x, y) ^ male(x) (similarly for mother(x, y))
    (∀x,y) daughter(x, y) <=> child(x, y) ^ female(x) (similarly for son(x, y))
  - (∀x,y) husband(x, y) <=> spouse(x, y) ^ male(x) (similarly for wife(x, y))
    (∀x,y) spouse(x, y) <=> spouse(y, x) (**spouse relation is symmetric**)
  - (∀x,y) parent(x, y) => ancestor(x, y)
    (∀x,y)(∃z) parent(x, z) ^ ancestor(z, y) => ancestor(x, y)
  - (∀x,y) descendent(x, y) <=> ancestor(y, x)
  - (∀x,y)(∃z) ancestor(z, x) ^ ancestor(z, y) => relative(x, y)
    (related by common ancestry)
    (∀x,y) spouse(x, y) => relative(x, y) (related by marriage)
    (∀x,y)(∃z) relative(z, x) ^ relative(z, y) => relative(x, y) (**transitive**)
    (∀x,y) relative(x, y) => relative(y, x) **(symmetric**)
- **Queries**
  - ancestor(Jack, Fred) /* the answer is yes */
  - relative(Liz, Joe) /* the answer is yes */
  - relative(Nancy, Mathews)
    /* no answer in general, no if under closed world assumption */

Connections between *Forall* and *Exists*

- "It is not the case that everyone is ..." is logically equivalent to "There is someone who is NOT ..."
- "No one is ..." is logically equivalent to "All people are NOT ..."
- We can relate sentences involving forall and exists using De Morgan's laws:

  $\sim(\forall x)P(x) <=> (\exists x) \sim P(x)$

  $\sim(\exists x) P(x) <=> (\forall x) \sim P(x)$

  $(\exists x) P(x) <=> \sim(\forall x) \sim P(x)$

  $(\forall x) P(x) <=> \sim (\exists x) \sim P(x)$

- Example: **no one likes everyone**
  - $\sim (\exists x)(\forall y)likes(x,y)$
  - $(\forall x)(\exists y)\sim likes(x,y)$

Semantics of FOL

- **Domain M:** the set of all objects in the world (of interest)
- **Interpretation I:** includes
  - Assign each constant to an object in M
  - Define each function of n arguments as a mapping M^n => M
  - Define each predicate of n arguments as a mapping M^n => {T, F}
  - Therefore, every ground predicate with any instantiation will have a truth value
  - In general there are infinite number of interpretations because |M| is infinite
- **Define of logical connectives: ~, ^, v, =>, <=>** as in PL
- **Define semantics of (∀x) and (∃x)**
  - (∀x) P(x) is true iff P(x) is true under all interpretations
  - (∃x) P(x) is true iff P(x) is true under some interpretation

- **Model:** an interpretation of a set of sentences such that every sentence is *True*
- **A sentence is**
  - **satisfiable** if it is true under some interpretation
  - **valid** if it is true under all possible interpretations
  - **inconsistent** if there does not exist any interpretation under which the sentence is true
- **logical consequence:** S |= X if all models of S are also models of X

Axioms, definitions and theorems

- **Axioms** are facts and rules which are known (or assumed) to be true facts and concepts about a domain.
  - –Mathematicians don't want any unnecessary (dependent) axioms -- ones that can be derived from other axioms.
  - –Dependent axioms can make reasoning faster, however.
  - –Choosing a good set of axioms for a domain is a kind of design problem.
- A **definition** of a predicate is of the form "P(x) <=> S(x)" (define P(x) by S(x)) and can be decomposed into two parts
  - –Necessary description: "P(x) => S(x)" (only if)
  - –Sufficient description "P(x) <= S(x)" (if)
  - –Some concepts don't have complete definitions (e.g. person(x))
- A **theorem** S is a sentence that logically follows the axiom set A, i.e. A |= S.

Higher order logic

- FOL only allows to quantify over variables, and variables can only range over objects.
- HOL allows us to quantify over relations
- Example: (quantify over functions)

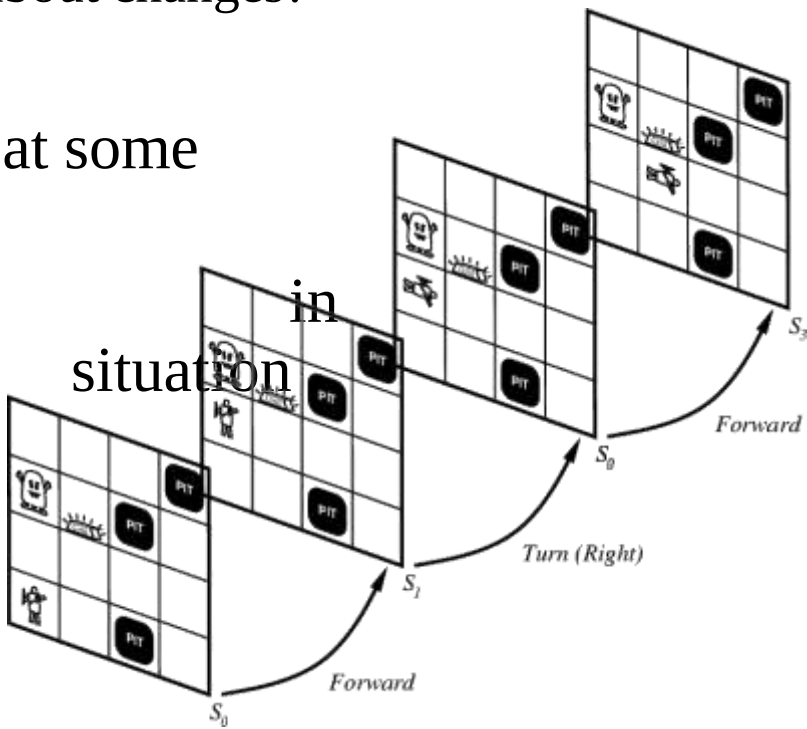
  "two functions are equal iff they produce the same value for all arguments"

  $\forall f \forall g (f = g) <=> (\forall x f(x) = g(x))$
- Example: (quantify over predicates)

  $\forall r$ transitive(r) => $(\forall xyz)$ r(x,y) ^ r(y,z) => r(x,z))
- More expressive, but undecidable.

Representing Change

- Representing change in the world in logic can be tricky.
- One way is to change the KB
  - add and delete sentences from the KB to reflect changes.
  - How do we remember the past, or reason about changes?
- **Situation calculus** is another way
- A **situation** is a snapshot of the world at some instant in time
- When the agent performs an action A in situation S1, the result is a new situation S2.



Situation Calculus

- A situation is a snapshot of the world at an interval of time when nothing changes
- Every true or false statement is made with respect to a particular situation.
  - Add situation variables to every predicate. E.g., feel(x, hungry) becomes feel(x, hungry, s0) to mean that feel(x, hungry) is true in situation (i.e., state) s0.
  - Or, add a special predicate holds(f,s) that means "f is true in situation s.” e.g., holds(feel(x, hungry), s0)
- Add a new special function called result(a,s) that maps current situation s into a new situation as a result of performing action a. For example, result(eating, s) is a function that returns the successor state in which x is no longer hungry
- Example: The action of eating could be represented by
- $(\forall x)(\forall s)(feel(x, hungry, s) \Rightarrow feel(x, not\text{-}hungry, result(eating(x), s))$

Frame problem

- An action in situation calculus only changes a small portion of the current situation
  - after eating, x is not-hungry, but many other properties related to x (e.g., his height, his relations to others such as his parents) are not change
  - Many other things unrelated to x's feeling are not changed
- Explicit copy those unchanged facts/relations from the current state to the new state after each action is inefficient (and counterintuitive)
- How to represent facts/relations that remain unchanged by certain actions is known as "**frame problem**", a very tough problem in AI
- One way to address this problem is to add **frame axioms**.
  - $(\forall x,s1,s2)P(x, s1) \wedge s2=result(a(s1)) \Rightarrow P(x, s2)$
- We may need a huge number of frame axioms

More on definitions

- A **definition** of P(x) by S(x)), denoted (∀x) P(x) <=> S(x), can be decomposed into two parts
  - Necessary description: "P(x) => S(x)" (**only if**, for P(x) being true, S(x) is necessarily true)
  - Sufficient description "P(x) <= S(x)" (**if**, S(x) being true is sufficient to make P(x) true)
- Examples: define father(x, y) by parent(x, y) and male(x)
  - parent(x, y) is a necessary (**but not sufficient**) description of father(x, y)

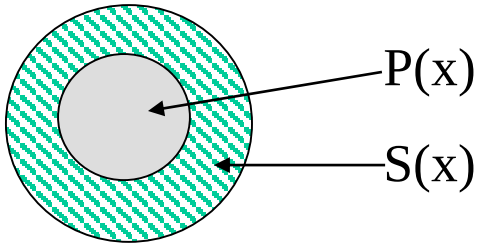
    father(x, y) => parent(x, y), parent(x, y) $\not\Rightarrow$ father(x, y)
  - parent(x, y) ^ male(x) is a necessary and sufficient description of father(x, y)

    parent(x, y) ^ male(x) <=> father(x, y)
  - parent(x, y) ^ male(x) ^ age(x, 35) is a sufficient (**but not necessary**) description of father(x, y) because

    father(x, y) $\not\Rightarrow$ parent(x, y) ^ male(x) ^ age(x, 35)

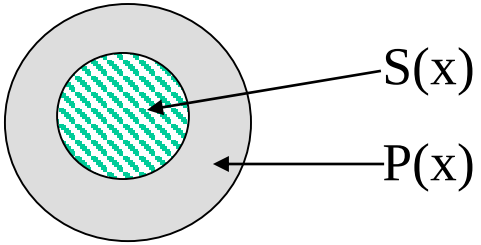
More on definitions

S(x) is a necessary condition of P(x)



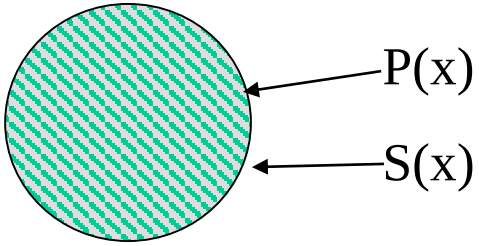
$(\forall x)\ P(x) \Rightarrow S(x)$

S(x) is a sufficient condition of P(x)



$(\forall x)\ P(x) \Leftarrow S(x)$

S(x) is a necessary and sufficient condition of P(x)



$(\forall x)\ P(x) \Leftrightarrow S(x)$

Inference in First Order Logic

Chapter 9

Some material adopted from notes
by Tim Finin,
Andreas Geyer-Schulz,
and Chuck Dyer

Inference Rules for FOL

- Inference rules for PL apply to FOL as well (Modus Ponens, And-Introduction, And-Elimination, etc.)
- New (sound) inference rules for use with **quantifiers**:
  - Universal Elimination
  - Existential Introduction
  - Existential Elimination
  - Generalized Modus Ponens (GMP)
- **Resolution**
  - Clause form (CNF in FOL)
  - Unification (consistent variable substitution)
  - Refutation resolution (proof by contradiction)

Universal Elimination (∀x) P(x) |-- P(c).

- If (∀x) P(x) is true, then P(c) is true for **any** constant c in the domain of x, i.e.,, (∀x) P(x) |= P(c).
- Replace all occurrences of x in the scope of ∀x by the **same** ground term (a constant or a ground function).
- Example: (∀x) eats(Ziggy, x) |-- eats(Ziggy, IceCream)

Existential Introduction P(c) |-- (∃x) P(x)

- If P(c) is true, so is (∃x) P(x), i.e., P(c) |= (∃x) P(x)
- Replace all instances of the given constant symbol by the same **new** variable symbol.
- Example eats(Ziggy, IceCream) |-- (∃x) eats(Ziggy, x)

Existential Elimination

- From (∃x) P(x) infer P(c), i.e., (∃x) P(x) |= P(c), where c is a new constant symbol,
  - All we know is there must be some constant that makes this true, so we can introduce a brand new one to stand in for that constant, *even though we don't know exactly what that constant refer to.*
  - Example: (∃x) eats(Ziggy, x) |= eats(Ziggy, Stuff)

- Things become more complicated when there are universal quantifiers

  $(\forall x)(\exists y)$ eats(x, y) |= $(\forall x)$eats(x, Stuff) ???

  $(\forall x)(\exists y)$ eats(x, y) |= eats(Ziggy, Stuff) ???

  – Introduce a **new** function *food_sk*(x) to stand for $\exists y$ because that y depends on x

    $(\forall x)(\exists y)$ eats(x, y) |-- $(\forall x)$eats(x, food_sk(x))

    $(\forall x)(\exists y)$ eats(x, y) |-- eats(Ziggy, food_sk(Ziggy))

  – What exactly the function food_sk(.) does is unknown, except that it takes x as its argument

- The process of existential elimination is called "*Skolemization",* and the new, unique constants (e.g., Stuff) and functions (e.g., food_sk(.)) are called skolem constants and skolem functions

Generalized Modus Ponens (GMP)

- Combines And-Introduction, Universal-Elimination, and Modus Ponens
- Ex: P(c), Q(c), (∀x)(P(x) ^ Q(x)) => R(x) |-- R(c)

  P(c), Q(c) |-- P(c) ^ Q(c) (*by and-introduction*)

  (∀x)(P(x) ^ Q(x)) => R(x)

  |-- (P(c) ^ Q(c)) => R(c) (*by universal-elimination*)

  P(c) ^ Q(c), (P(c) ^ Q(c)) => R(c) |-- R(c) (*by modus ponens*)
- All occurrences of a quantified variable must be instantiated to the same constant.

  P(a), Q(c), (∀x)(P(x) ^ Q(x)) => R(x) |-- R(c)

  because all occurrences of x must either instantiated to a or c which makes the modus ponens rule not applicable.

Resolution for FOL

- Resolution rule operates on two *clauses*
  - A clause is a **disjunction** of literals (without explicit quantifiers)
  - Relationship between clauses in KB is **conjunction**
- Resolution Rule for FOL:
  - clause C1: (l_1, l_2, ... ***l_i***, ... l_n) and

    clause C2: (l'_1, l'_2, ... ***l'_j***, ... l'_m)
  - if l_i and l'_j are two **opposite literals** (e.g., P and ~P) and their argument lists can be be made the same (**unified)** by a set of variable bindings θ = {x1/y1, ... Xk/yk} where x1, ... Xk are variables and y1, ... Yk are terms, then derive a new clause (called resolvent)

    subst((l_1, l_2, ... l_n, l'_1, l'_2, ... l'_m), θ)

    where function subst(expression, θ) returns a new expression by applying all variable bindings in θ to the original expression

We need answers to the following questions

- How to convert FOL sentences to clause form (especially how to remove quantifiers)
- How to unify two argument lists, i.e., how to find their most general unifier (**mgu**) θ
- How to determine which two clauses in KB should be resolved next (among all resolvable pairs of clauses) and how to determine a proof is completed

Converting FOL sentences to clause form

- *Clauses* are quantifier free CNF of FOL sentences
- Basic ideas
  - How to handle quantifiers
    - Careful on quantifiers with preceding negations (explicit or implicit)

      $\sim\forall x\ P(x)$ is really $\exists x\ \sim P(x)$

      $(\forall x\ P(x)) \Rightarrow (\forall y\ Q(y)) \equiv \sim(\forall x\ P(x)) \vee (\forall y\ Q(y))$

      $\equiv \exists x\ \sim P(x) \vee \forall y\ Q(y)$
    - Eliminate true existential quantifier by Skolemization
    - For true universally quantified variables, treat them as such without quantifiers
  - How to convert to CNF (similar to PL but need to work with quantifiers)

Conversion procedure

***step* 1:** remove all "=>" and "<=>" operators

(using P => Q ≡ ~P v Q and P <=> Q ≡ P => Q ^ Q => P)

***step* 2:** move all negation signs to individual predicates

(using de Morgan's law)

***step* 3:** remove all existential quantifiers ∃y

case 1: y is not in the scope of any universally quantified variable, then replace all occurrences of y by a skolem constant

case 2: if y is in scope of universally quantified variables x1, ... xi, then replace all occurrences of y by a skolem function

***step* 4:** remove all universal quantifiers ∀x (with the understanding that all remaining variables are universally quantified)

***step* 5:** convert the sentence into CNF (using distribution law, etc)

***step* 6:** use parenthesis to separate all disjunctions, then drop all v's and ^'s

Conversion examples

∀x (P(x) ^ Q(x) => R(x))

∀x ~(P(x) ^ Q(x)) v R(x) (by step 1)

∀x ~P(x) v ~Q(x) v R(x) (by step 2)

~P(x) v ~Q(x) v R(x) (by step 4)

(~P(x), ~Q(x), R(x)) (by step 6)

∃y rose(y) ^ yellow(y)

rose(c) ^ yellow(c)

(where c is a skolem constant)

(rose(c)), (yellow(c))

∀x [person(x) => ∃y (person(y) ^ father(y, x))]

∀x [~person(x) v ∃y (person(y) ^ father(y, x))] (by step 1)

∀x [~person(x) v (person(f_sk(x)) ^ father(f_sk(x), x))] (by step 3)

~person(x) v (person(f_sk(x)) ^ father(f_sk(x), x)) (by step 4)

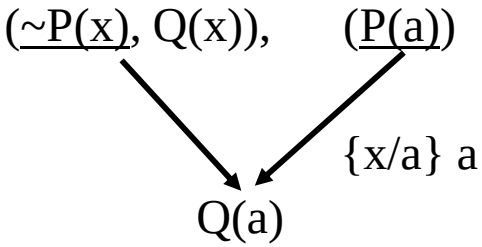
(~person(x) v person(f_sk(x)) ^ (~person(x) v father(f_sk(x), x)) (by step 5)

(~person(x), person(f_sk(x)), (~person(x), father(f_sk(x), x)) (by step 6)

(where f_sk(.) is a skolem function)

Unification of two clauses

- Basic idea: ∀**x P(x) => Q(x), P(a) |-- Q(a)**



{x/a} a substitution in which variable x is bound to a

  - The goal is to find a set of variable bindings so that the argument lists of two opposite literals (in two clauses) can be made the same.
  - Only variables can be bound to other things.
    - a and b cannot be unified (different constants in general refer to different objects)
    - a and f(x) cannot be unified (unless the inverse function of f is known, which is not the case for general functions in FOL)
    - f(x) and g(y) cannot be unified (function symbols f and g in general refer to different functions and their exact definitions are different in different interpretations)

- Cannot bind variable x to y if x appears anywhere in y
  - Try to unify x and f(x). If we bind x to f(x) and apply the binding to both x and f(x), we get f(x) and f(f(x)) which are still not the same (and will never be made the same no matter how many times the binding is applied)
- Otherwise, bind variable x to y, written as x/y (this guarantees to find the most general unifier, or **mgu**)
  - Suppose both x and y are variables, then they can be made the same by binding both of them to any constant c or any function f(.). Such bindings are less general and impose unnecessary restriction on x and y.
- To unify two terms of the same function symbol, unify their argument lists (**unification is recursive**)

  Ex: to unify f(x) and f(g(b)), we need to unify x and g(b)

– When the argument lists contain multiple terms, unify each pair of terms

  Ex. To unify (x, f(x), ...) (a, y, ...)

  1. unify x and a ($\theta = \{x/a\}$)
  2. apply θ to the remaining terms in both lists, resulting (f(a), ...) and (y, ...)
  1. unify f(a) and y with binding y/f(a)
  2. apply the new binding y/f(a) to θ
  3. add y/f(a) to new θ

Unification Examples

- parents(x, father(x), mother(Bill)) and parents(Bill, father(Bill), y)
  - unify x and Bill: θ = {x/Bill}
  - unify father(Bill) and father(Bill): θ = {x/Bill}
  - unify mother(Bill) and y: θ = {x/Bill}, /mother(Bill)}
- parents(x, father(x), mother(Bill)) and parents(Bill, father(y), z)
  - unify x and Bill: θ = {x/Bill}
  - unify father(Bill) and father(y): θ = {x/Bill, y/Bill}
  - unify mother(Bill) and z: θ = {x/Bill, y/Bill, z/mother(Bill)}
- parents(x, father(x), mother(Jane)) and parents(Bill, father(y), mother(y))
  - unify x and Bill: θ = {x/Bill}
  - unify father(Bill) and father(y): θ = {x/Bill, y/Bill}
  - unify mother(Jane) and mother(Bill): Failure because Jane and Bill are different constants

More Unification Examples

- P(x, g(x), h(b)) and P(f(u, a), v, u))
  - unify x and f(u, a): θ = {x/ f(u, a)};
    remaining lists: (g(f(u, a)), h(b)) and (v, u)
  - unify g(f(u, a)) and v: θ = {x/f(u, a), v/g(f(u, a))};
    remaining lists: (h(b)) and (u)
  - unify h(b) and u: θ = {x/f(h(b), a), v/g(f(h(b), a)), u/h(b)};
- P(f(x, a), g(x, b)) and P(y, g(y, b))
  - unify f(x, a) and y: θ = {y/f(x, a)}
    remaining lists: (g(x, b)) and (g(f(x, a), b))
  - unify x and f(x, a): failure because x is in f(x, a)

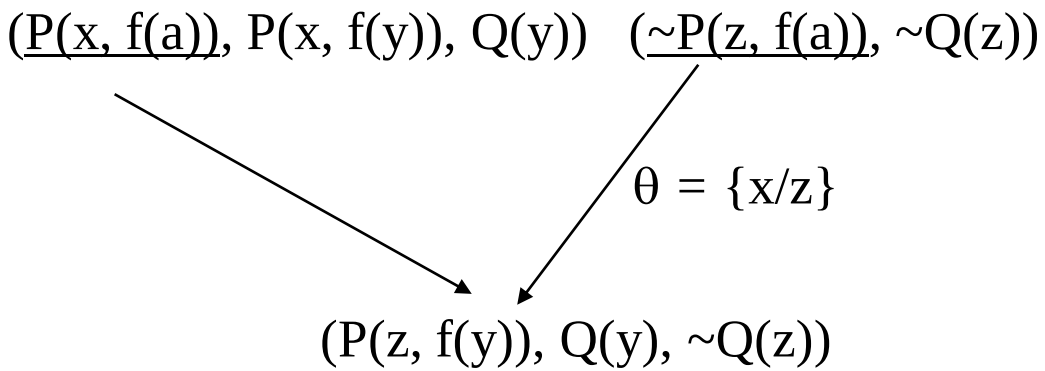
Unification Algorithm (pp. 302-303, Chapter 10)

```
procedure unify(p, q, θ)        /* p and q are two lists of terms and |p| = |q| */
   if p = empty then return θ;  /* success */
   let r = first(p) and s = first(q);
   if r = s then return unify(rest(p), rest(q), θ);
   if r is a variable then temp = unify-var(r, s);
   else if s is a variable then temp = unify-var(s, r);
        else if both r and s are functions of the same function name then
             temp = unify(arglist(r), arglist(s), empty);
             else return "failure";
   if temp = "failure" then return "failure";  /* p and q are not unifiable */
   else θ = subst(θ, temp) ∪ temp;   /* apply tmp to old θ then insert it into θ */
          return unify(subst(rest(p), tmp), subst(rest(q), tmp), θ);
end{unify}
procedure unify-var(x, y)
   if x appears anywhere in y then return "failure";
   else return (x/y)
end{unify-var}
```

Resolution in FOL

- Convert all sentences in KB (axioms, definitions, and known facts) and the goal sentence (the theorem to be proved) to clause form
- Two clauses C1 and C2 can be resolved if and only if **r** in C1 and **s** in C2 are two opposite literals, and their argument list arglist_r and arglist_s are unifiable with mgu = θ.
- Then derive the resolvent sentence: subst((C1 – {r}, C2 – {s}), θ)

  (substitution is applied to all literals in C1 and C2, but not to any other clauses)
- Example

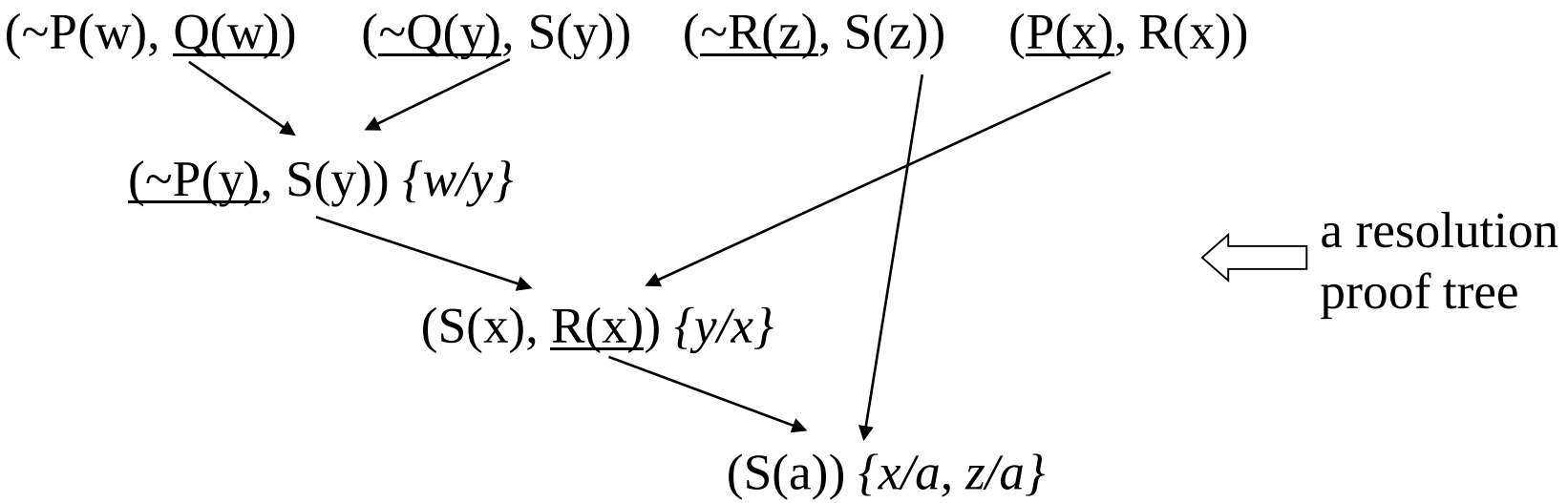


Resolution example

- Prove that

$\forall w\ P(w) \Rightarrow Q(w),\ \forall y\ Q(y) \Rightarrow S(y),\ \forall z\ R(z) \Rightarrow S(z),\ \forall x\ P(x) \vee R(x) \models \exists u\ S(u)$

- Convert these sentences to clauses ($\exists u\ S(u)$ skolemized to S(a))
- Apply resolution

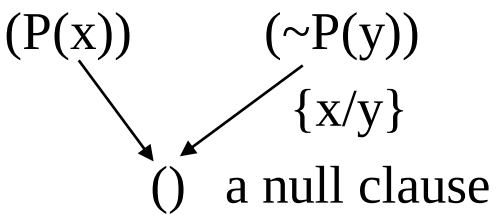


- Problems
  - The theorem S(a) does not actively participate in the proof
  - Hard to determine if a proof (with consistent variable bindings) is completed if the theorem consists of more than one clause

Resolution Refutation: a better proof strategy

- Given a consistent set of axioms KB and goal sentence Q, show that KB |= Q.
- **Proof by contradiction**: Add ~Q to KB and try to prove false.

  because (KB |= Q) <=> (KB ^ ~Q |= False, or KB ^ ~Q is inconsistent)
- How to represent "**false**" in clause form
  - P(x) ^ ~P(y) is inconsistent
  - Convert them to clause form then apply resolution

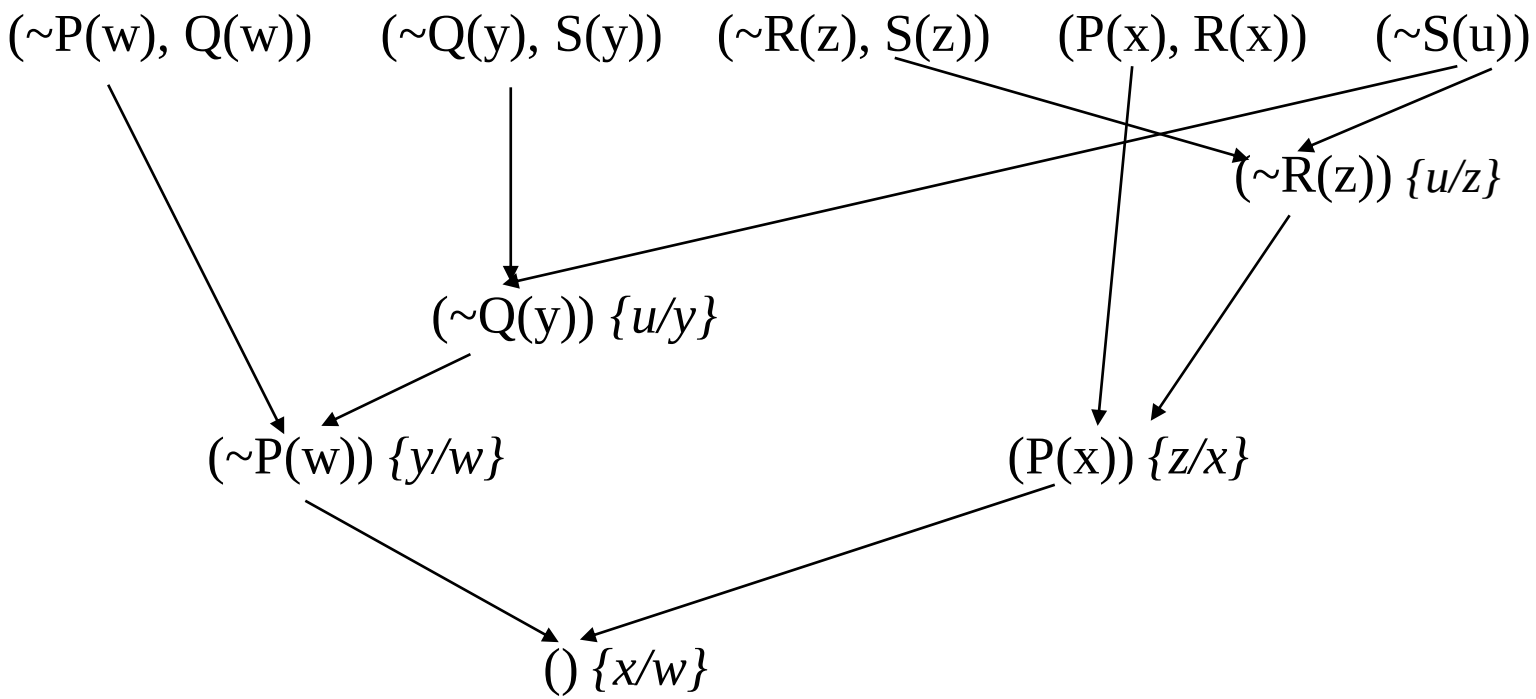


  - A null clause represents false (inconsistence/contradiction)
  - KB |= Q if we can derive a null clause from KB ^ ~Q by resolution

- Prove by resolution refutation that

$\forall w\ P(w) \Rightarrow Q(w),\ \forall y\ Q(y) \Rightarrow S(y),\ \forall z\ R(z) \Rightarrow S(z),\ \forall x\ P(x) \vee R(x) \models \exists u\ S(u)$

- Convert these sentences to clauses (~ ∃u S(u) becomes ~S(u))



Refutation Resolution Procedure

```
procedure resolution(KB, Q)
   /* KB is a set of consistent, true FOL sentences, Q is a goal sentence.
       It  returns success if KB |-- Q, and failure otherwise */
   KB = clause(union(KB, {~Q}))   /* convert KB and ~Q to clause form */
   while null clause is not in KB do
        pick 2 sentences, S1 and S2, in KB that contain a pair of opposite
            literals whose argument lists are unifiable
        if none can be found then return "failure"
        resolvent = resolution-rule(S1, S2)
        KB = union(KB, {resolvent})
   return "success "
end{resolution}
```

Control Strategies

- At any given time, there are multiple pairs of clauses that are resolvable. Therefore, we need a systematic way to select one such pair at each step of proof
  - May lead to a null clause
  - Without losing potentially good threads (of inference)
- There are a number of general (domain independent) strategies that are useful in controlling a resolution theorem prover.
- We'll briefly look at the following
  - Breadth first
  - Set of support
  - Unit resolution
  - Input Resolution
  - Ordered resolution
  - Subsumption

Breadth first

- Level 0 clauses are those from the original KB and the negation of the goal.
- Level k clauses are the resolvents computed from two clauses, one of which must be from level k-1 and the other from any earlier level.
- Compute all level 1 clauses possible, then all possible level 2 clauses, etc.
- Complete, but very inefficient.

Set of Support

- At least one parent clause must be from the negation of the goal or one of the "descendents" of such a goal clause (i.e., derived from a goal clause).
- Complete (assuming all possible set-of-support clauses are derived)
- Gives a goal directed character to the search

Unit Resolution

- At least one parent clause must be a "unit clause," i.e., a clause containing a single literal.
- Not complete in general, but complete for Horn clause KBs

Input Resolution

- At least one parent from the set of original clauses (from the axioms and the negation of the goal)
- Not complete in general, but complete for Horn clause KBs

Linear Resolution

- Is an extension of Input Resolution
- use P and Q if P is in its initial KB (and query) or P is an ancestor of Q.
- Complete.

Ordered Resolution

- Do them in order (Left to right)
- This is how Prolog operates
- Do the first element in the sentence first.
- This forces the user to define what is important in generating the "code."
- The way the sentences are written controls the resolution.

Subsumption

- Eliminate all clauses that are subsumed (more specific than) by an existing clause to keep the KB small.
- Like factoring, this is just removing things that merely clutter up the space and will not affect the final result.
- I.e. if P(x) is already in the KB, adding P(A) makes no sense -- P(x) is a superset of P(A).
- Likewise adding P(A) v Q(B) would add nothing to the KB either.

Example of Automatic Theorem Proof:
Did Curiosity kill the cat

- Jack owns a dog. Every dog owner is an animal lover. No animal lover kills an animal. Either Jack or Curiosity killed the cat, who is named Tuna. Did Curiosity kill the cat?
- These can be represented as follows:

  A. (∃x) Dog(x) ^ Owns(Jack,x)

  B. (∀x) ((∃y) Dog(y) ^ Owns(x, y)) => AnimalLover(x)

  C. (∀x) AnimalLover(x) => (∀y) Animal(y) => ~Kills(x,y)

  D. Kills(Jack,Tuna) v Kills(Curiosity,Tuna)

  E. Cat(Tuna)

  F. (∀x) Cat(x) => Animal(x)

  Q. Kills(Curiosity, Tuna)

- **Convert to clause form**

  A1. (Dog(D)) /* D is a skolem constant */

  A2. (Owns(Jack,D))

  B. (~Dog(y), ~Owns(x, y), AnimalLover(x))

  C. (~AnimalLover(x), ~Animal(y), ~Kills(x,y))

  D. (Kills(Jack,Tuna), Kills(Curiosity,Tuna))

  E. Cat(Tuna)

  F. (~Cat(x), Animal(x))

- **Add the negation of query:**

  Q: (~Kills(Curiosity, Tuna))

- **The resolution refutation proof**

  R1: Q, D, {}, (Kills(Jack, Tuna))

  R2: R1, C, {x/Jack, y/Tuna}, (~AnimalLover(Jack), ~Animal(Tuna))

  R3: R2, B, {x/Jack}, (~Dog(y), ~Owns(Jack, y), ~Animal(Tuna))

  R4: R3, A1, {y/D}, (~Owns(Jack, D), ~Animal(Tuna))

  R5: R4, A2, {}, (~Animal(Tuna))

  R6: R5, F, {x/Tuna}, (~Cat(Tuna))

  R7: R6, E, {} ()

Horn Clauses

- A Horn clause is a clause with at most one positive literal:

  (~P1(x), ~P2(x), ..., ~Pn(x) v Q(x)), equivalent to

  ∀x P1(x) ^ P2(x) ... ^ Pn(x) => Q(x) or

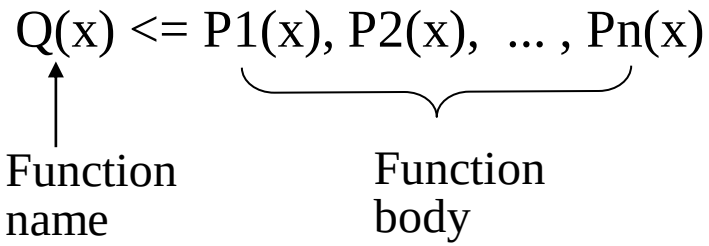
  Q(x) <= P1(x), P2(x), ... , Pn(x) (in prolog format)

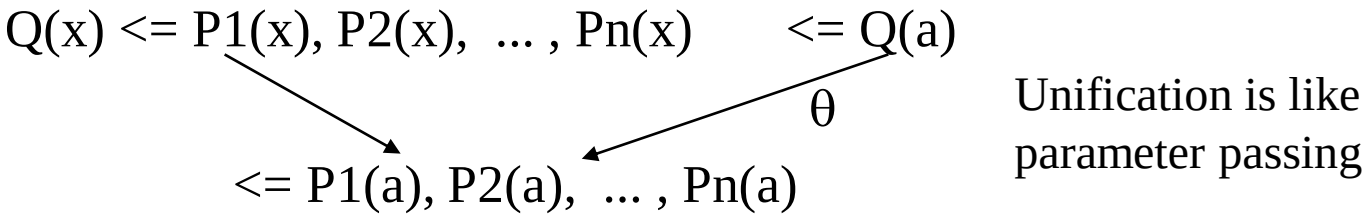
  - if contains no negated literals (i.e., Q(a) <=): facts
  - if contains no positive literals (<= P1(x), P2(x), ... , Pn(x)): query
  - if contain no literal at all (<=): null clause
- Most knowledge can be represented by Horn clauses
- Easier to understand (keeps the implication form)
- Easier to process than FOL
- Horn clauses represent a subset of the set of sentences representable in FOL (e.g., it cannot represent uncertain conclusions, e.g.,

  Q(x) v R(x) <= P(x)).

Logic Programming

- Resolution with Horn clause is like a function all:





To solve Q(a), we solve P1(a), P2(a), ... , and Pn(a). This is called problem reduction (P1(a), ... Pn(a) are subgoals).

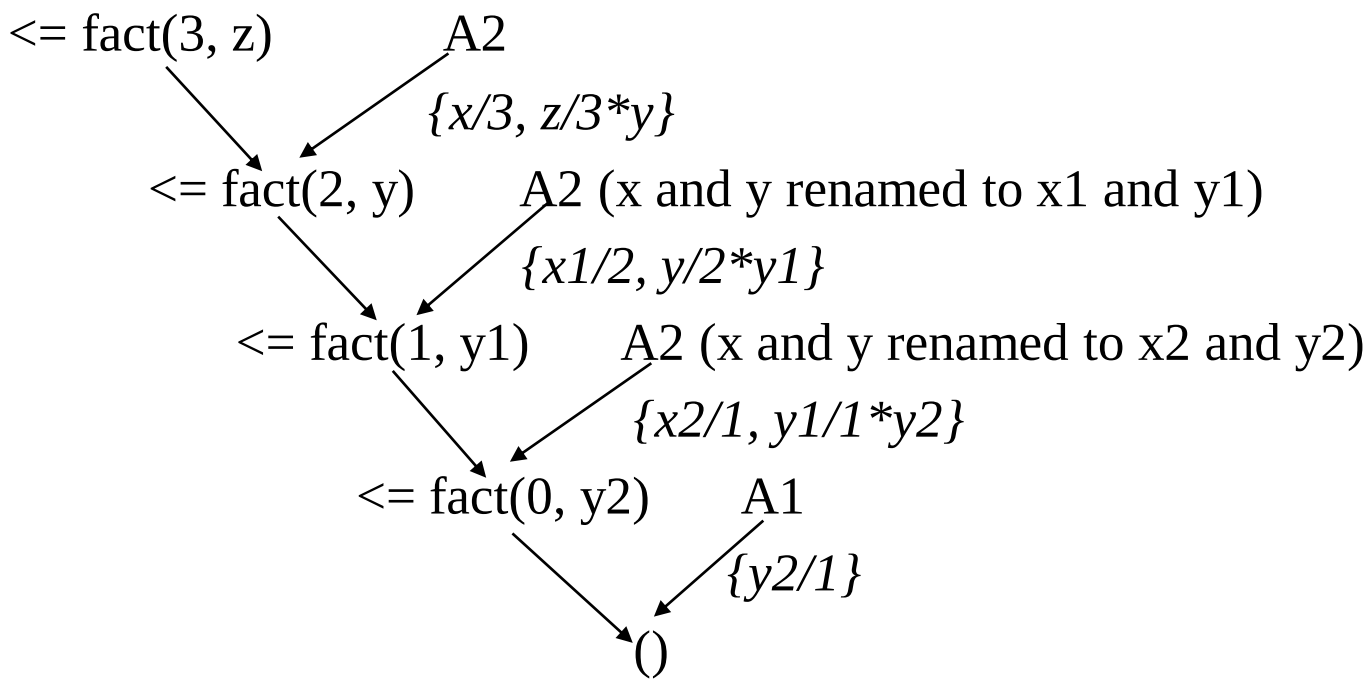
We then continue to call functions to solve P1(a), ..., by resolving <= P1(a), P2(a), ... , Pn(a) with clauses P(y) <= R1(y), ... Rm(y), etc.

Example of Logic Programming

Computing factorials

A1: fact(0, 1) <= /* base case: 0! = 1 */

A2: fact(x, x*y) <= fact(x-1, y) /* recursion: x! = x*(x-1)! */



Extract answer from the variable bindings:

z = 3*y = 3*2*y1 = 3*2*1*y2 = 3*2*1*1 = 6

Prolog

- A logic programming language based on Horn clauses
  - Resolution refutation
  - Control strategy: goal directed and depth-first
    - always start from the goal clause,
    - always use the new resolvant as one of the parent clauses for resolution
    - backtracking when the current thread fails
    - complete for Horn clause KB
  - Support answer extraction (can request single or all answers)
  - Orders the clauses and literals with a clause to resolve non-determinism
    - Q(a) may match both Q(x) <= P(x) and Q(y) <= R(y)
    - A (sub)goal clause may contain more than one literals, i.e., <= P1(a), P2(a)
  - Use “closed world” assumption (negation as failure)
    - If it fails to derive P(a), then assume ~P(a)

Other issues

- FOL is semi-decidable
  - We want to answer the question if KB |= S
  - If actually KB |= S (or KB |= ~S), then a complete proof procedure will terminate with a positive (or negative) answer within finite steps of inference
  - If neither S nor ~S logically follows KB, then there is **no** proof procedure will terminate within **finite** steps of inference for **arbitrary** KB and S.
  - The semi-decidability is caused by
    - infinite domain and incomplete axiom set (knowledge base)
    - Ex: KB contains only one clause fact(x, x*y) <= fact(x-1, y). To prove fact(3, z) will run forever
  - By Godel's Incomplete Theorem, no logical system can be complete (e.g., no matter how many pieces of knowledge you include in KB, there is always a legal sentence S such that neither S nor ~S logically follow KB).
  - Closed world assumption is a practical way to circumvent this problem, but it make the logical system non-monotonic, therefore non-FOL

- Forward chaning
  - Proof starts with the new fact **P(a) <=,** (often case specific data)
  - Resolve it with rules **Q(x) <= P(x)** to derived new fact **Q(a) <=**
  - Additional inference is then triggered by **Q(a) <=**, etc. The process stops when the theorem intended to proof (if there is one) has been generated or no new sentenced can be generated.
  - Implication rules are always used in the way of modus ponens (*from premises to conclusions*), i.e., in the direction of implication arrows
  - This defines a forward chaining inference procedure because it moves "forward" from fact oward the goal (also called data driven).

- Backward chanining
  - Proof starts with the goal query (theorem to be proven) <= **Q(a)**
  - Resolve it with rules **Q(x)** <= **P(x)** to derived new query <= **P(a)**
  - Additional inference is then triggered by <= **P(a)**, etc. The process stops when a null clause is derived.
  - Implication rules are always used in the way of modus tollens (*from conclusions to premises*), i.e., in the reverse direction of implication arrows
  - This defines a backward chaining inference procedure because it moves “backward" from the goal (also called goal driven).
  - Backward chaining is more efficient than forward chaining as it is more focused. However, it requires that the goal (theorem to be proven) be known prior to the inference

Logical Reasoning Systems

Chapter 10

Some material adopted from notes
by Tim Finin,
Andreas Geyer-Schulz
and Chuck Dyer

Introduction

- Real knowledge representation and reasoning systems come in several major varieties.
- They all based on FOL but departing from it in different ways
- These differ in their intended use, degree of formal semantics, expressive power, practical considerations, features, limitations, etc.
- Some major families of reasoning systems are
  - Theorem provers
  - Logic programming languages
  - Rule-based or production systems
  - Semantic networks
  - Frame-based representation languages
  - Databases (deductive, relational, object-oriented, etc.)
  - Constraint reasoning systems
  - Truth maintenance systems
  - Description logics

Production Systems (forward-chaining)

- The notion of a "production system" was invented in 1943 by Post to describe re-write rules for symbol strings
- Used as the basis for many rule-based expert systems
- Most widely used KB formulation in practice
- A production is a rule of the form:

C1, C2, … Cn => A1 A2 …Am

Left hand side (LHS)
Conditions/antecedents
Condition which must hold before the rule can be applied

Right hand side (RHS)
Conclusion/consequence
Actions to be performed or conclusions to be drawn when the rule is applied

Three Basic Components of PS

- **Rule Base**
  - Unordered set of user-defined "if-then" rules.
  - Form of rules: *if P1* ^ ... ^ *Pm then A1, ..., An*
  - the *Pis* are conditions (often facts) that determine when rule is applicable.
  - Actions can add or delete facts from the Working Memory.
  - Example rule (in CLIPS format)

```
(defrule determine-gas-level
    (working-state engine does-not-start)
    (rotation-state engine rotates)
    (maintenance-state engine recent)
     => (assert (repair "Add gas.")))
```

- **Working Memory (WM) --** A set of "facts“, represented as literals, defining what's known to be true about the world
  - Often in the form of “flat tuples” (similar to predicates), e.g., (age Fred 45)
  - WM initially contains case specific data (not those facts that are always true in the world)
  - Inference may add/delete fact from WM
  - WM will be cleared when a case is finished
- **Inference Engine --** Procedure for inferring changes (additions and deletions) to Working Memory.
  - Can be both forward and backward chaining
  - Usually a cycle of three phases: match, conflict resolution, and action, (in that order)

Basic Inference Procedure

While changes are made to Working Memory do:

- **Match** the current WM with the rule-base
  - Construct the Conflict Set -- the set of all possible (*rule*, *facts*) pairs where rule is from the rule-base, facts from WM that unify with the conditional part (i.e., LHS) of the rule.
- **Conflict Resolution**: Instead of trying all applicable rules in the Conflict set, select one from the Conflict Set for execution. (**depth-first**)
- **Act/fire:** Execute the actions associated with the conclusion part of the selected rule, after making variable substitutions determined by unification during match phase
- **Stop** when conflict resolution fails to returns any (rule, facts) pair

Conflict Resolution Strategies

- **Refraction**
  - A rule can only be used once with the same set of facts in WM. This strategy prevents firing a single rule with the same facts over and over again (avoiding loops)
- **Recency**
  - Use rules that match the facts that were added most recently to WM, providing a kind of "focus of attention" strategy.
- **Specificity**
  - Use the most specific rule,
  - If one rule's LHS is a superset of the LHS of a second rule, then the first one is more specific
  - If one rule's LHS implies the LHS of a second rule, then the first one is more specific
- **Explicit priorities**
  - E.g., select rules by their pre-defined order/priority
- **Precedence of strategies**

- **Example**
  - R1: P(x) => Q(x); R2: Q(y) => S(y); WM = {P(a), P(b)}

    conflict set: {(R1, P(a)), (R1, P(b))}

    by rule order: apply R1 on P(a); WM = {*Q(a)*, P(a), P(b)}

    conflict set: {(R2, Q(a)), (R1, P(a)), (R1, P(b))}

    by recency: apply R2 on Q(a) WM = {*S(a)*, Q(a), P(a), P(b)}

    conflict set: {(R2, Q(a)), (R1, P(a)), (R1, P(b))}

    by refraction, apply R1 on P(b): WM = {*Q(b)*, S(a), Q(a), P(a), P(b)}

    conflict set: {(R2, Q(b)), (R2, Q(a)), (R1, P(a)), (R1, P(b))}

    by recency, apply R2 on P(b): WM = {*S(b)*, Q(b), S(a), Q(a), P(a), P(b)}
  - Specificity

    R1: bird(x) => fly(x) WM={bird(tweedy), penguin(tweedy)}

    R2: penguin(z) => bird(z)

    R3: penguin(y) => ~fly(y)

    R3 is more specific than R1 because according to R2, penguin(x) implies bird(x)

Default Reasoning

- Reasoning that draws a plausible inference on the basis of *less than conclusive* evidence in the *absence* of information to the contrary
  - If WM = {bird(tweedy)}, then by default, we can conclude that fly(tweedy)
  - When also know that penguin(tweedy), then we should change the conclusion to ~fly(tweedy)
  - Bird(x) => fly(x) is a default rule (true in general, in most cases, almost)
  - Default reasoning is thus non-monotonic
  - Formal study of default reasons: *default logic* (Reiter), *nonmonotonic logic* (McDermott), *circumscription* (McCarthy)

    one conclusion: default reasoning is totally undecidable
  - Production system can handle simple default reasoning
    - By specificity: default rules are less specific
    - By rule priority: put default rules at the bottom of the rule base
    - Retract default conclusion (e.g., fly(tweedy)) is complicated

Other Issues

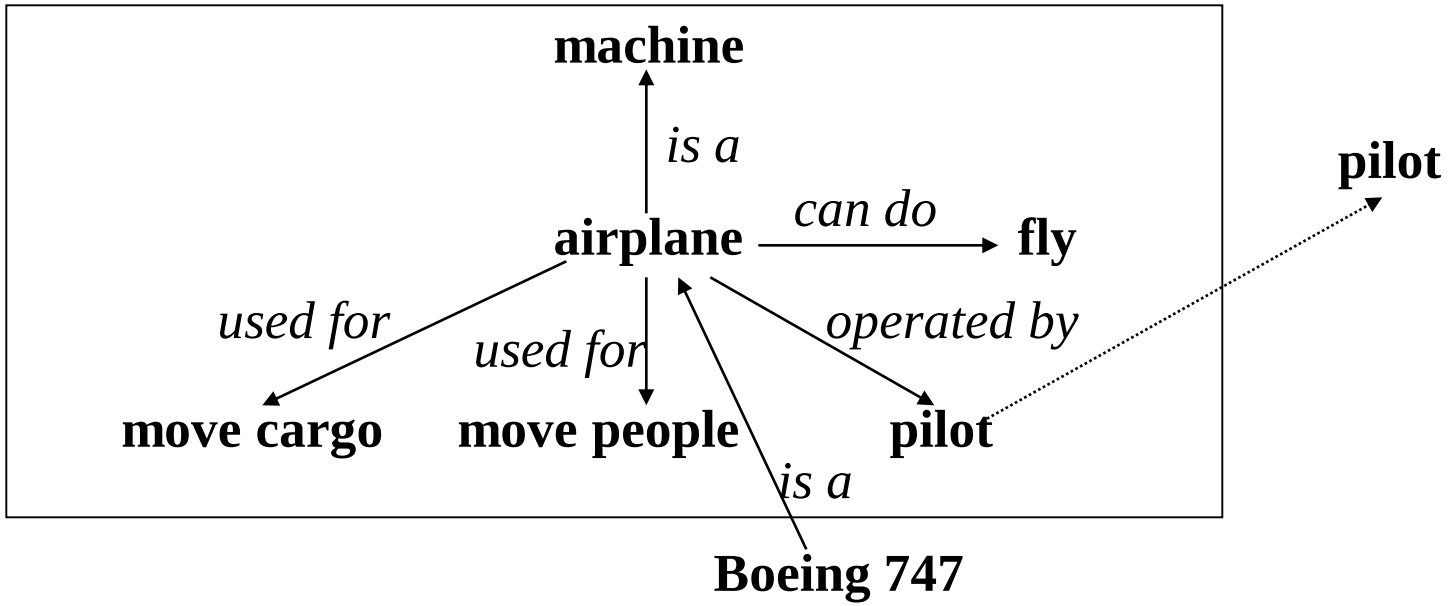
- PS can work in backward chaining mode
  - Match RHS with the goal statement to generate subgoals
  - Mycin: an expert system for diagnosing blood infectious diseases
- Expert system sell
  - A rule-based system with empty rule base
  - Contains data structure, inference procedures, AND user interface to help encode domain knowledge
  - Emycin (backward chaining) from Stanford U
  - OPP5 (forward chaining) from CMU and its descendents CLIPS, Jess.
- Metarules
  - Rules about rules
  - Specify under what conditions a set of rules can or cannot apply
  - For large, complex PS
- Consistency check of the rule-base is crucial (as in FOL)
- Uncertainty in PS (to be discussed later)

Comparing PS and FOL

- Advantages
  - Simplicity (both KR language and inference),
  - Inference more efficient
  - Modularity of knowledge (rules are considered, to a degree, independent of each other), easy to maintain and update
  - Similar to the way humans express their knowledge in many domains
  - Can handle simple default reasoning
- Disadvantages
  - No clearly defined semantics (may derive incorrect conclusions)
  - Inference is not complete (mainly due to the depth-first procedure)
  - Inference is sensitive to rule order, which may have unpredictable side effects
  - Less expressive (may not be suitable to some applications)
- ***No explicit structure among pieces of knowledge in BOTH FOL*** *(a un-ordered set of clauses)* ***and PS*** *(a list of rules)*

Semantic Networks

- Structured representations (semantic networks and frame systems)
  - Put structures into KB (capture the interrelations between pieces of knowledge
  - Center around object/classes
  - More for what it is than what to do
- History of semantics networks (Quillian, 1968)
  - To represent semantics of natural language words by dictionary-like definitions in a graphic form
  - Defining the meaning of a word in terms of its relations with other words
  - Semantic networks were very popular in the 60's and 70's and enjoy a much more limited use today.
  - The **graphical depiction** associated with a semantic network is a big reason for their popularity.



- Nodes for words
- Directed links for relations/associations between words
- Each link has its own meaning
- You know the meaning (semantics) of a word if you know the meaning of all nodes that are used to define the word and the meaning of the associated links
- Otherwise, follow the links to the definitions of related words

Semantic Networks

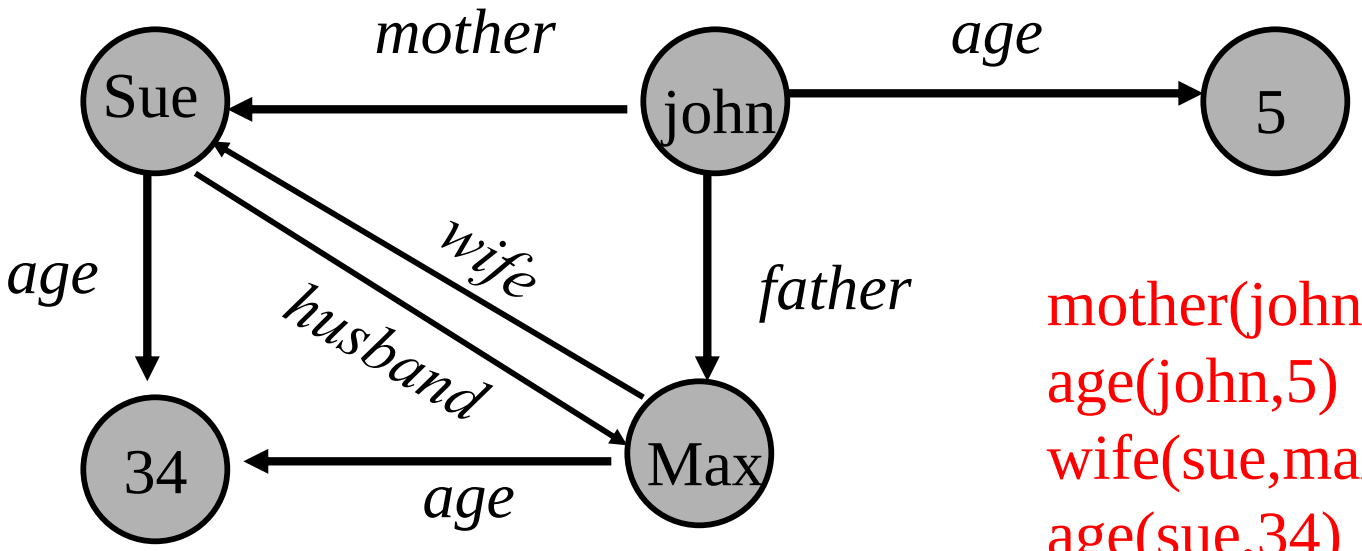
- A semantic (or associative) network is a simple representation scheme which uses a graph of *labeled* nodes and *labeled*, *directed* arcs to encode knowledge.
  - Labeled nodes: objects/classes/concepts.
  - Labeled links: relations/associations between nodes
  - Labels define the semantics of nodes and links
  - Large # of node labels (there are many distinct objects/classes)

    Small # of link labels (types of associations can be merged into a few)

    buy, sale, give, steal, confiscation, etc., can all be represented as a single relation of "*transfer ownership*" between recipient and donor
  - Usually used to represent static, taxonomic, concept dictionaries
- Semantic networks are typically used with a special set of accessing procedures which perform "reasoning"
  - e.g., inheritance of values and relationships
- often much less expressive than other KR formalisms

Nodes and Arcs

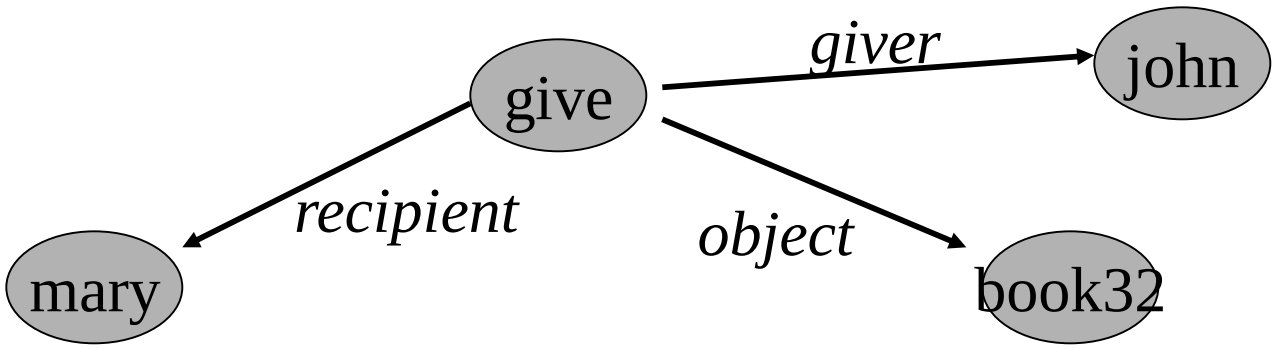
- Nodes denote objects/classes
- arcs define binary relationships between objects.



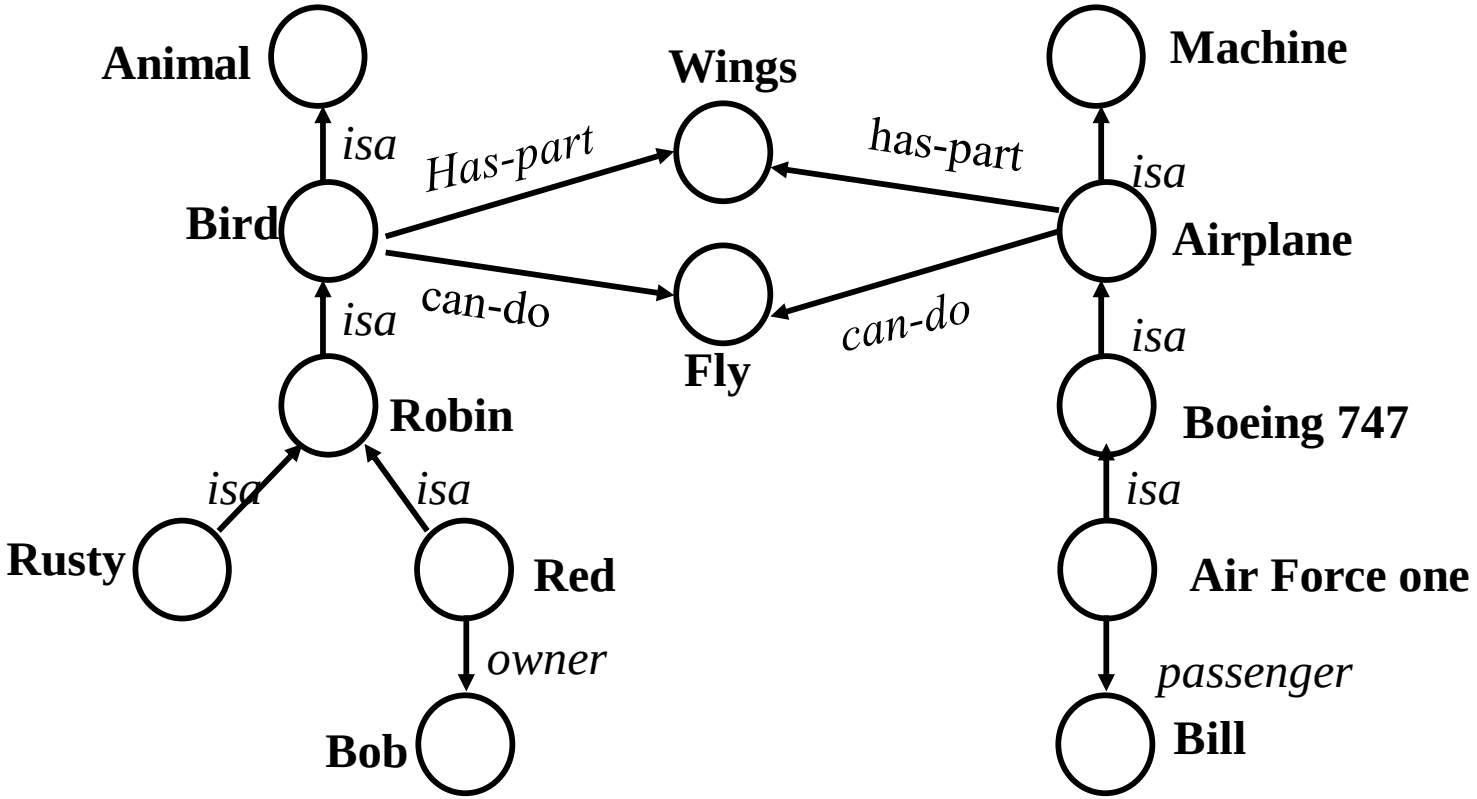
mother(john,sue)
age(john,5)
wife(sue,max)
age(sue,34)
...

Reification

- Non-binary relationships can be represented by “turning the relationship into an object”
- This is an example of what logicians call “reification”
  - reify v : consider an abstract concept to be real
- We might want to represent the generic “give” event as a relation involving three things: a giver, a recipient and an object, give(john, mary, book32)



Inference by association



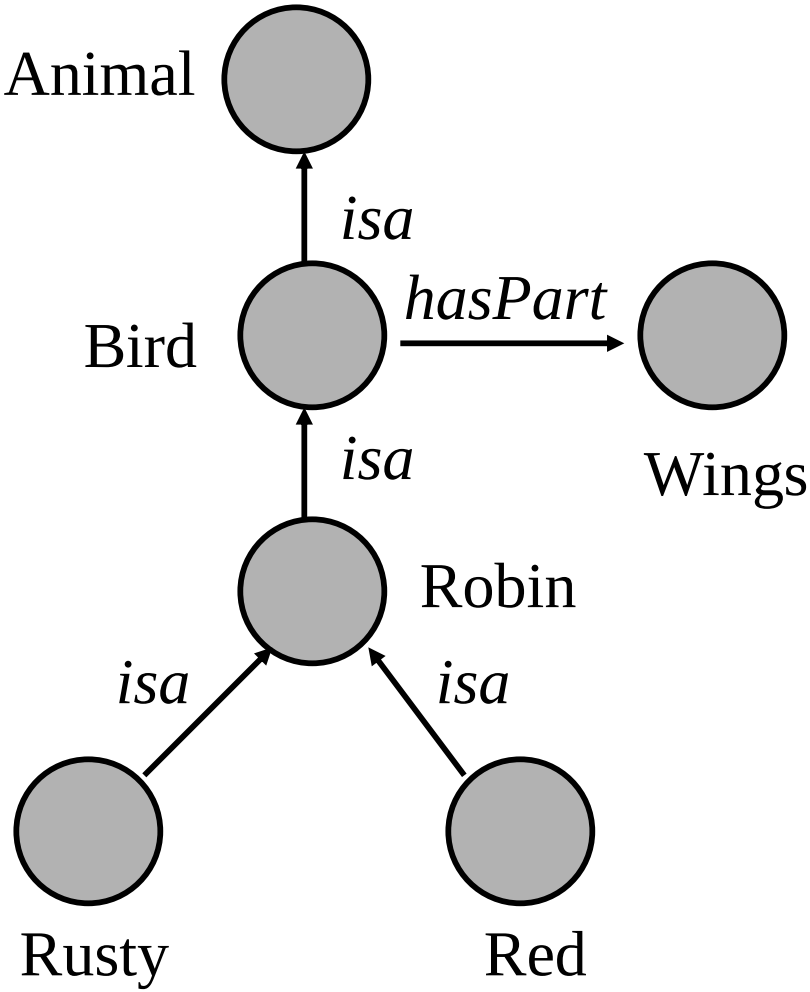
- Red (a robin) is related to Air Force One by association (as directed path originated from these two nodes join at nodes Wings and Fly)
- Bob and Bill are not related (no paths originated from them join in this network

Inferring Associations

- Marker passing
  - Each node has an unique marker
  - When a node is activated (from outside), it sends copies of its marker to all of its neighbors (following its outgoing links)
  - Any nodes receiving a marker sends copies of that marker to its neighbors
  - If two different markers arrive at the same node, then it is concluded that the owners of the two markers are associated
- Spreading activation
  - Instead of passing labeled markers, a node sends labeled activations (a numerical value), divided among its neighbors by some weighting scheme
  - A node usually consumes some amount of activation it receives before passing it to others
  - The amount of activation received by a node is a measure of the strength of its association with the originator of that activation
  - The spreading activation process will die out after certain radius

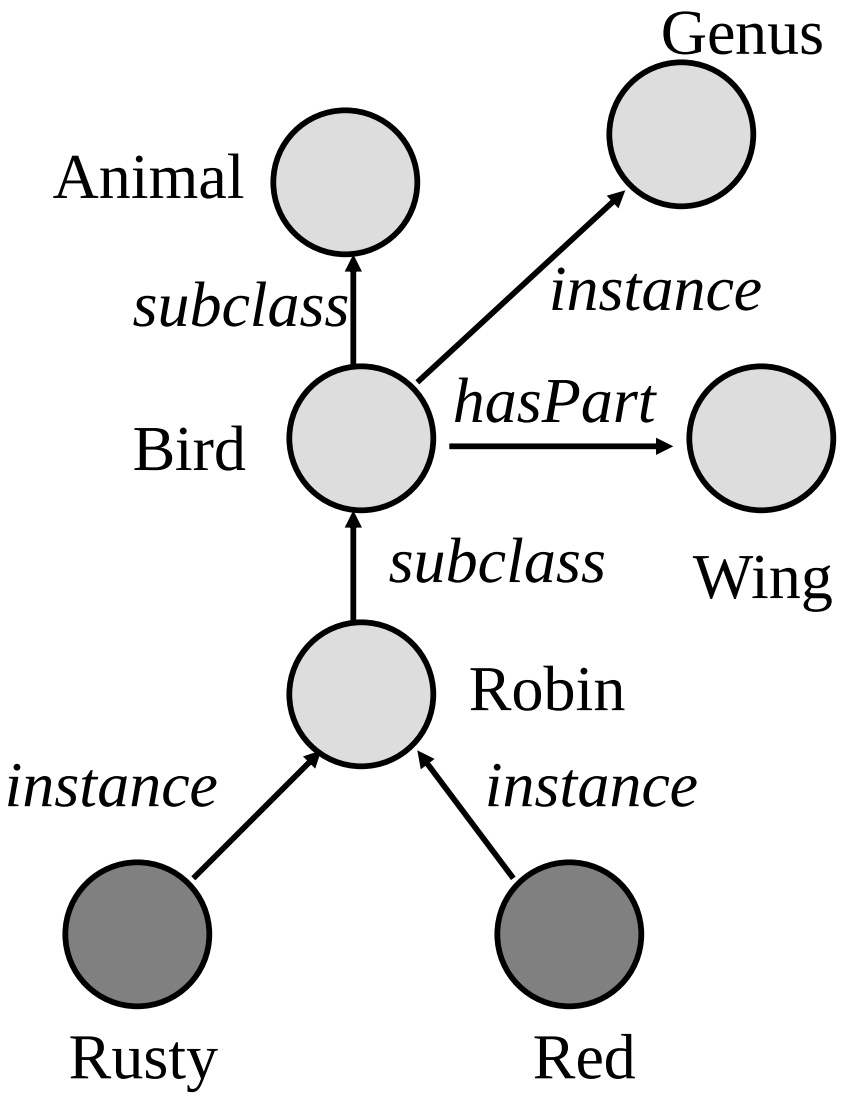
ISA hierarchy

- The ISA (is a) or AKO (a kind of) relation is often used to link a class and its superclass.
- And sometimes an instance and it's class.
- Some links (e.g. has-part) are inherited along ISA paths.
- The semantics of a semantic net can be relatively informal or very formal
  - often defined at the implementation level



Individuals and Classes

- Many semantic networks distinguish
  - nodes representing individuals and those representing classes
  - the “subclass” relation from the “instance-of” relation



Inference by Inheritance

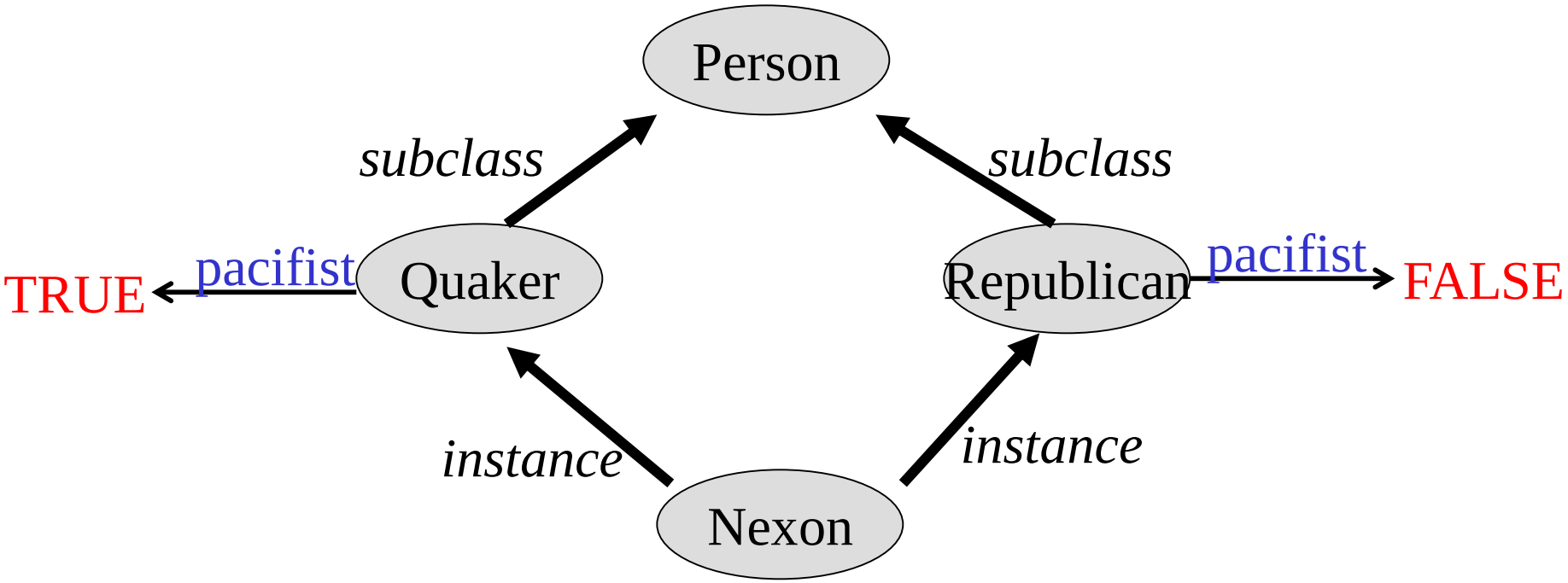
- One of the main types of reasoning done in a semantic net is the inheritance of values (properties) along the subclass and instance links.
- Semantic Networks differ in how they handle the case of inheriting multiple different values.
  - All possible properties are inherited
  - Only the “lowest” value or values are inherited

Multiple inheritance

- A node can have any number of superclasses that contain it, enabling a node to inherit properties from multiple "parent" nodes and their ancestors in the network.
- Conflict or inconsistent properties can be inherited from different ancestors
- Rules are used to determine inheritance in such "tangled" networks where multiple inheritance is allowed:
  - if $X \subseteq A \subseteq B$ and both A and B have property P (possibly with different variable instantiations), then X inherits A's property P instance (closer ancestors override far away ones).
  - If $X \subseteq A$ and $X \subseteq B$ but neither $A \subseteq B$ nor $B \subseteq A$ and both A and B have property P with different and inconsistent values, then X will not inherit property P at all; or X will present both instances of P (from A and B) to the user

Nixon Diamond

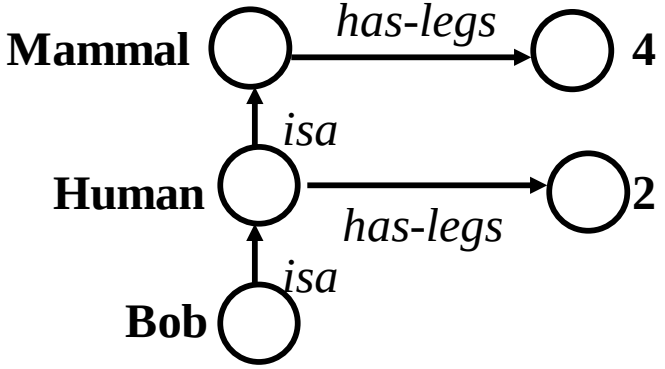
- This was the classic example circa 1980.



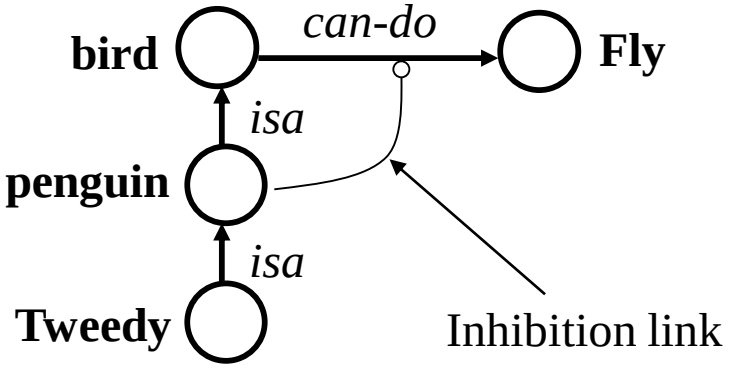
Exceptions in ISA hierarchy

- Properties of a class are often default in nature (there are exceptions to these associations for some subclasses/instances)

- Closer ancestors (more specific) overriding far way ones (more general)



- Use explicit inhibition links to prevent inheriting some properties

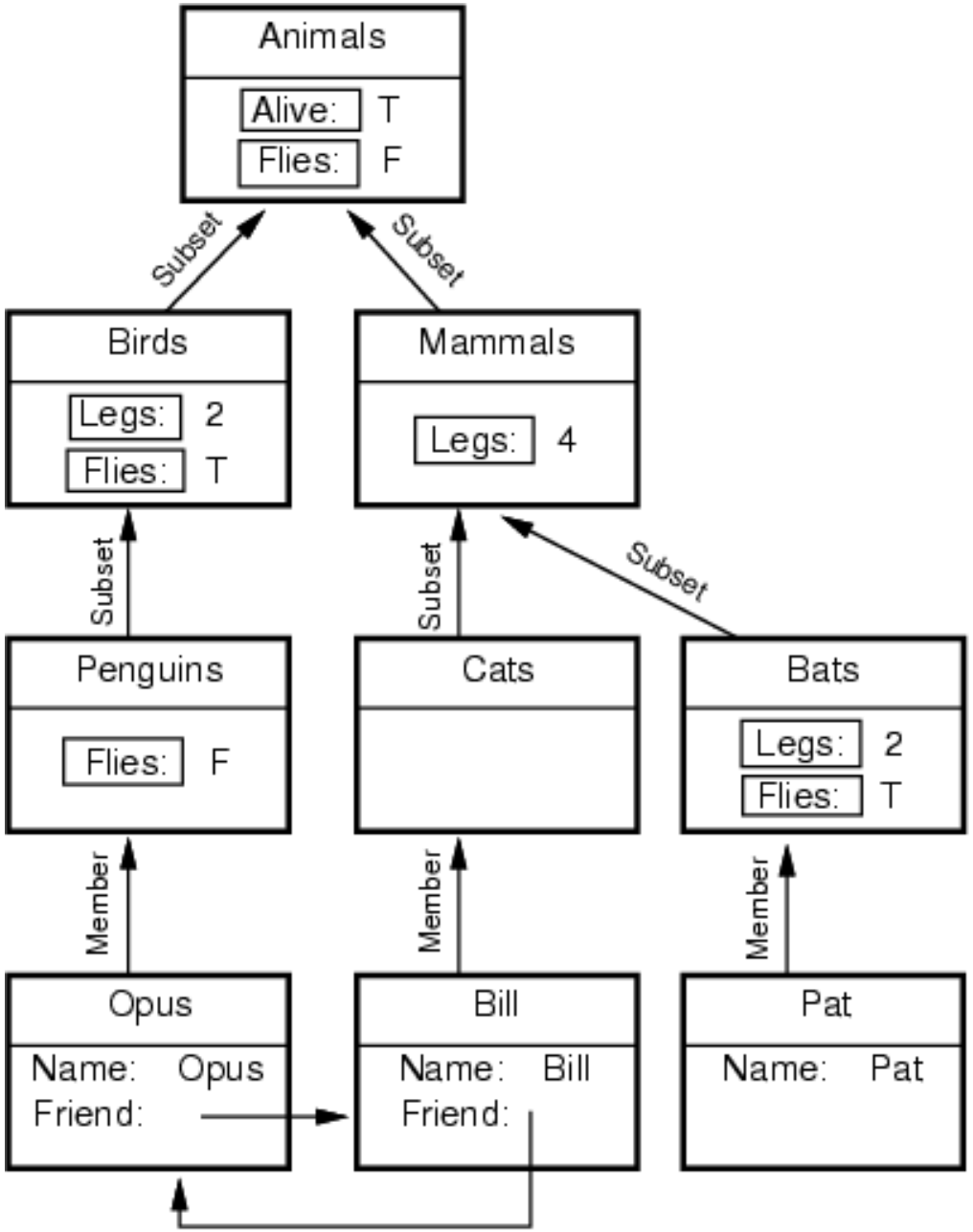


From Semantic Nets to Frames

- Semantic networks morphed into Frame Representation Languages in the 70's and 80's.
- A Frame is a lot like the notion of an object in OOP, but has more meta-data.
- A frame represents a **stereotypical/expected/default** view of an object
- Frame system can be viewed as adding additional structure into semantic network, a frame includes the object node and all other nodes which directly related to that object, organized in a **record like** structure
- A frame has a set of **slots,** each represents a relation to another frame (or value).
- A slot has one or more **facets,** each represents some aspect of the relation

Facets

- A slot in a frame holds more than a value.
- Other facets might include:
  - current fillers (e.g., values)
  - default fillers
  - minimum and maximum number of fillers
  - type restriction on fillers (usually expressed as another frame object)
  - attached procedures (if-needed, if-added, if-removed)
  - salience measure
  - attached constraints or axioms
  - pointer or name of another frame



(a) A frame–based knowledge base

Rel(Alive,Animals,T)
Rel(Flies,Animals,F)

Birds $\subset$ Animals
Mammals $\subset$ Animals

Rel(Flies,Birds,T)
Rel(Legs,Birds,2)
Rel(Legs,Mammals,4)

Penguins $\subset$ Birds
Cats $\subset$ Mammals
Bats $\subset$ Mammals

Rel(Flies,Penguins,F)
Rel(Legs,Bats,2)
Rel(Flies,Bats,T)

Opus $\in$ Penguins
Bill $\in$ Cats
Pat $\in$ Bats

Name(Opus,"Opus")
Name(Bill,"Bill")
Friend(Opus,Bill)
Friend(Bill,Opus)
Name(Pat,"Pat")

(b) Translation into first–order logic

Other issues

- **Procedural attachment**
  - In early time, AI community was against procedural approach and stress declarative KR
  - Procedures came back to KB systems when frame systems were developed, and later also adopted by some production systems (action can be a call to a procedure)
  - It is not called by a central control, but triggered by activities in the frame system
  - When an attached procedure can be triggered

    if-added: when a new value is added to one of the slot in the frame

    if-needed: when the value of this slot is needed

    if-updated: when value(s) that are parameters of this procedure is changed

- **Example:** a real estate frame system
  - Slots in a real estate property frame

    location

    area

    price
  - A facet in “price” slot is a procedure that finds the unit price (by location) and computes the prince value as the product of the unit price and the area
  - If the procedure is the type of *if-needed*, it then will be triggered by a request for the price from other frame (i.e., transaction frame)
  - If it is the type of *if-updated*, it then will be triggered by any change in either location or area
  - If it is the type of *if-added*, it then will be triggered by the first time when both location and area values are added into this frame

- **Description logic**
  - There is a family of Frame-like KR systems with a formal semantics.
    - E.g., KL-ONE, LOOM, Classic, …
  - An additional kind of inference done by these systems is automatic **classification**
    - finding the right place in a hierarchy of objects for a new description
  - Current systems take care to keep the language simple, so that all inference can be done in polynomial time (in the number of objects)
    - ensuring tractability of inference

- **Objects with multiple perspectives**
  - An object or a class may be associated with different sets of properties when viewed from different perspectives.
  - A passenger in an airline reservation system can be viewed as
    - a *traveler*, whose frame should include slots such as the
      date of the travel,
      departure/arrive airport;
      departure/arrive time, ect.
    - A *customer*, whose frame should include slots such as
      fare amount
      credit card number and expiration date
      frequent flier's id, etc.
  - Both traveler frame and customer frame should be children of the *passenger* frame, which has slots for properties not specific to each perspective. They may include name, age, address, phone number, etc. of that person

Midterm Review (CMSC 471/671, Fall 2000)

- **State Space**
  - States: initial, goal.
  - State transition rules/operators/actions and costs associate with operations
  - State space as directed graph (nodes, arcs, parents/children)
  - Node generation and node expansion: open/closed nodes, open/closed lists.
  - Solution, solution path and its cost.
  - Be able to represent simple problem-solving as state space search

- **Uninformed (blind) Search Methods**
  - Breadth-first.
  - Depth-first, Depth-limited (plus back-tracking).
  - IDDF: Iterative-deepening depth-first. (motivation, advantage over BF and DF methods.)
  - Uniform-cost search.
  - Bi-directional search
  - Algorithm, time and space complexities, optimality and completeness of each of these search methods

- **Informed Search methods**
  - Evaluation function f(n),
  - Heuristic estimate function h(n)
    - what does h(n) estimate
    - admissible h(n), null h(n), perfect h*(n), more informed h(n)
    - idea of automatic generation of h functions
  - Algorithm A and A*
    - f(n) = g(n) + h(n): what does each of the terms stand for;
    - algorithm (maintaining open/closed lists, delayed termination test; node expansion and generation, handling duplicate nodes, back pointers);
    - difference between algorithms A and A*
    - time and space complexity, completeness and optimality of A*
    - be able to apply A* to simple problems.
  - Ways to improving A* search
    - IDA* (basic idea; how to set f_limit at each iteration; advantages over A*)
    - Dynamic weighting (an algorithm)
    - Pruning open list by f+ (where f+ is the cost of any known solution)
  - Best-first search
  - Greedy search and hill-climbing (algorithms; time and space complexity, completeness and optimality)
  - Basic ideas of other incremental improving search methods
  - Simulated annealing; Genetic algorithm

- Game-Tree Search
  - Perfect 2-player games
  - Game tree (Max and Min nodes; terminal and leave nodes)

- What to search for (one move for Max)
- Heuristic evaluation function f(n) (merit of a board)
- Minimax rule for game tree search
- Idea of alpha-beta pruning, its time and space complexities.
- Difference between general state space search and game tree search
- Be able to apply Minimax rule and alpha-beta pruning to simple problems.

- Propositional Logic (PL)
  - Syntax
    - Propositions
    - Symbols (T, F, proposition symbols)
    - Connectives
    - Definition of PL sentences
  - Semantics
    - Interpretation (an assignment of truth values to all prepositional symbols); models
    - Truth tables for logical connectives
    - Valid (tautology), satisfiable and inconsistent (contradiction) sentences
    - Logical consequence or theorem (S |= X)
  - Equivalence laws
    - P Q iff they have the same truth tables
    - P => Q ~P v Q; distribution /associative/communicative laws, De Morgan's laws
  - Deductive rules
    - Derivation using inference rules: S |-- X
    - Modus Ponens, Modus Tollens, Chaining, And Introduction, And Elimination, etc.
    - Resolution rule (and CNF)
  - Deductive inference
    - Using truth table (S |= X iff S => X is valid)
    - Proof procedure (using inference rules)
    - Sound inference rules and proof procedures (if S |-- X then S |= X)
    - Complete proof procedures (if S |= X then S |-- X). (exponential time complexity)
  - Treating PL inference as state space search

- First Order Logic (FOL)
  - Syntax
    - Terms (constants, variables, functions of terms)
    - Predicates (special functions, ground predicates), atoms and literals
    - Logical connectives
    - Quantifiers (universal and existential), their scopes, De Morgan's law with quantifiers
    - Definitions of FOL sentences and well-formed formulas (wffs)
  - Semantics
    - Interpretation (constants, functions, and predicates) and models
    - Semantics of logical connectives and quantifiers
    - Valid, satisfiable, and inconsistent sentence(s)
    - Logical consequences
    - Be able to translate between English sentences and FOL sentences

Review 2: Logic-Based Inference

- **Deductive Inference in FOPL** (Chapter 9, Sections 10.3, 10.4)
  - Convert first order sentences to clause form
    - Definition of clauses
    - Conversion procedure
      step 1: Eliminate implication and equivalence symbols
      step 2: Move all negation symbols to individual predicates
      step 3: Eliminate all existential quantifiers (Skolemization)
      step 4: Eliminate all universally quantifiers
      step 5: Convert the sentence to conjunctive normal form
      step 6: use parenthesis to separate all disjunctions, then drop all v's and ^'s
  - Unification (obtain mgu θ)
    - Two terms x and y can be unified only if one of them, say x, is a variable and x does not appear anywhere in y. Then x/y is added into the substitution θ.
    - When one binding variable/term is found, apply it to the remainders of both argument lists and to previously found bindings before proceeding to unify other arguments
    - Two argument lists of the same predicate are unifiable if every corresponding pair of terms, one from each list, is unifiable
  - Resolution .
    - Two clauses C1 and C2 can be resolved if one contain literal P and the other contains ~P and the argument lists of P and ~P can be unified with mgu θ
    - The resulting clause (resolvant) is composed of all literals of C1 and C2 except P and ~P, subject to variable substitution according to θ.
  - Resolution Refutation
    - Write the axioms as FOL sentences and convert them into clause form
    - Write the goal (theorem) as a FOL sentence
    - Negate the goal and convert it to clause form
    - Select a pair of clauses for resolution which are
      i) resolvable, and ii) promising toward deriving a null clause,
    - Inference stops when a null clause is derived
  - Control strategies
    - Depth-first: incomplete
    - Breadth-first: complete
    - Unit resolution (one of the two parent clauses is a singleton clause): incomplete in general but complete for Horn clauses
    - Input resolution (one of the two parent clauses is from the original set of clauses): incomplete in general but complete for Horn clauses
    - Set of support: complete
    - Linear resolution (one of the two parent clauses is either from the original set of clauses or is an ancestor of the other parent clause): complete
  - Horn clauses and logic programming
    - Definition of Horn clauses
    - Advantages and limitations of using Horn clauses
    - Logic programming as a general purpose programming language (viewing and resolution as function calls answer extraction)
    - Features of Prolog (Horn clauses, top-down/left-right, depth-first plus backtracking)

- Advantages
  - Clearly defined semantics (least ambiguous
  - Expressiveness
  - Natural for some domains
- Disadvantages
  - Semantics is too rigid
  - Inefficiency (inference is NP}-hard with complete control strategies; semi-decidable)
  - Unnatural for many domains

- **Production (Rule-Based) Systems**] (Section 10.5)
  - System components: WM, rule base, inference engine (rule interpreter)
  - Inference procedure
    - Cycle of three phases: match, conflict-resolution, act/fire
    - Forward and backward inference
  - Conflict resolution
    - conflict set
    - conflict resolution policies (refraction, specificity, recency, priority/rule-ordering)
  - Advantages
    - Simplicity (for both language and inference)
    - Efficiency
    - Modularity (easy for KB maintenance)
    - Natural for many application domains
  - Disadvantages
    - No clearly defined semantics (based on informal understanding)
    - Incomplete inference procedure
    - Unpredictable side effects of ordering of rule applications
    - Less expressive (may not be suitable for some applications)

- **Structured representation**
  - Semantics (associative) networks
    - labeled nodes: objects, classes, concepts
    - Labeled directed links: relations (associations) between nodes
    - Reasoning about associations (marker passing and spreading activation)
  - ISA hierarchy and property inheritance
    - Super/subclass and instance/class relation
    - Inference by inheritance
    - Multiple inheritance (from different parents, from ancestors of different distances)
    - Exceptions in inheritance
  - Frame Systems
    - Definition (stereotypical views of the world; record like structure)
    - Slots, their values and facets
    - Procedural attachment and how they work (if-added, if-needed, if-updated)
    - Frames from different perspectives

- **Default reasoning**
  - Definition (inference is drawn in the absence of info to the contrary) and examples

- Default reasoning is non-monotonic, and it totally undecidable
- How production systems and semantic networks (and frame systems) handle simple default reasoning

Review 3: Abduction, Uncertainty, and Probabilistic Reasoning

- **Abduction**
  - Definition
  - Difference between abduction, deduction, and induction
  - Characteristics of abductive inference
    - Inference results are hypotheses, not theorems (may be false)
    - There may be multiple plausible hypotheses
    - Reasoning is often a hypothesize-and-test cycle
    - Reasoning is non-monotonic
    - Sources of uncertainty (uncertain data, knowledge, and inference)

- **Simple Bayesian approach to evidential/diagnostic reasoning**
  - Bayes' theorem
  - Conditional independence (and evidence) and single fault assumptions
  - Methods for computing posterior probability and relative likelihood of a hypothesis, given some evidence

$$P(H_i \mid E_1,...,E_l) = \frac{P(E_1,...E_l \mid H_i)P(H_i)}{P(E_1,...E_l)}$$

$$rel(H_i \mid E_1,...,E_l) = P(E_1,...,E_l \mid H_i)P(H_i) = P(H_i)\prod_{j=1}^{l} P(E_j \mid H_i)$$

$$P(H_i \mid E_1,...,E_l) = \frac{rel(H_i \mid E_1,...,E_l)}{\sum_{k=1}^{n} rel(H_k \mid E_1,...,E_l)} = \frac{P(H_i)\prod_{j=1}^{l} P(E_j \mid H_i)}{\sum_{k=1}^{n} P(H_k)\prod_{j=1}^{l} P(E_j \mid H_k)}$$

  - Evidence accumulation

$$rel(H_i \mid E_1,...,E_l, \sim E_{l+1}) = (1 - P(E_{l+1} \mid H_i))rel(H_i \mid E_1,...,E_l)$$

$$rel(H_i \mid E_1,...,E_l, E_{l+1}) = P(E_{l+1} \mid H_i)rel(H_i \mid E_1,...,E_l)$$

  - Limitation
    - Assumptions unreasonable for many problems
    - Not suitable for multi-fault problems
    - Can not represent causal chaining

- **Bayesian belief networks (BBN)**
  - Integration of probability theory and causal networks
  - Definition of BBN (DAG and CPD).

    $P(x_i \mid \pi_i)$ where π_i is the set of all parent nodes of x_i
  - Independence assumption

    $P(x_i \mid \pi_i, q) = P(x_i \mid \pi_i)$
  - Computing joint probability distribution

    $P(x_1,..., x_n) = \Pi_{i=1}^{n} P(x_i \mid \pi_i)$

- Inference (e.g., belief update, MAP) is NP}-complete (exponential time)
- BBN of noise-or gate (advantages and limitations)
- Learning BBN from case data (difficulty in learning the DAG)

- **Dempster-Shafer theory (for representing ignorance)**
  - Difference between probability of an event and ignorance
  - How to represent uncommitted belief (ignorance)
    - Lattice of subsets of frame of discernment
    - Basic probability assignment (function m(S))
    - Bel(S), Pls(S), and belief interval
  - Problem with this theory (high complexity)

- **Fuzzy set theory (for representing vague linguistic terms)**
  - Difference between fuzzy sets and ordinary sets
  - Fuzzy membership functions
  - Rules for fuzzy logic connectives
  - Problems with fuzzy logic (comparing with probability theory)

- **Uncertainty in rule-based system (certainty factors in MYCIN)**
  - CF of WM elements
  - CF of rules
  - CF propagation
  - Problems with CF