

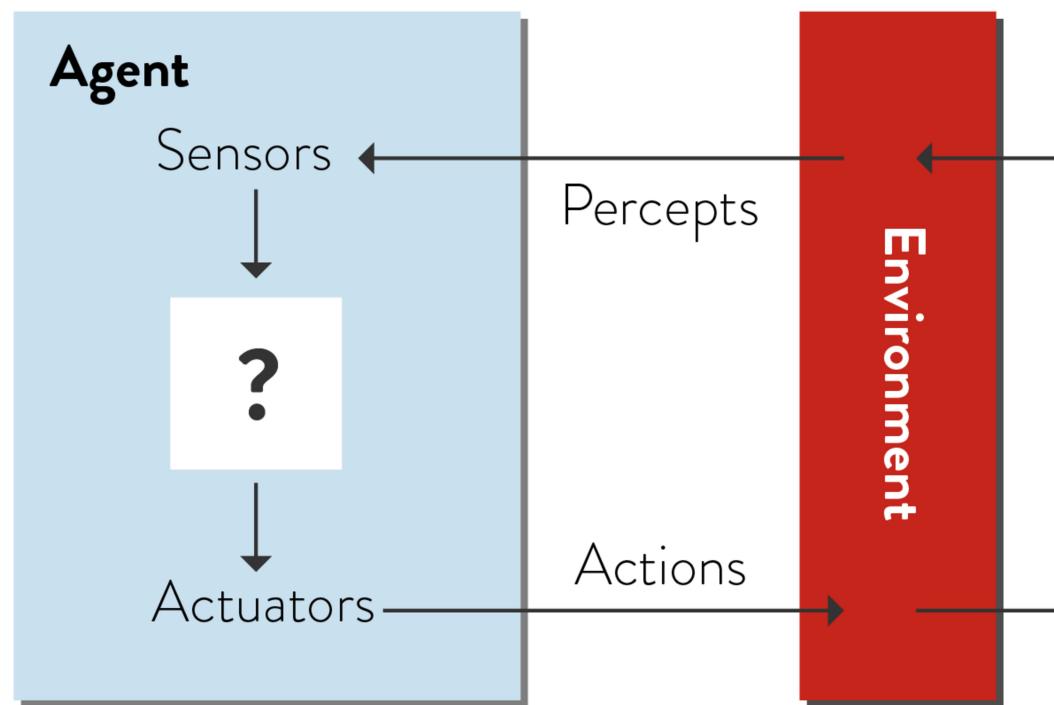
Agents

COMP3411/9814:
Artificial Intelligence

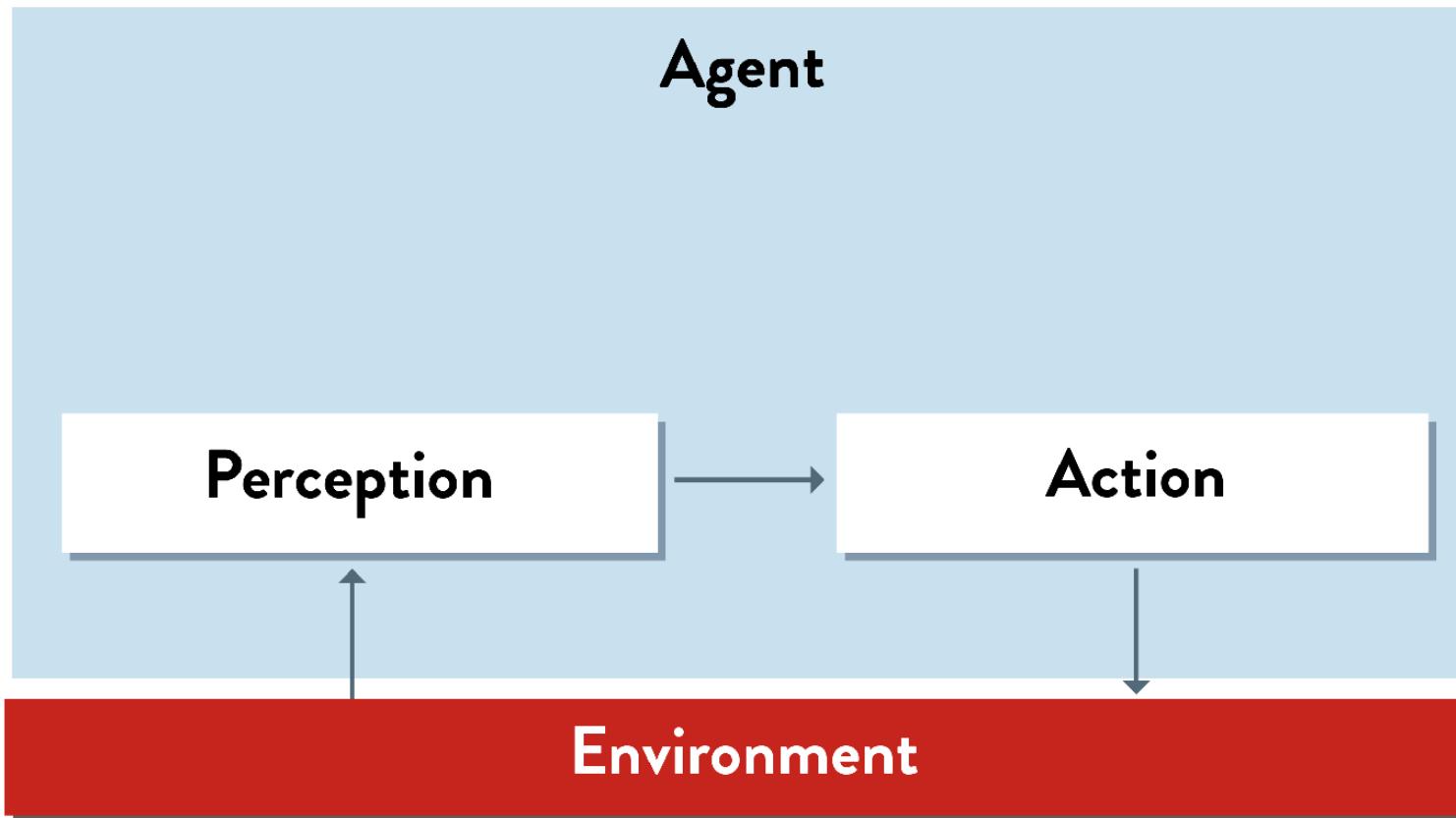
Types of Agents

- Reactive Agent
- Model-Based Agent
- Planning Agent
- Utility-based agent
- Game Playing Agent
- Learning Agent

Agent Model



Reactive Agent



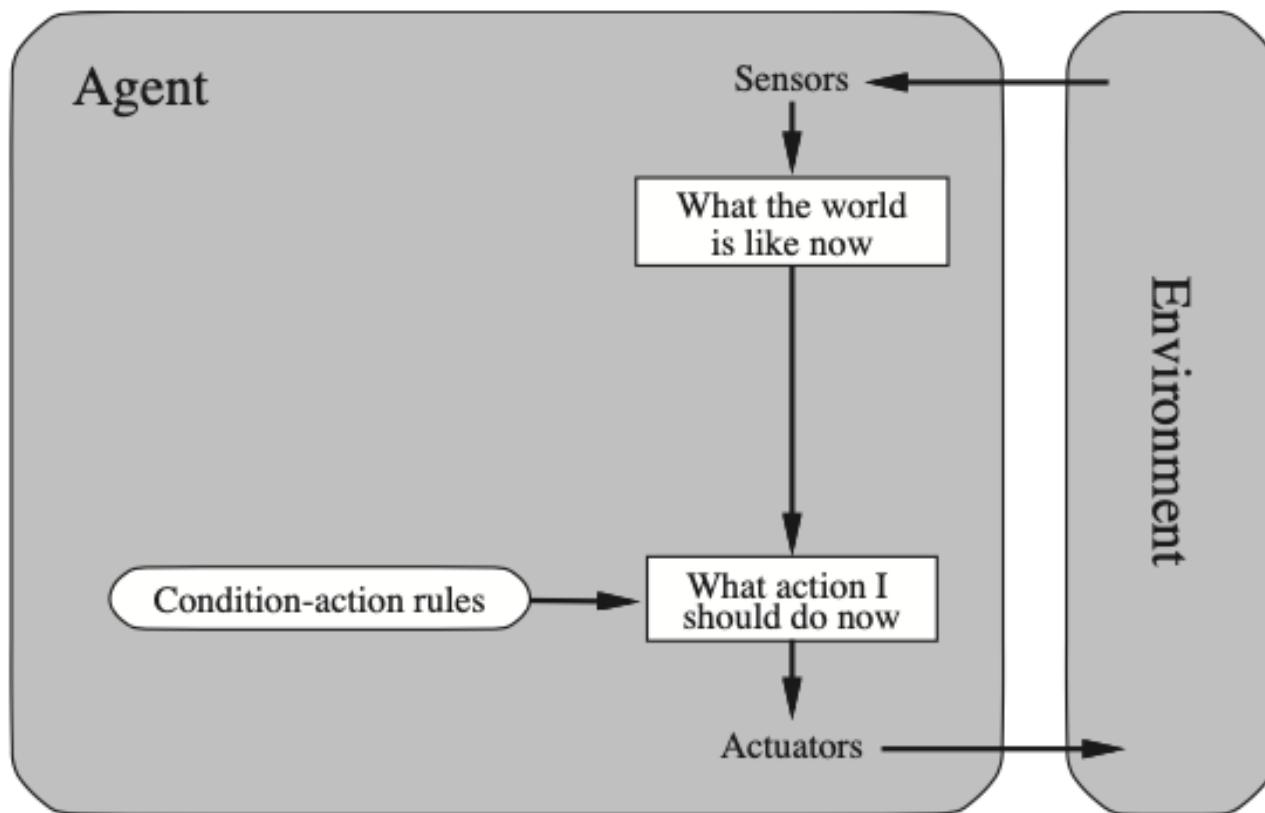
Reactive Agent

- Choose the next action based only on what agent currently perceives
 - Uses a “policy” or set of rules that are simple to apply
 - Sometimes called “simple reflex agents”
 - but they can do surprisingly sophisticated things

Reactive Agent



Reactive Agent

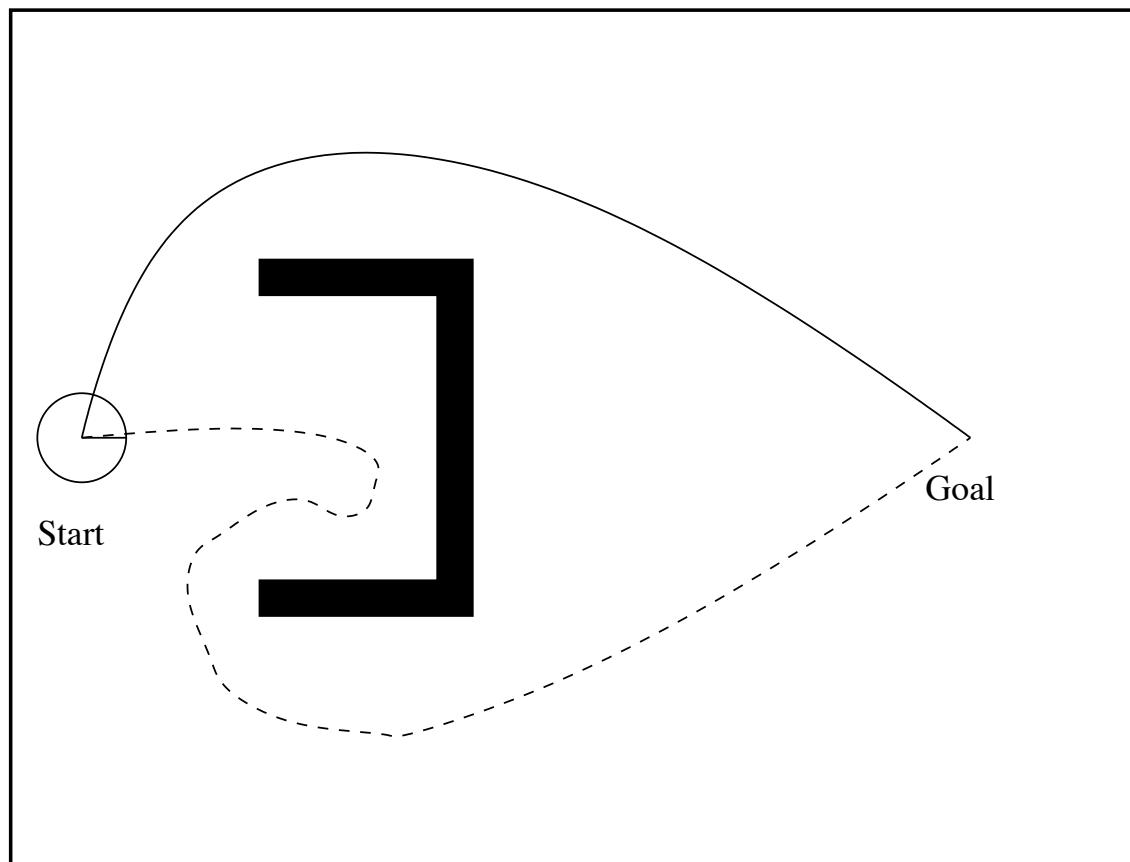


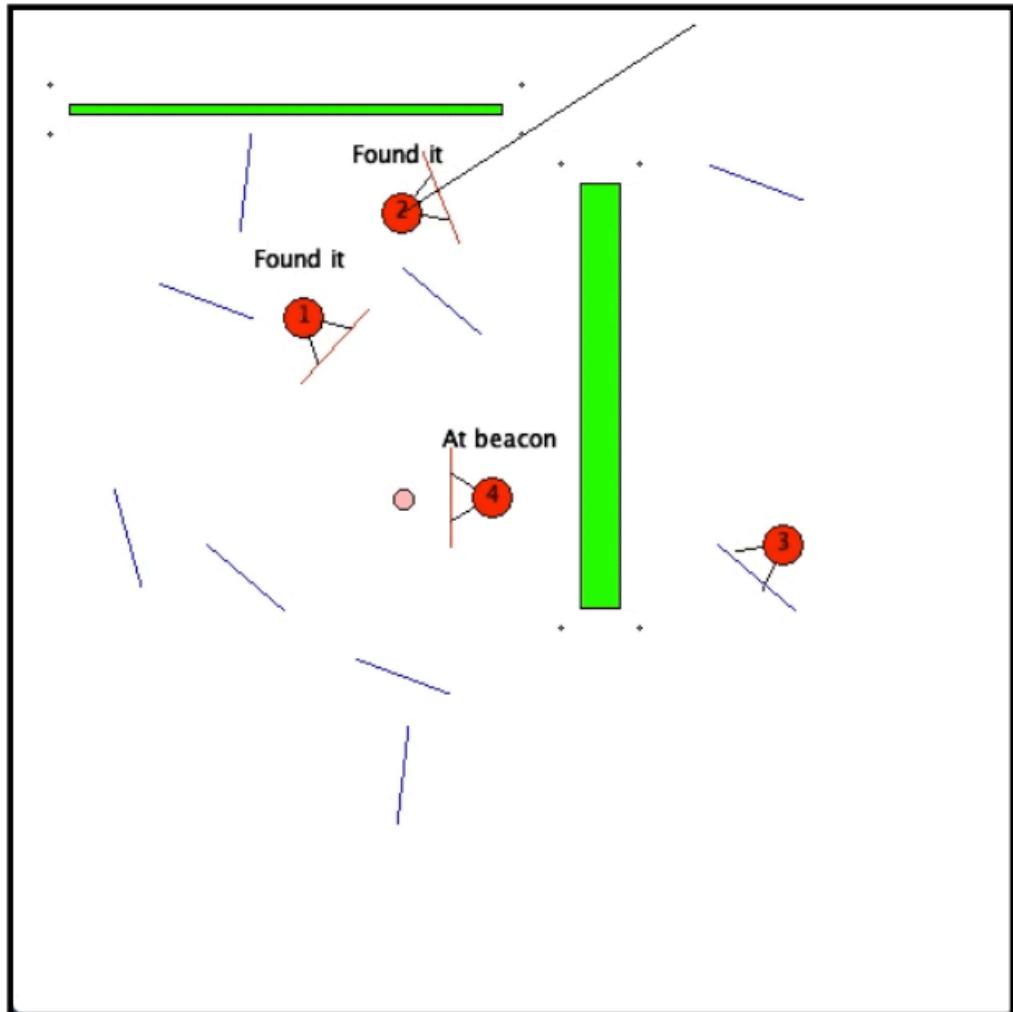
Reflex (reactive) agent — applies condition-action rules to each percept

Reactive Robots



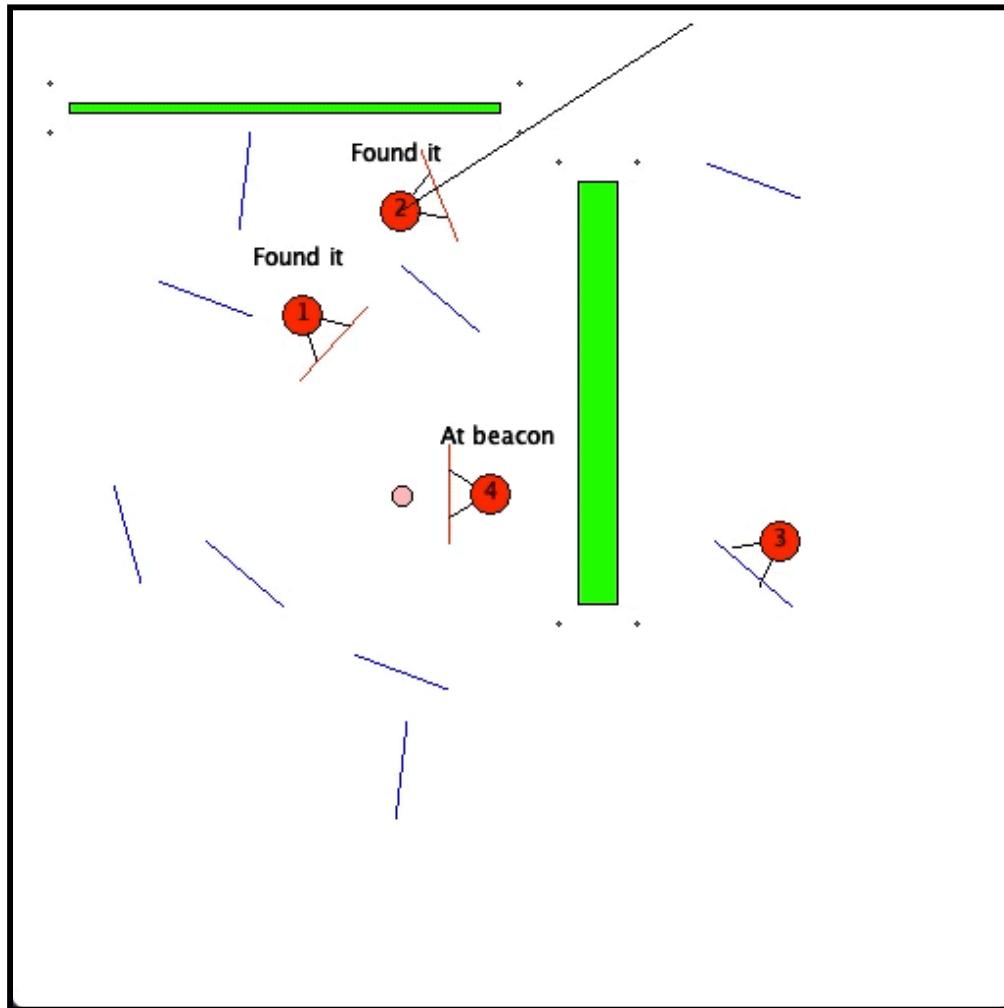
Limitations of Reactive Agents





```
botworld(512, 512,
[  
    box(1, 30, 50, 250, 55),  
    box(2, 290, 90, 310, 305),  
  
    beacon(1, 200, 250),  
  
    bar(1, 100, 150, 20),  
    bar(2, 200, 390, 275),  
    bar(3, 220, 100, 75),  
    bar(4, 380, 90, 20),  
    bar(5, 80, 80, 275),  
    bar(6, 60, 270, 75),  
    bar(7, 200, 340, 20),  
    bar(8, 120, 90, 275),  
    bar(9, 280, 250, 75),  
    bar(10, 120, 290, 40),  
    bar(11, 380, 290, 40),  
    bar(12, 220, 150, 40)  
],  
[  
    bot(1, "north"),  
    bot(2, "south"),  
    bot(3, "east"),  
    bot(4, "west")  
];
```

Teleo-Reactive Program (TOP)

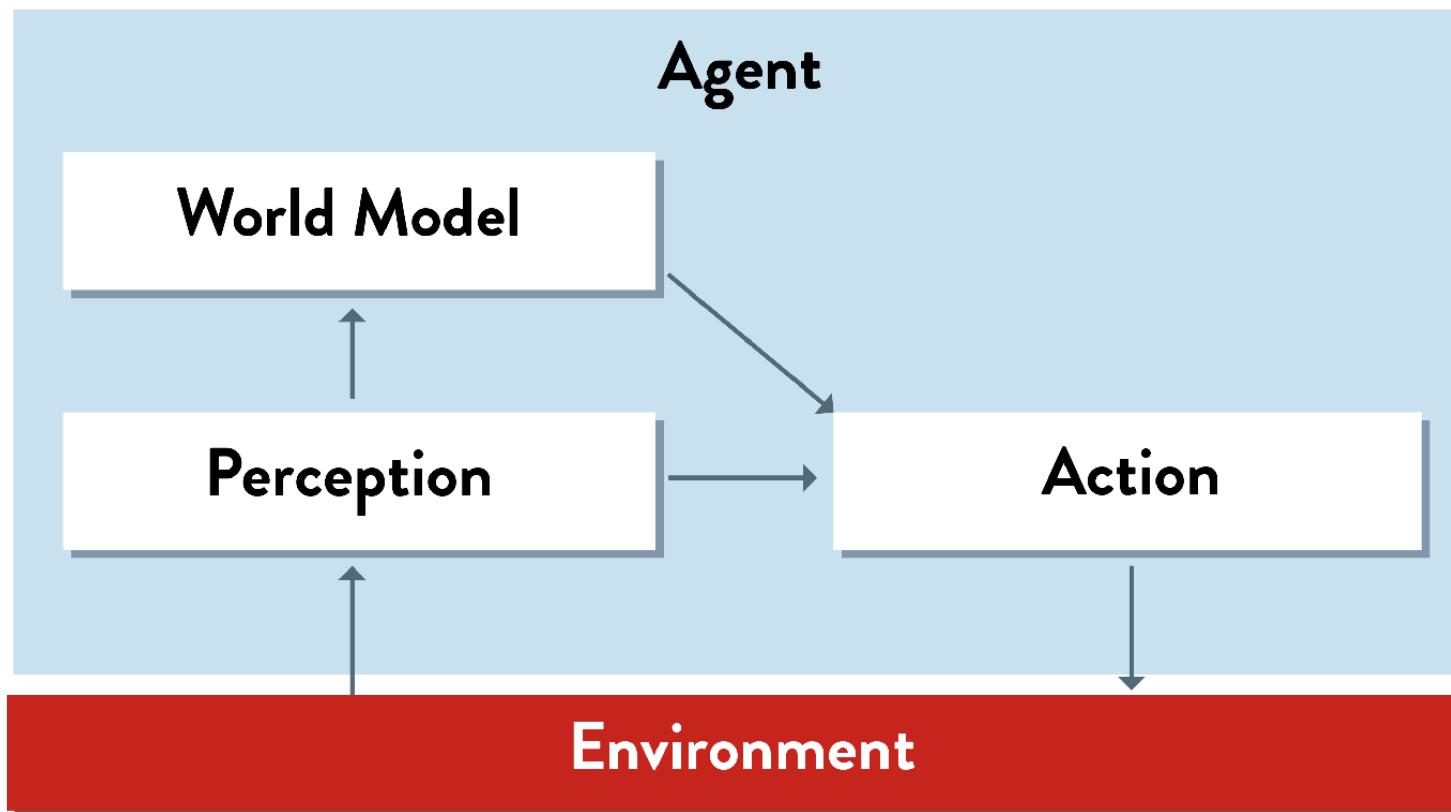


```
def bot(myNum, side) =  
{  
    var barNum = nearest_bar();  
  
    atBeacon(1, side) ->  
    {  
        say("At beacon");  
        turnTo(200, 250);  
        stop();  
    }  
    |  
    holding() and obstructedBeacon(1, side) -> find_opening(200)  
    |  
    holding() -> gotoBeacon(1, side)  
    |  
    gotoBar(barNum) ->  
    {  
        grab(barNum);  
        say("Found it");  
    }  
    |  
    true ->  
    {  
        turn(random(-180, 180));  
        move(random(50, 300));  
    };  
};  
  
def find_opening(dist) =  
obstructed_to(dist) -> turn(random(5, 15))  
|  
true -> move(random(0, dist))  
;
```

Limitations of Reactive Agents

- Reactive Agents have no memory or “state”
 - unable to base decision on previous observations
 - may repeat the same sequence of actions over and over
 - Escape from infinite loops is (sometimes) possible if the agent can randomise its actions.

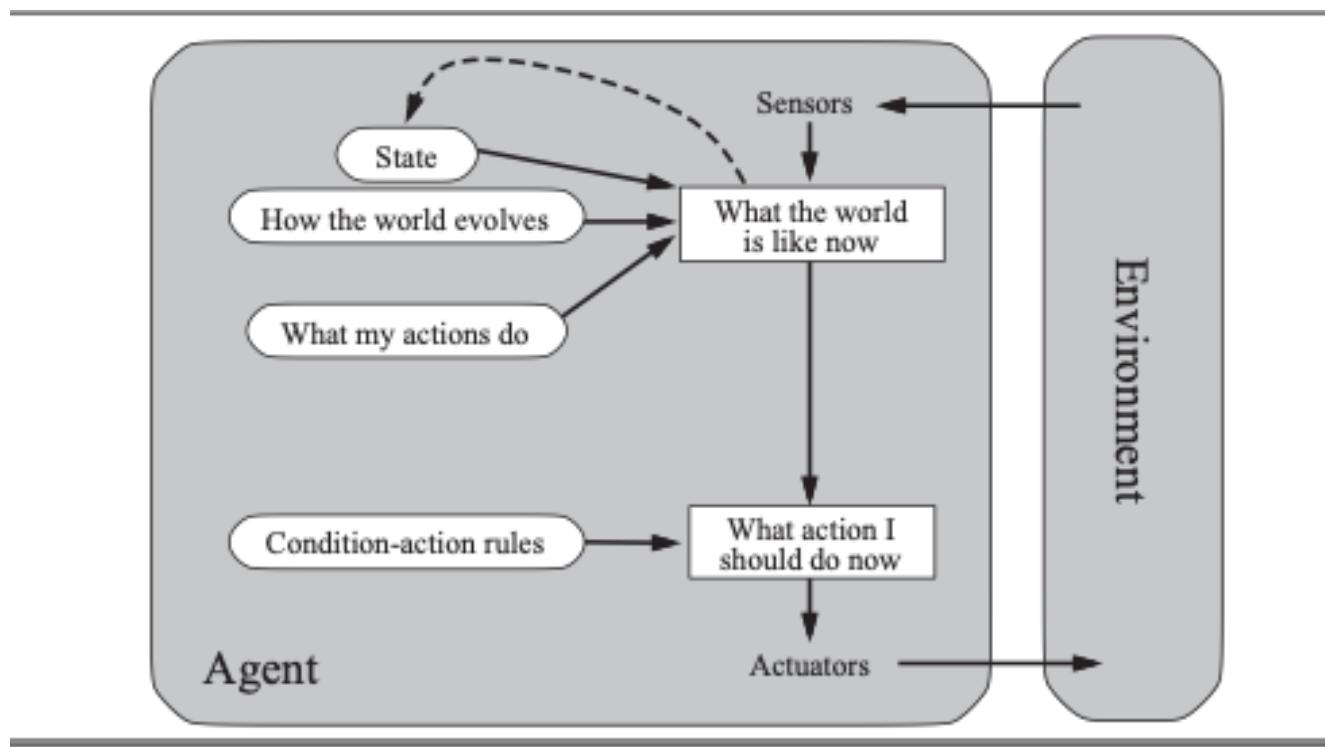
Model-Based Agent



Model-based Agents

- Handle *partial observability* by *keeping track of the part of the world it can't see now*.
- Maintain internal state that depends on the percept history and remembers at least some of the unobserved aspects of the current state.
- Knowledge about “how the world works” is called a **model** of the world.
- An agent that uses such a model is called a **model-based agent**.

Model-based Reflex Agent



A model-based reflex agent. It keeps track of the current state of the world, using an internal model. It then chooses an action in the same way as the reflex agent.

Model-based Reflex Agent

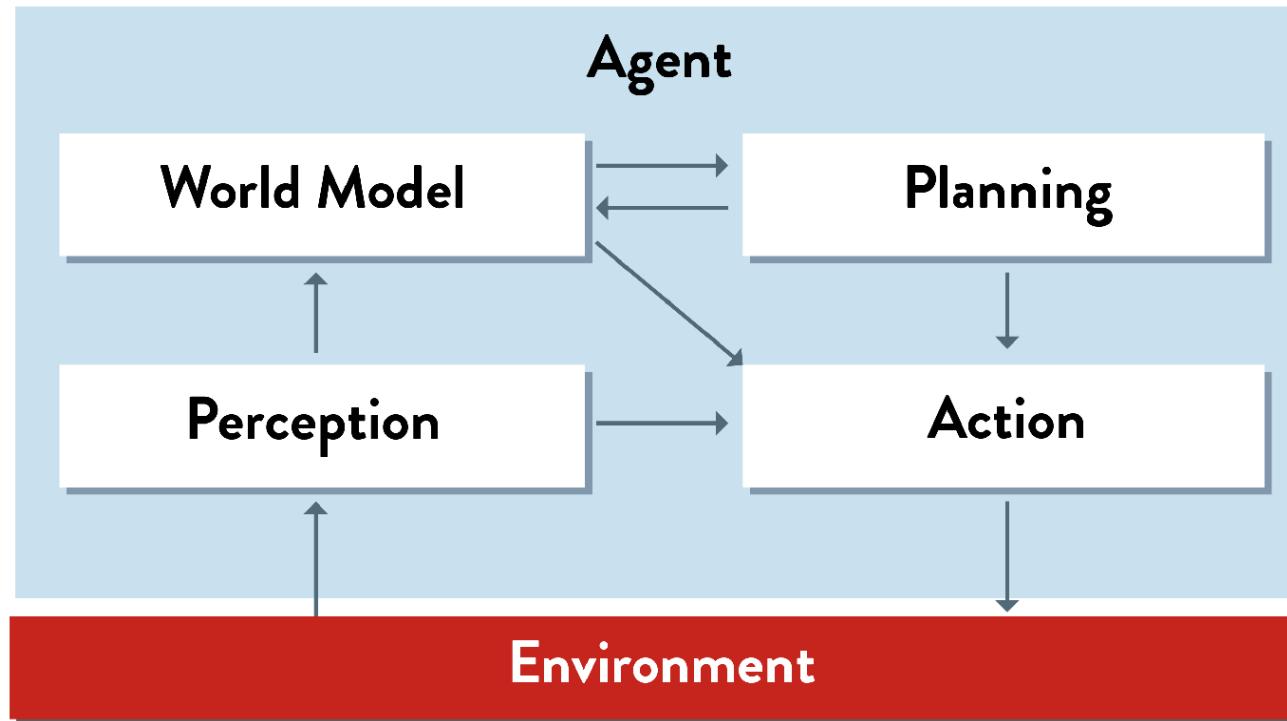


Limitations of Model-Based Agents

- An agent with a world model but no planning can look into the past, but not into the future; it will perform poorly when the task requires any of the following:
 - searching several moves ahead
 - Chess, Rubik's cube
 - complex tasks requiring many individual step
 - cooking a meal, assembling a watch
 - logical reasoning to achieve goals
 - travel to New York

Sometimes we may need to plan several steps into the future

Planning Agent



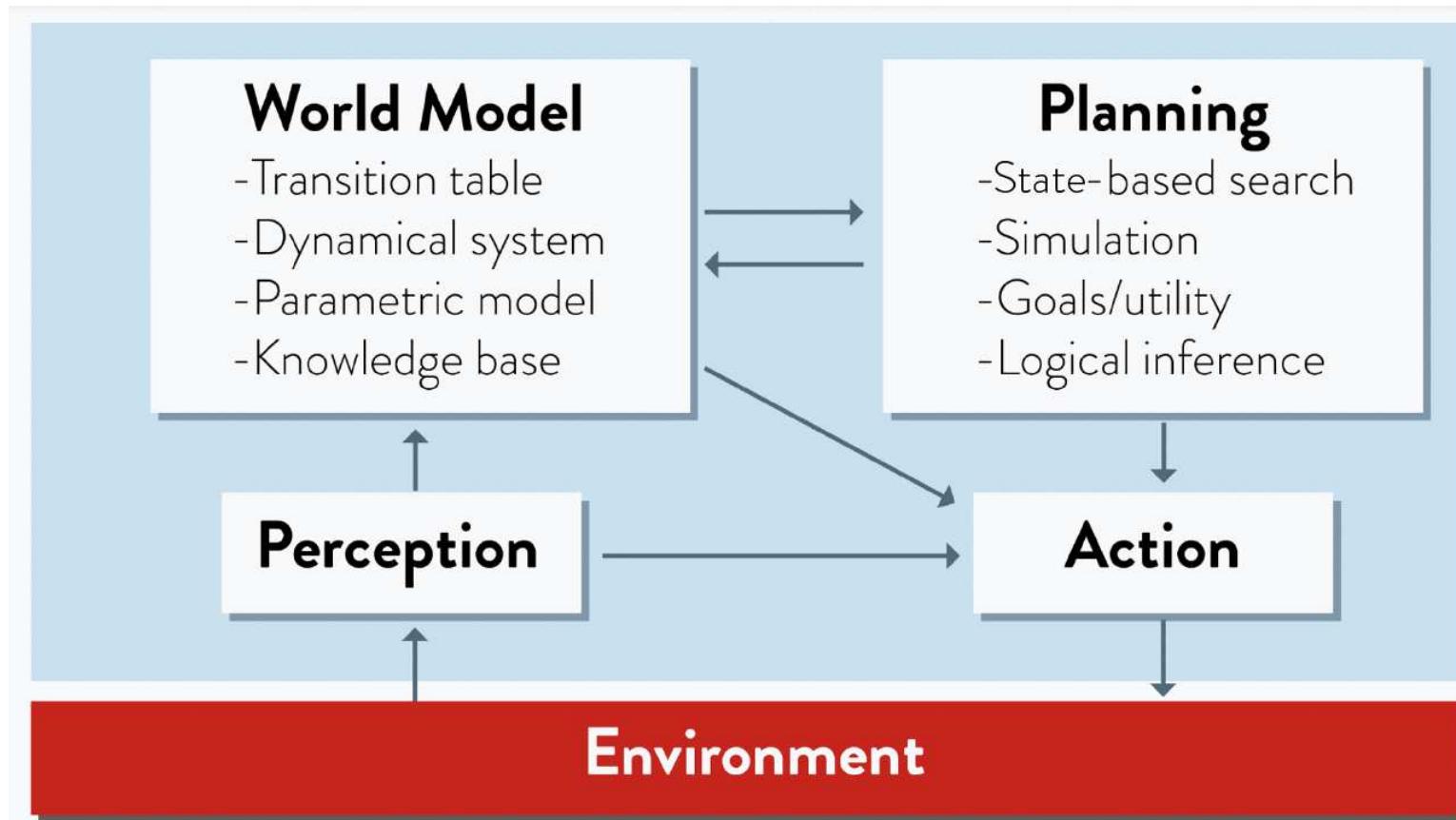
Goal-Based Agent

Planning Agent

- Decision making of this kind is fundamentally different from the condition–action rules
- It involves consideration of the future
 - “What will happen if I do such-and-such?” and
 - “Will that make me happy?”

In the reflex agent designs, this information is not explicitly represented, because the built-in rules map directly from states to actions

Models and Planning



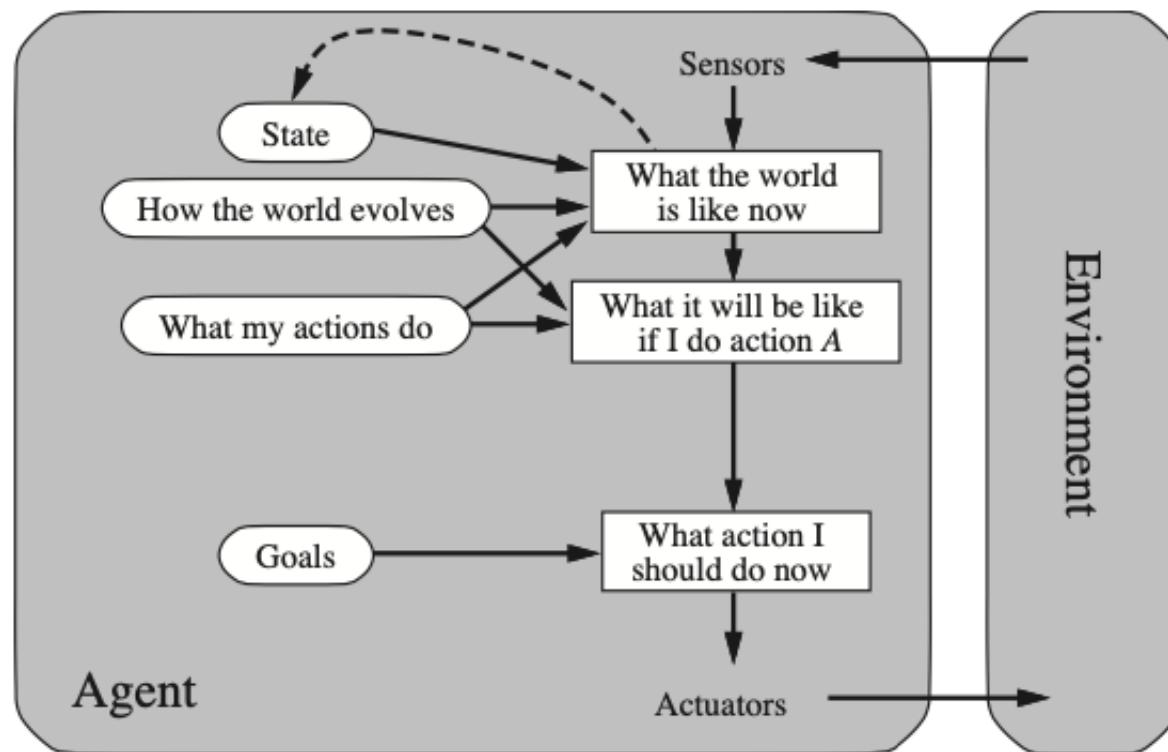
Reasoning about Future States

- What is the best action in this situation?
- Faking it
 - Sometimes an agent may appear to be planning ahead but is actually just applying reactive rules.
 - These rules can be hand-coded, or learned from experience.
 - Agent may not be flexible enough to adapt to new situations.

Planning Agent – Goal-based

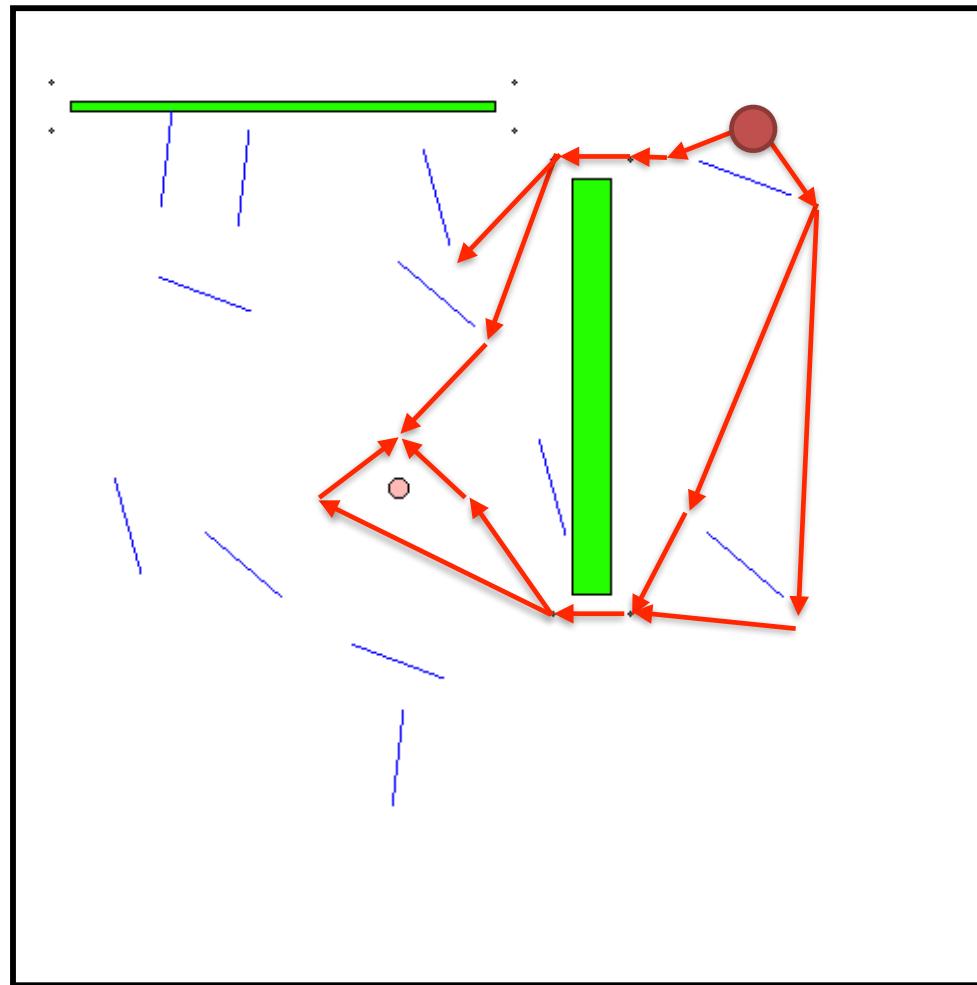
- The planning agent or goal-based agent is more flexible because the knowledge that supports its decisions is represented explicitly and **can be modified**.
- The agent's behaviour can easily be changed.
- But ...
 - it's slower to react because it has to “think” about what it's doing.

Goal-based (teleological) agent



- State description often not sufficient for agent to decide what to do
- Needs to consider its goals (may involve searching and planning)

Planning usually needs search



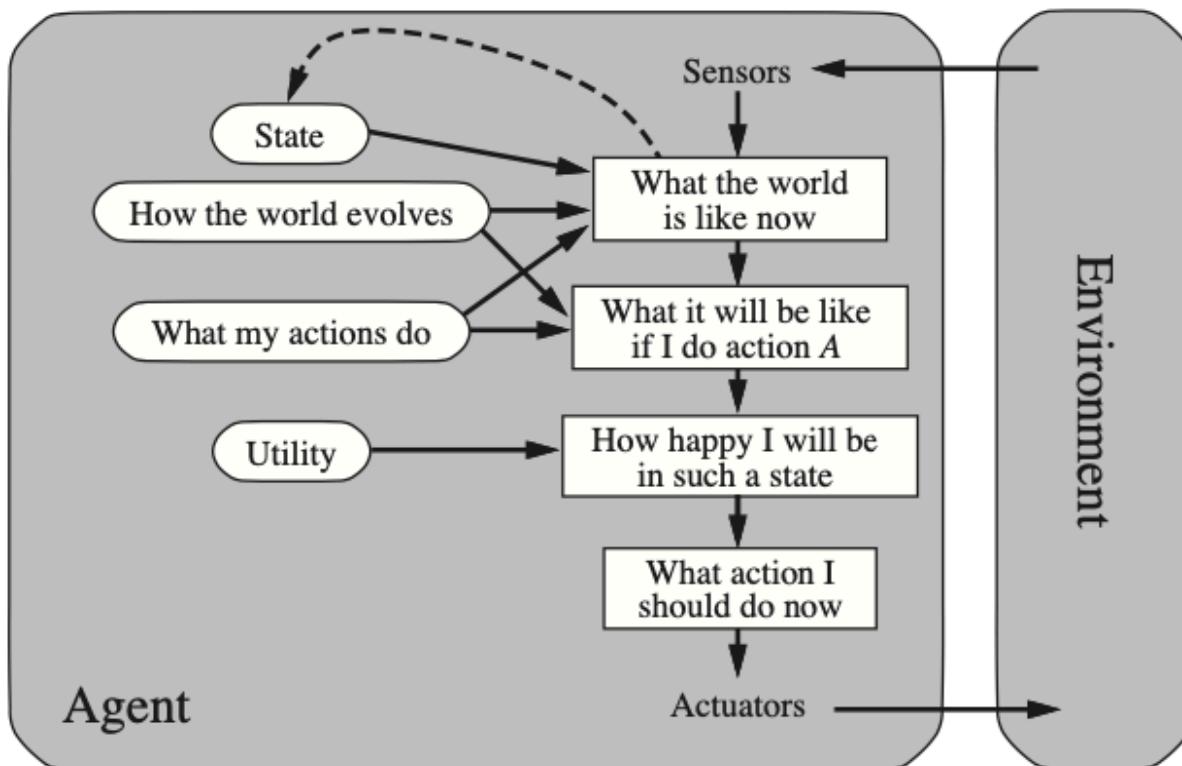
@Home Robot



Utility-based agent

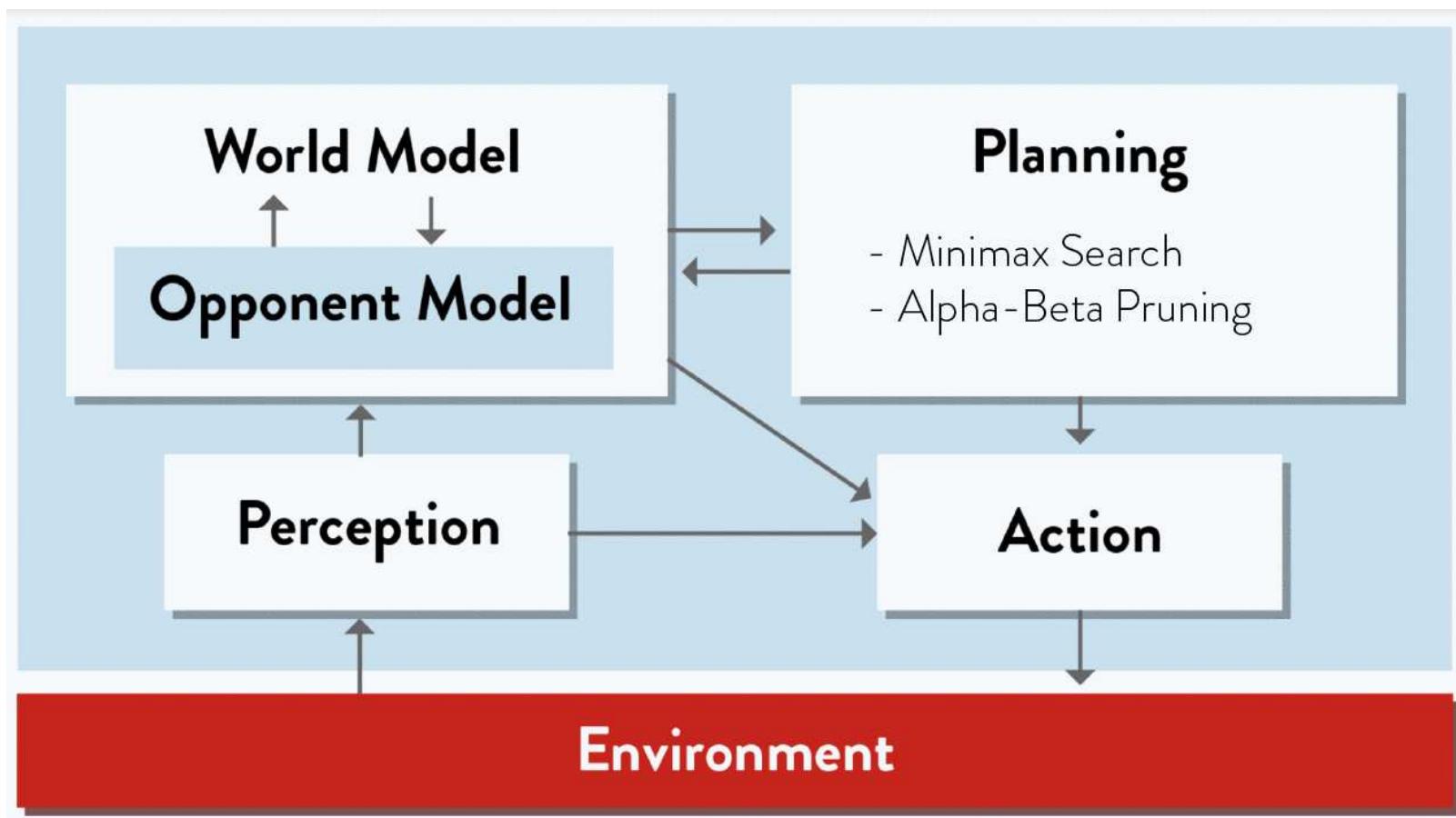
- A rational utility-based agent chooses the action that maximises the expected utility of the action outcomes
 - that is, the utility the agent expects to derive, on average, given the probabilities and utilities of each outcome.
- The utility-based agent is not easy to implement
 - It has to model and keep track of its environment
 - Tasks involved a great deal of research on perception, representation, reasoning, and learning.
 - It can be implemented as a **decision-making agent** that must handle the uncertainty inherent in stochastic or partially observable environments.

Utility-based agent

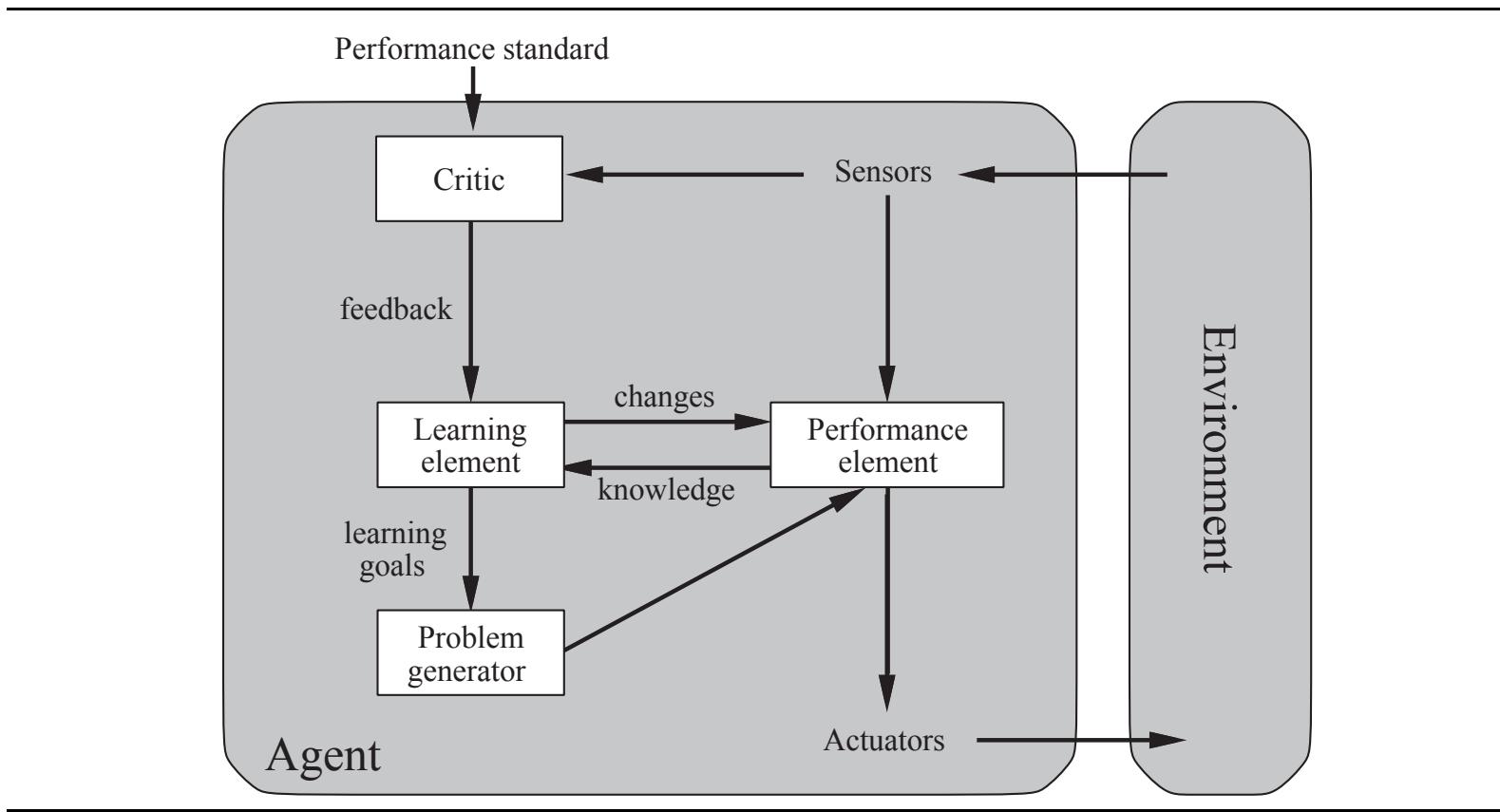


- Model-based, utility-based agent uses
 - model of world
 - utility function that measures preferences among states of world
- Chooses action that leads to best expected utility
 - Expected utility is computed by averaging over all possible outcome states
 - Weighted by probability of outcome.

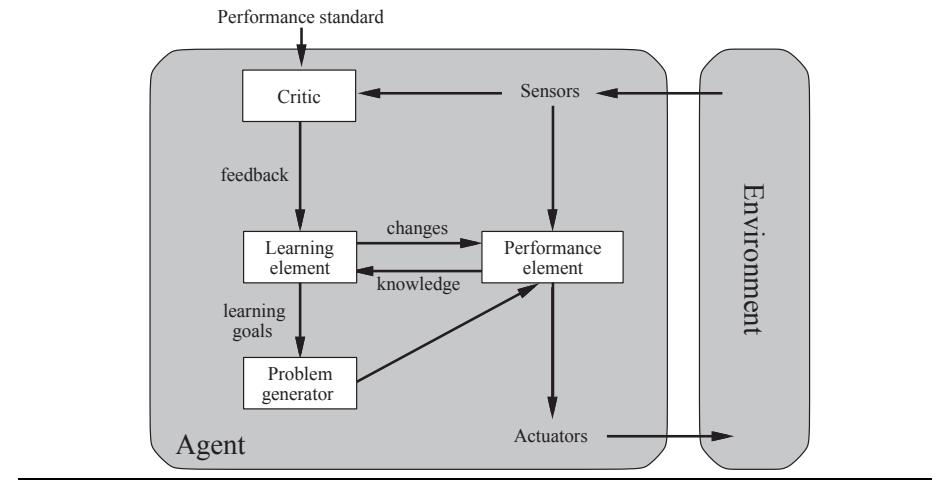
Game Playing Agent



Learning Agent

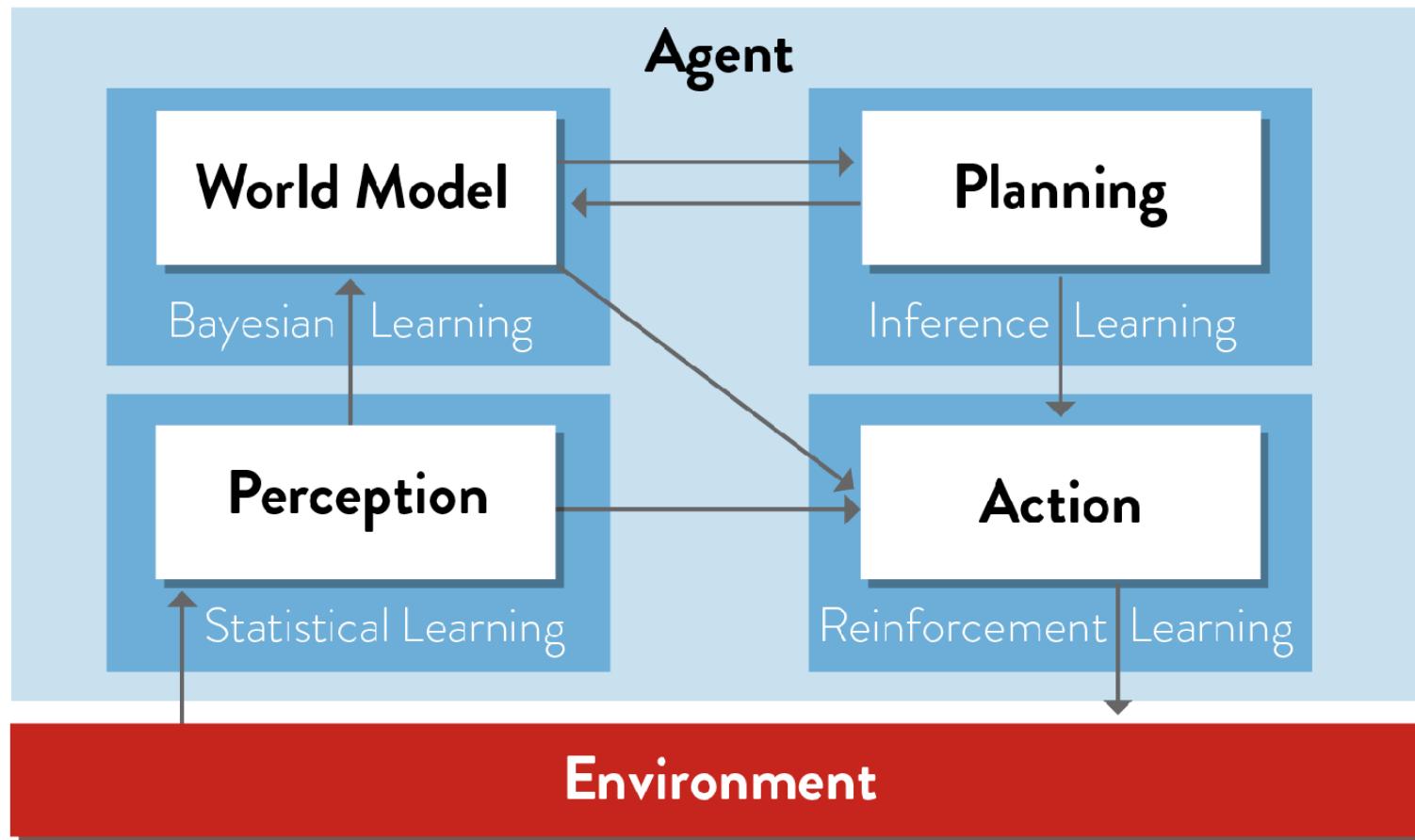


Learning Agent



- **Performance element** takes percepts; decides actions
- **Critic** gives feedback on how performance element is doing
- **Learning element** uses feedback to determine how performance element should be modified to do better in future
- **Problem generator** creates new tasks to provide new and informative experiences.

Learning Agent



Learning

- Learning is not a separate module, but rather a set of techniques for improving the existing modules
- Learning is necessary because:
 - may be difficult or even impossible for a human to design all aspects of the system by hand
 - the agent may need to adapt to new situations without being re-programmed by a human

Summary

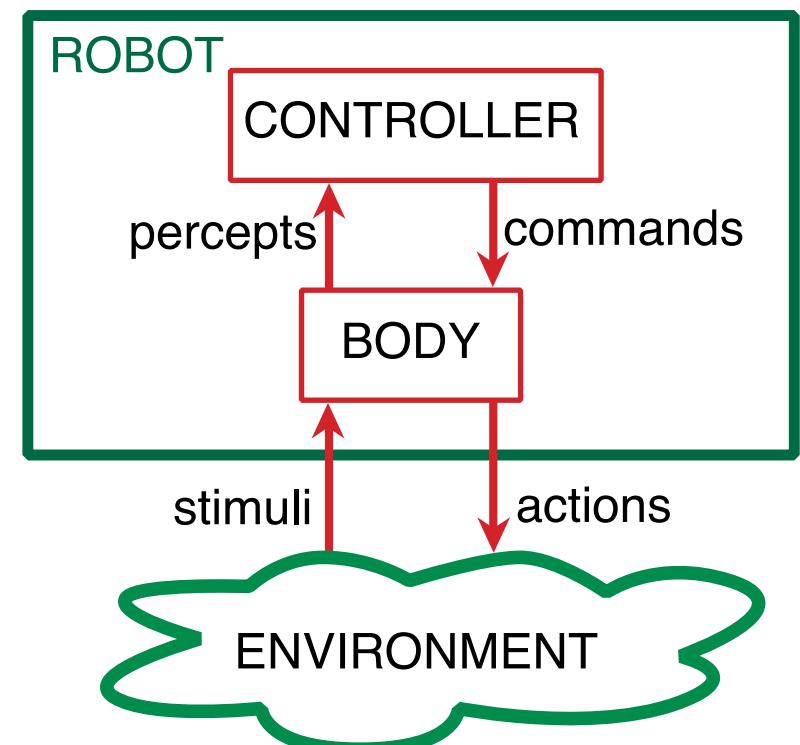
- **Reactive agents** respond directly to percepts
- **Model-based reflex agents** maintain internal state to track aspects of the world that are not evident in the current percept
- **Planning (Goal-based) agents** act to achieve their goals
- **Utility-based agents** try to maximise expected “happiness.”
- All agents can improve their performance through learning.

Representation and Search

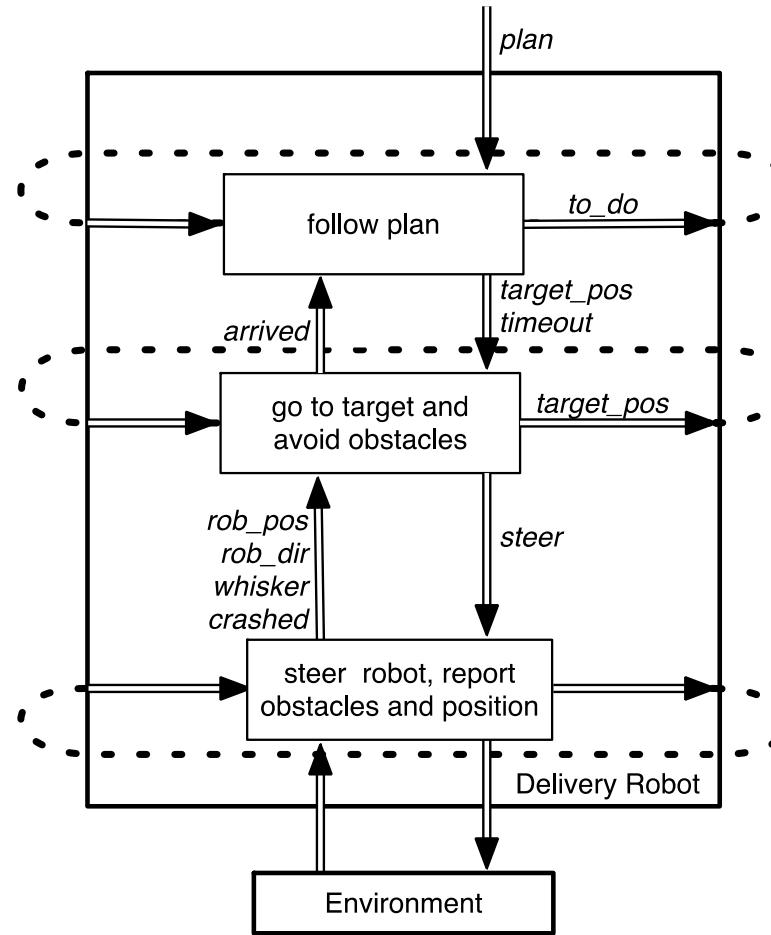
- The world model must be represented in a way that makes reasoning easy.
- Reasoning (problem solving and planning) in AI almost always involves some kind of search amongst possible solutions.

Layered Architecture

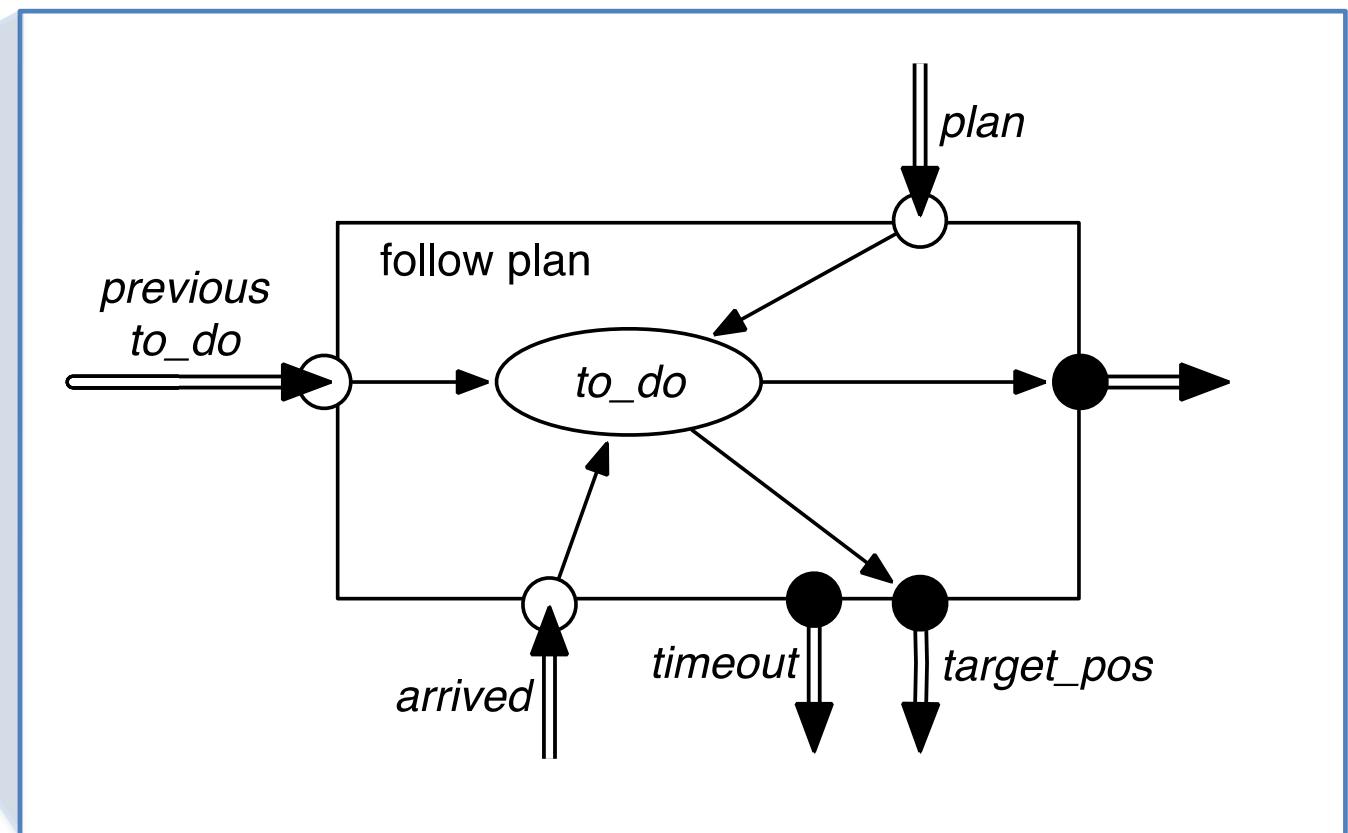
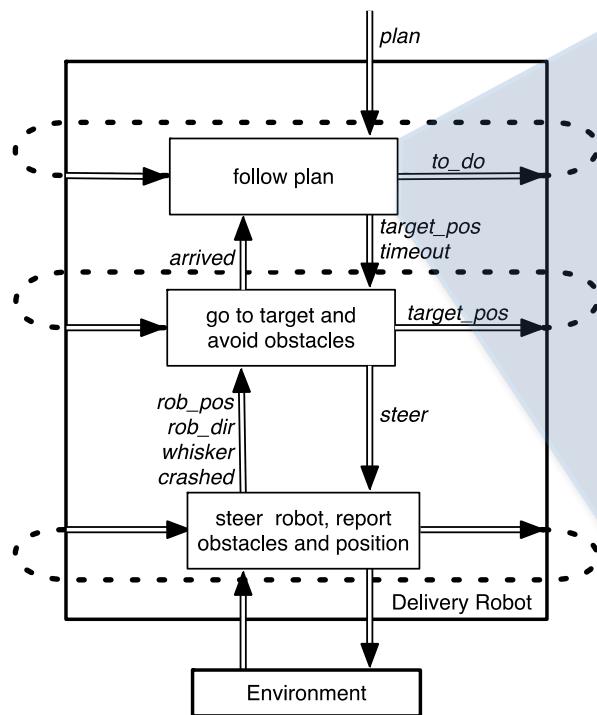
- Hierarchy of controllers
- Controller gets percepts from and sends commands to the lower layer
 - Abstracts low level features into higher level (perception)
 - Translates high level commands into actuator instructions (action)
- Controllers have different representations, programs
- Controllers operate at different time scales
- Lower-level controller can override its commands



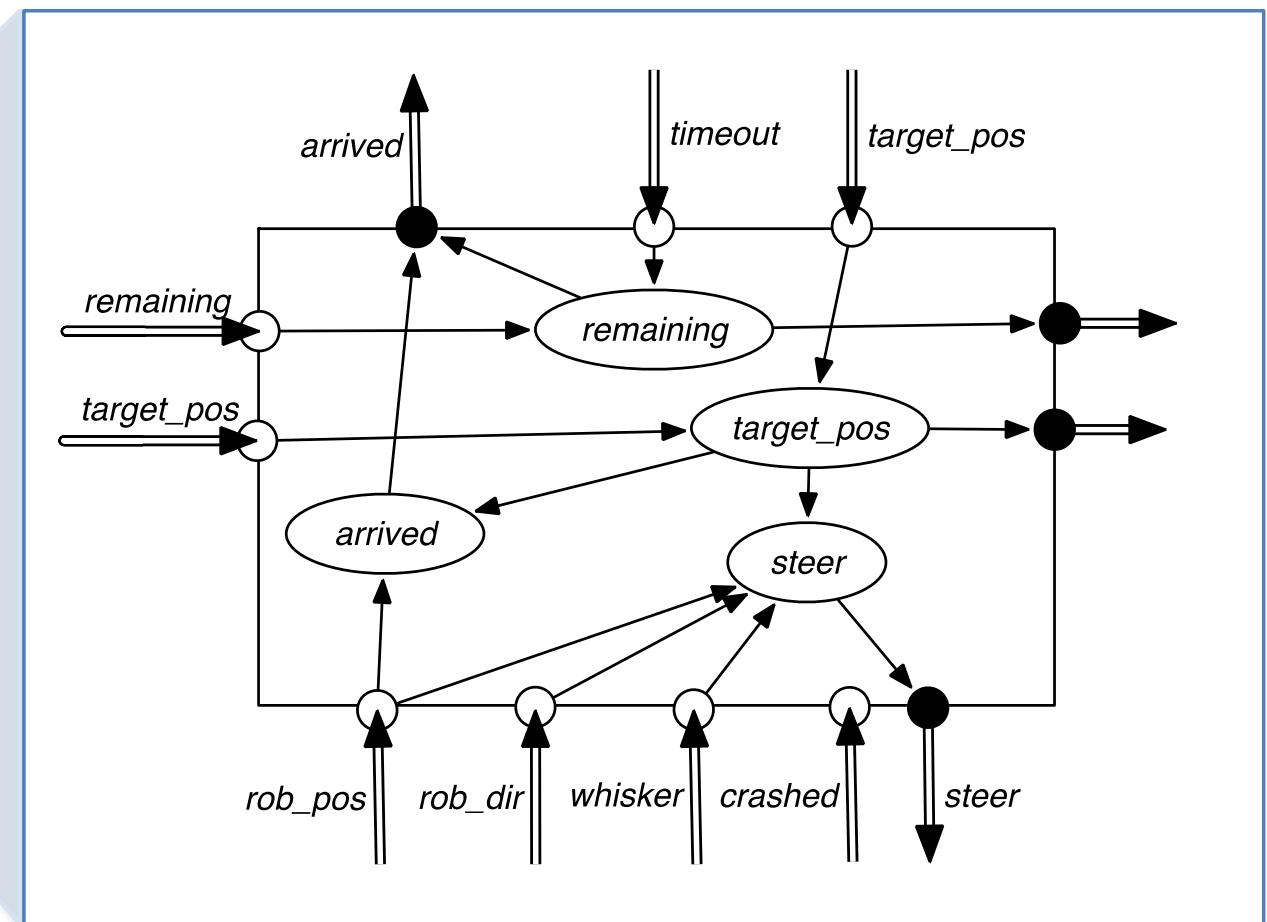
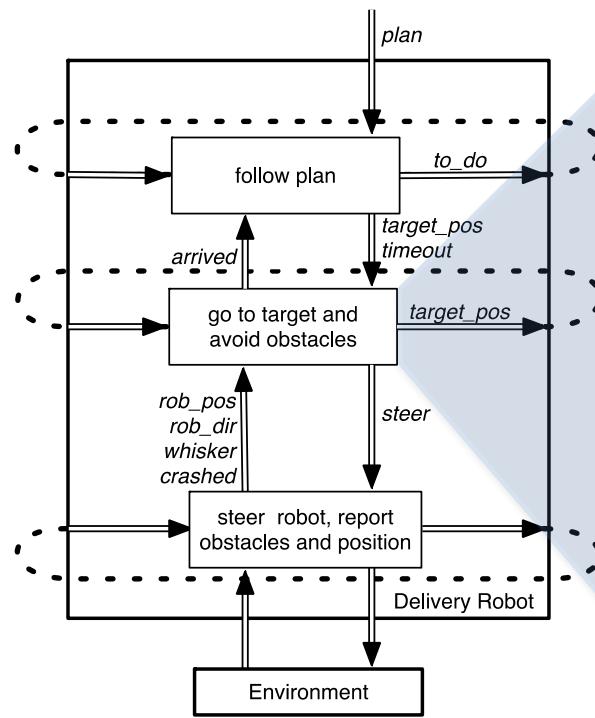
Example – Delivery Robot



Delivery Robot – Top Layer



Delivery Robot – Middle Layer



Delivery Robot – TR Code Example

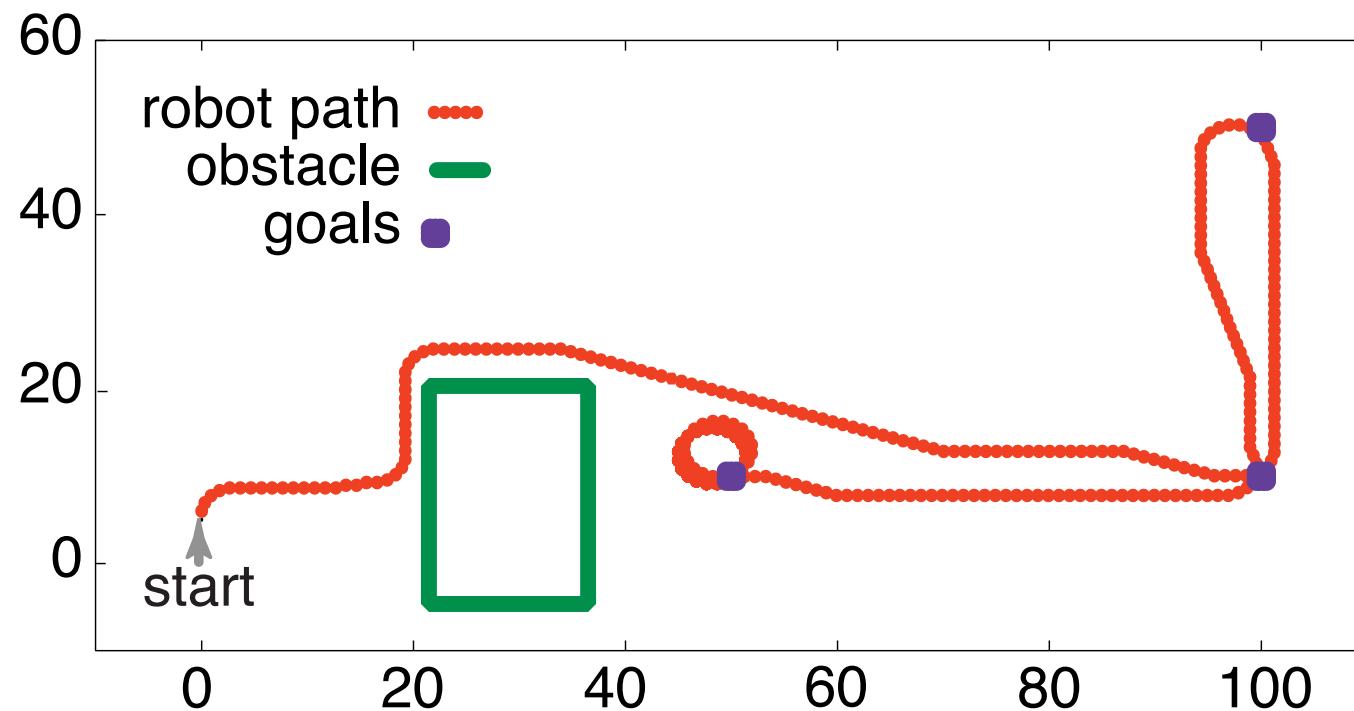
followPlan(to_do):

```
empty(to_do) → stop;  
  
arrived() or timeout() →  
{  
    resetTimer(200);  
    plan := rest(to_do);  
}  
  
true → goToTarget(coordinates(first(to_do));
```

goToTarget(target_pos):

```
arrived() or timeout() → {set arrived; stop;}  
  
whisker_sensor = on → steer left;  
  
straight_ahead(rob_pos, robot_dir, target_pos) → steer(straight);  
  
left_of(rob_pos, robot_dir, target_pos) → steer(left);  
  
true → steer(right)
```

Delivery Robot – Simulation



References

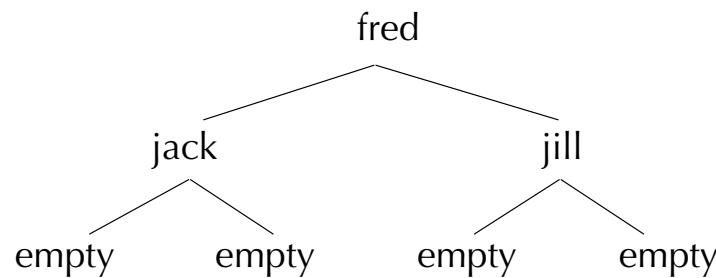
- Poole & Mackworth, Artificial Intelligence: Foundations of Computational Agents, Chapter 1 & 2
- Russell & Norvig, *Artificial Intelligence: a Modern Approach*, Chapter 2.

Recursive Programs

- Compound terms can contain other compound terms.
- A compound term can contain the same kind of term, i.e. it can be *recursive*.

tree(tree(empty, jack, empty), fred, tree(empty, jill, empty))

- "empty" is an arbitrary symbol used to represent the empty tree.
- A structure like this could be used to represent a binary tree that looks like:



Binary Trees

- A binary tree is either empty or it is a structure that contains data and left and right subtrees which are also trees.
- To test if some datum is in the tree:

```
in_tree(X, tree(_, X, _)).  
in_tree(X, tree(Left, Y, _)) :-  
    X \= Y,  
    in_tree(X, Left).  
in_tree(X, tree(_, Y, Right)) :-  
    X \= Y,  
    in_tree(X, Right).
```

The size of a tree

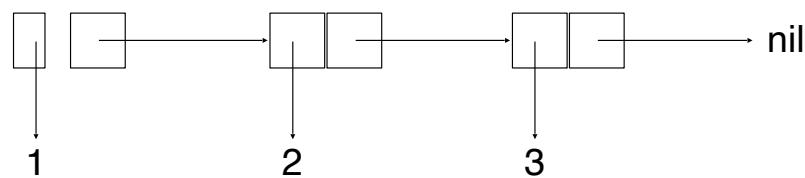
- The size of the empty tree is 0.
- The size of a non-empty tree is the size of the left subtree plus the size of the right subtree plus one for the current node.

```
tree_size(empty, 0).  
tree_size(tree(Left, _, Right), N) :-  
    tree_size(Left, LeftSize),  
    tree_size(Right, RightSize),  
    N is LeftSize + RightSize + 1.
```

Lists

- A list may be nil or it may be a term that has a head and a tail. The tail is another list.
- A list of numbers, [1, 2, 3] can be represented as:

```
list(1, list(2, list(3, nil)))
```



- Since lists are used so often, Prolog has a special notation:

```
[1, 2, 3] = list(1, list(2, list(3, nil)))
```

Examples of Lists

```
?- [x, y, z] = [1, 2, 3].
```

```
x = 1
```

```
y = 2
```

```
z = 3
```

Unify the two terms on either side of the equals sign.

Variables match terms in corresponding positions.

```
?- [x | y] = [1, 2, 3].
```

```
x = 1
```

```
y = [2, 3]
```

The head and tail of a list are separated by using '|' to indicate that the term following the bar should unify with the tail of the list

```
?- [x | y] = [1].
```

```
x = 1
```

```
y = []
```

The empty list is written as '[]'.

The end of a list is *usually* '[]'.

More list examples

```
?- [x, y | z] = [fred, jim, jill, mary].
```

There must be at least two elements in the list on the right

```
x = fred  
y = jim  
z = [jill, mary]
```

```
?- [x | y] = [[a, f(e)], [n, b, [2]]].
```

The right hand list has two elements:

```
x = [a, f(e)]  
y = [[n, b, [2]]]
```

[a, f(e)] [n, b, [2]]
Y is the tail of the list, [n, b, [2]] is just one element

List Membership

```
member(X, [X | _]).  
member(X, [_ | Y]) :-  
    member(X, Y).
```

Rules about writing recursive programs:

- Only deal with one element at a time.
- Believe that the recursive program you are writing has already been written and works.
- Write definitions, not programs.

Concatenating Lists

```
conc([1, 2, 3], [4, 5], [1, 2, 3, 4, 5])
```

Start planning by considering simplest case:

```
conc([], [1, 2, 3], [1, 2, 3])
```

Clause for this case:

```
conc([], X, X).
```

Concatenating Lists

Next case:

```
conc([1], [2], [1, 2])
```

Since **conc([], [2], [2])**

```
conc([A | B], C, [A | D]) :- conc(B, C, D).
```

Entire program is:

```
conc([], X, X).  
conc([A | B], C, [A | D]) :-  
    conc(B, C, D).
```

Reversing Lists

```
rev([1, 2, 3], [3, 2, 1])
```

Start planning by considering simplest case:

```
rev([], [])
```

Note:

```
rev([2, 3], [3, 2])
```

```
rev([], []).
rev([A | B], C) :-
    rev(B, D),
    conc(D, [A], C).
```

and

```
conc([3, 2], [1], [3, 2, 1])
```

An Application of Lists

Find the total cost of a list of items:

```
cost(flange, 3).  
cost(nut, 1).  
cost(widget, 2).  
cost(splice, 2).
```

We want to know the total cost of [flange, nut, widget, splice]

```
total_cost([], 0).  
total_cost([A | B], C) :-  
    total_cost(B, B_cost),  
    cost(A, A_cost),  
    C is A_cost + B_cost.
```

Reference

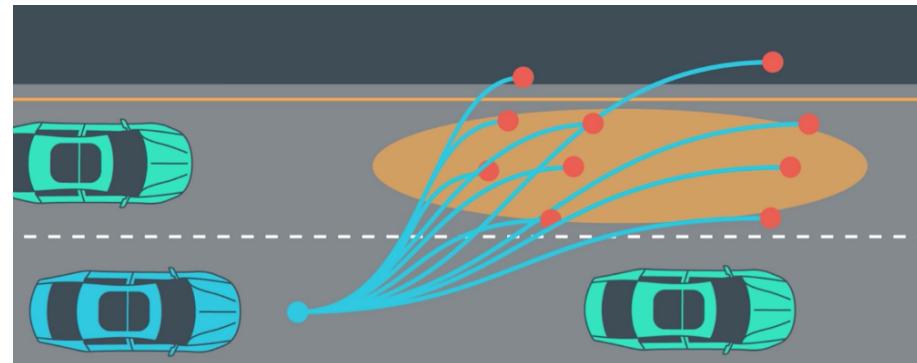
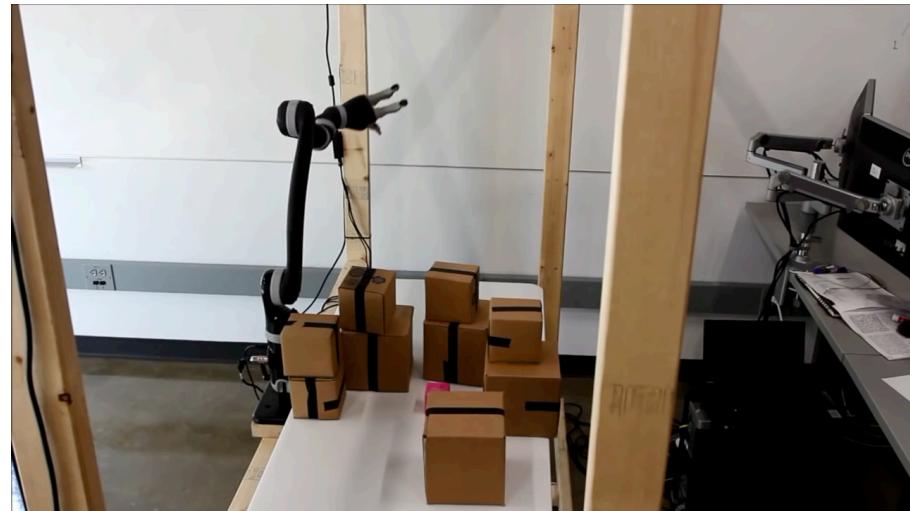
- Ivan Bratko, *Programming in Prolog for Artificial Intelligence*, 4th Edition, Pearson, 2013.

Uninformed Search

COMP3411/9814: Artificial Intelligence

When is Search Needed?

- Motion Planning
- Navigation
- Speech and Natural Language
- Task Planning
- Machine Learning
- Game Playing



Search Methods

- Uninformed search
 - use no problem-specific information
 - Uninformed (or “blind”) search strategies use only the information available in the problem definition (can only distinguish a goal from a non-goal state)
- Informed search
 - use heuristics to improve efficiency
 - Informed (or “heuristic”) search strategies use task-specific knowledge.

Overview

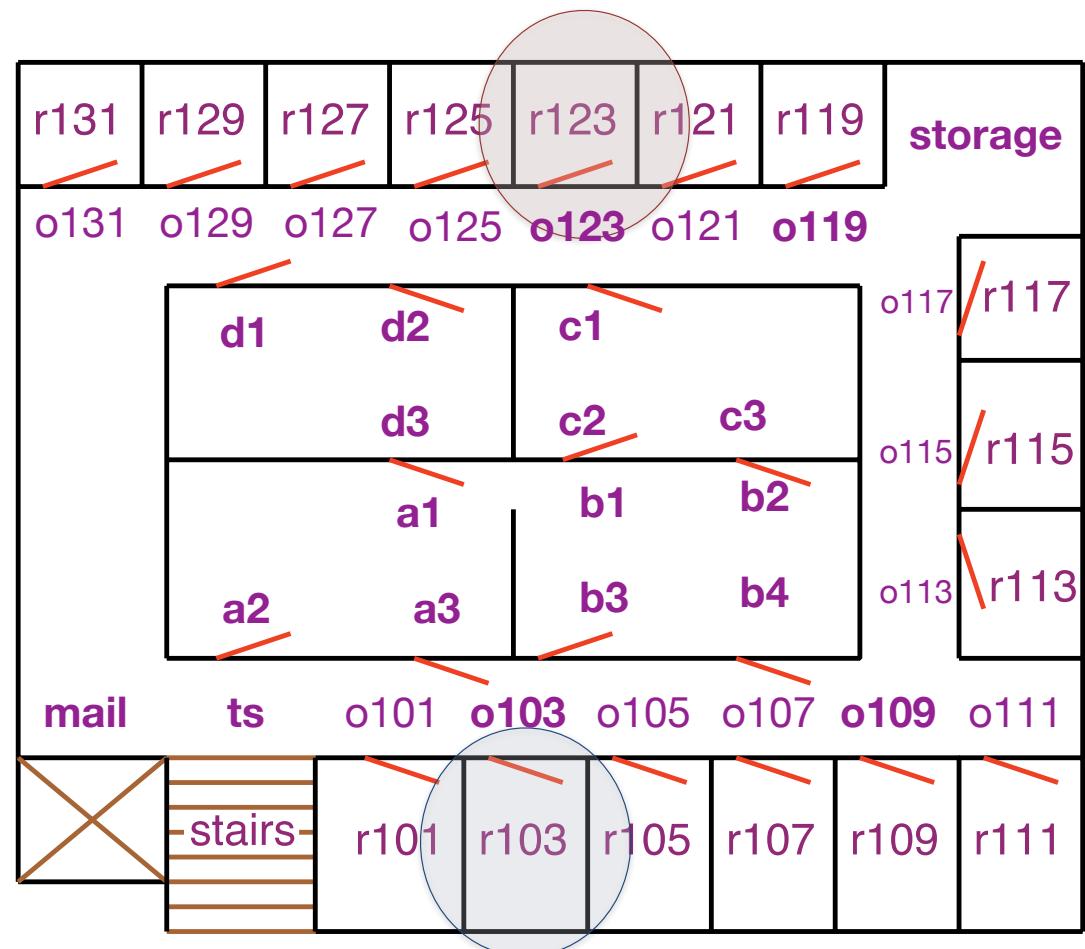
- Basic search algorithms
 - Breadth First Search
 - Depth First Search
 - Uniform Cost Search
 - Depth Limited Search
 - Iterative Deepening Search
 - Bidirectional Search

State Space Search Problems

- **State space** — set of all states reachable from initial state(s) by any action sequence
- **Initial state(s)** — element(s) of the state space
- Transitions
 - **Operators** — set of possible actions at agent's disposal; describe state reached after performing action in current state, or
 - **Successor function** — $s(x)$ = set of states reachable from state x by performing a single action
- **Goal state(s)** — element(s) of the state space
- **Path cost** — cost of a sequence of transitions used to evaluate solutions (applies to optimisation problems)

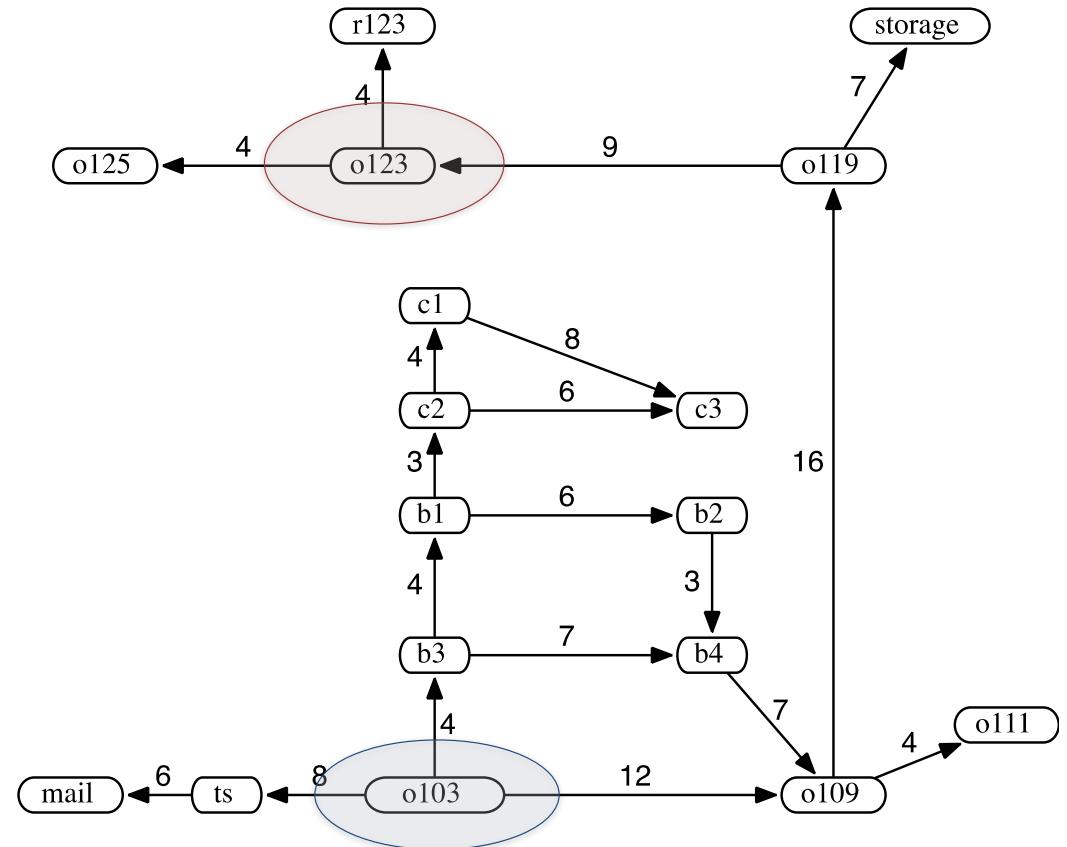
Delivery Robot

- The robot wants to get from outside room 103 to the inside of room 123.
- The only way a robot can get through a doorway is to push the door open in the direction shown.
- The task is to find a path from o103 to r123



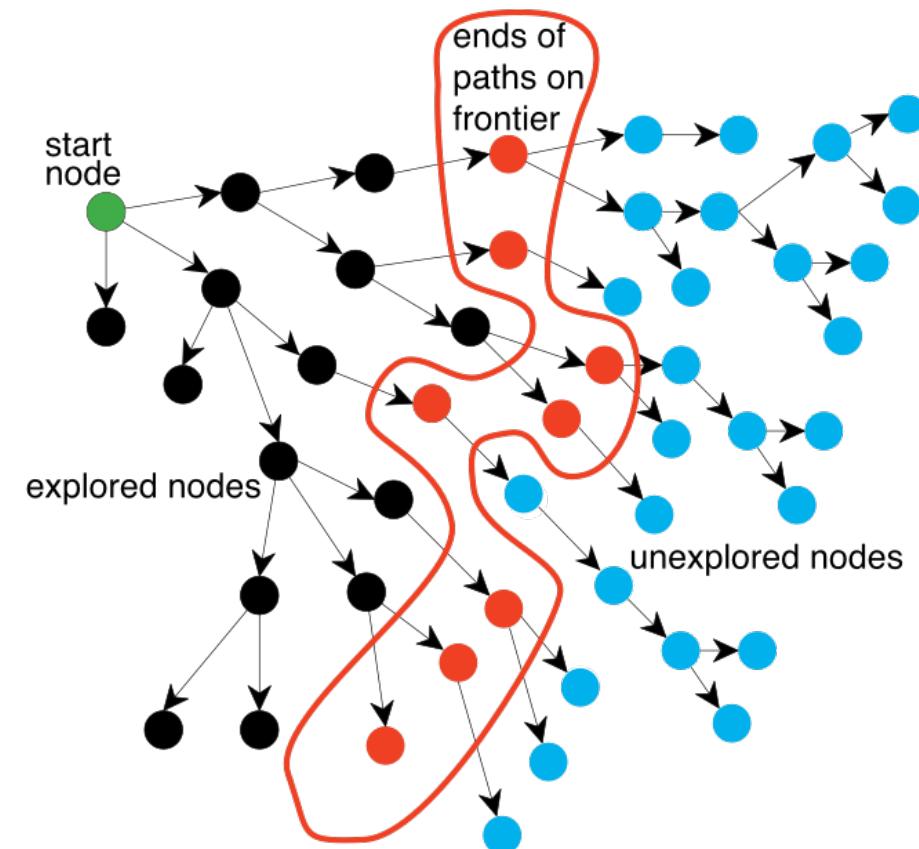
State-Space Graph for Delivery Robot

- Modelled as a state-space search problem
- States are locations.



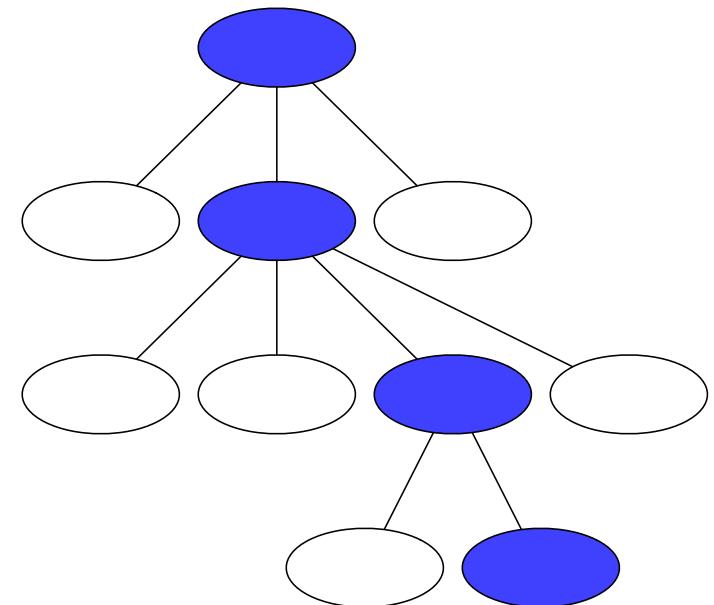
Problem Solving by Graph Searching

Search strategy differ in the way they expand the frontier

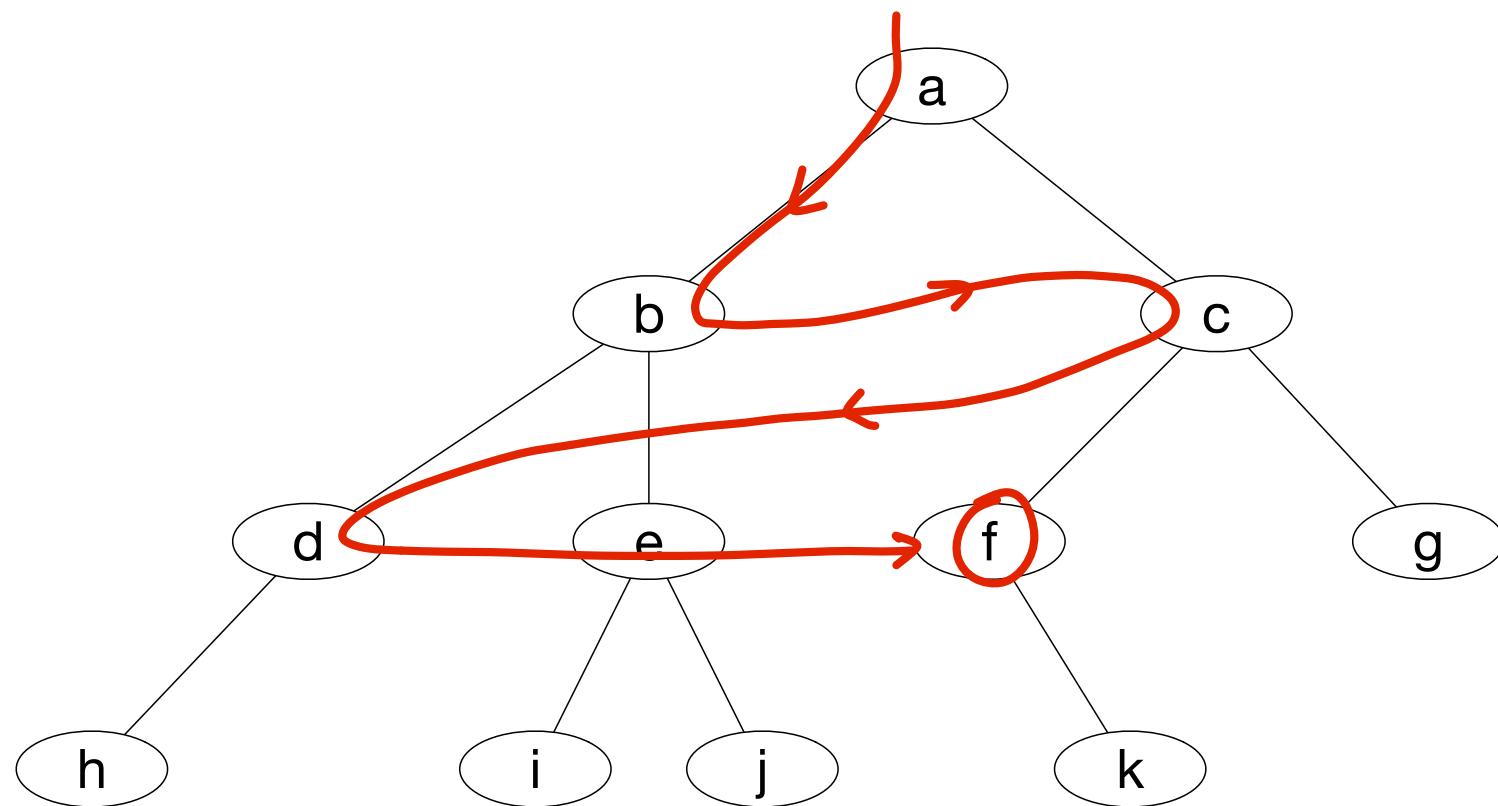


Search Tree

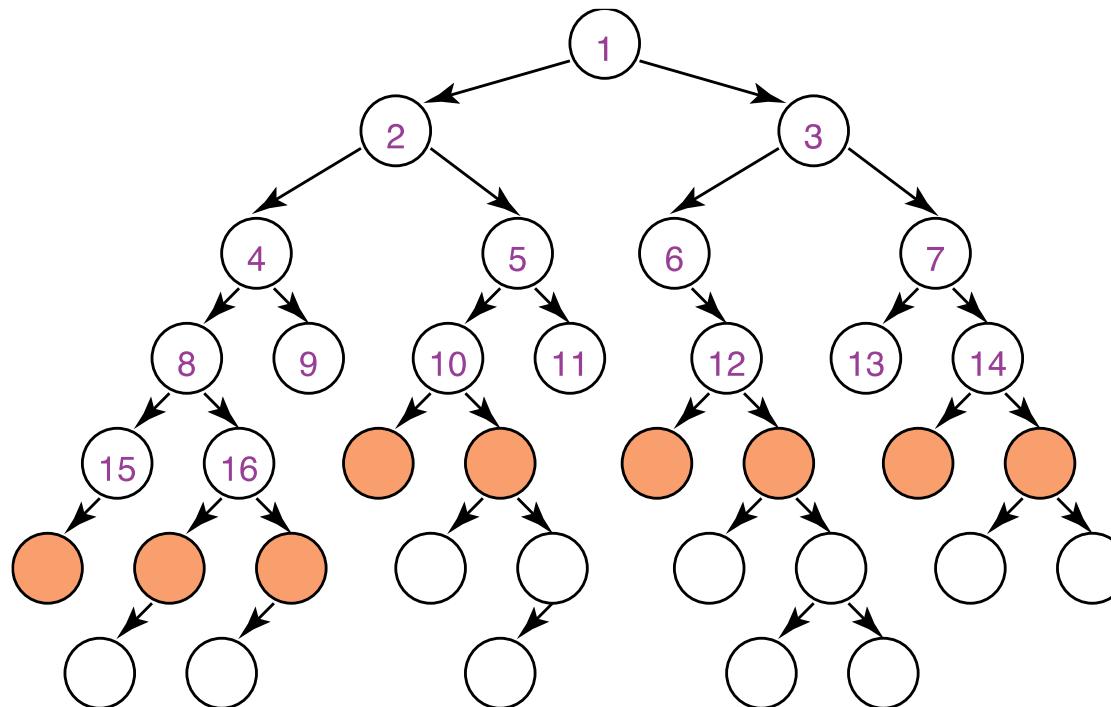
- **Search tree:** superimposed over the state space.
- **Root:** search node corresponding to the initial state.
- **Leaf nodes:** correspond to states that have no successors in the tree because they were not expanded or generated no new nodes.



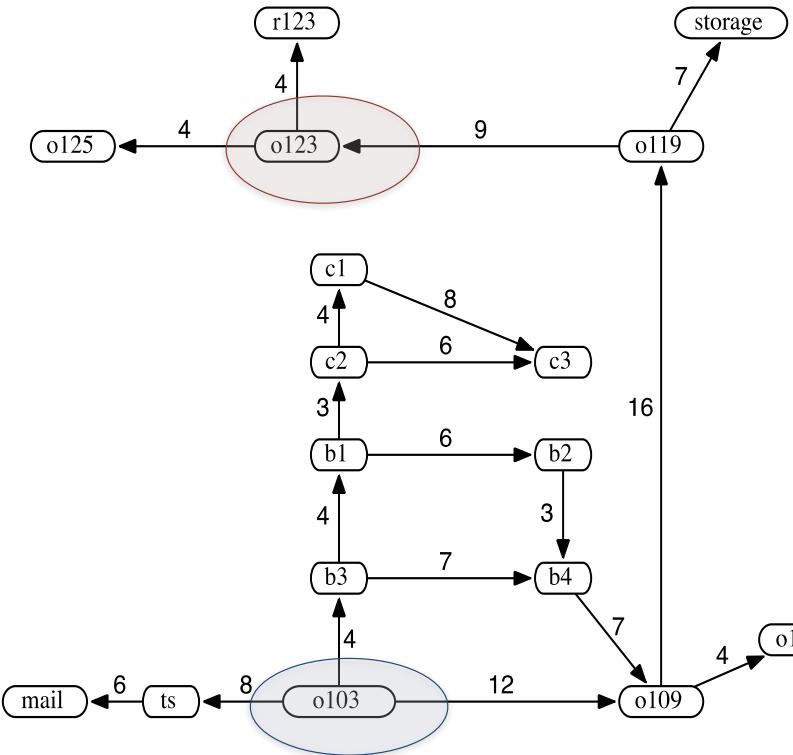
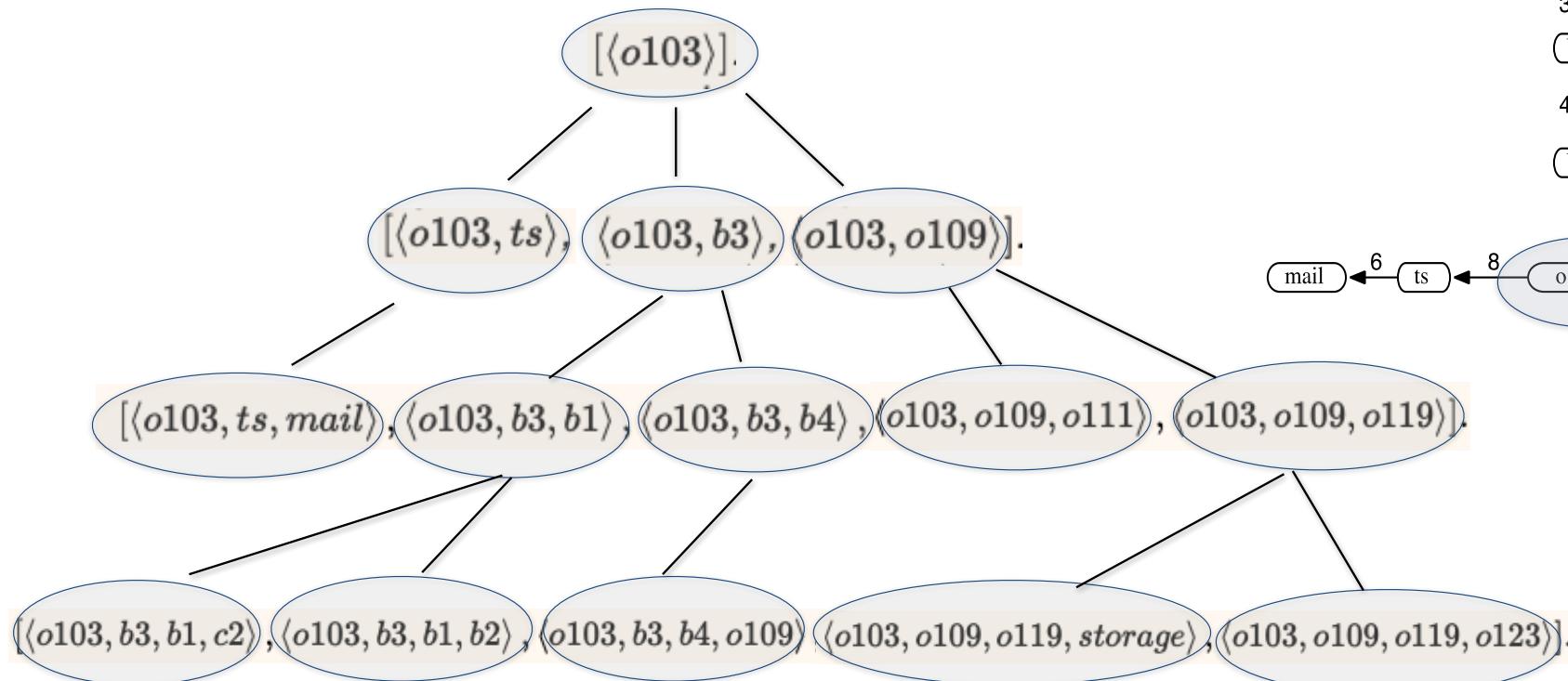
Breadth-First Search



Breadth-first Search Frontier



Breadth-First Search



After each iteration, each path on the frontier has either the same number of arcs

Breadth-first Search

- Breadth-first search treats the frontier as a queue
- It selects the first element in the queue to explore next
- If the list of paths on the frontier is $[p_1, p_2, \dots, p_r]$:
 - p_1 is selected. Its neighbours are added to the end of the queue, after p_r .
 - p_2 is selected next.

Breadth-First Search

- All nodes are expanded at a same depth in the tree before any nodes at the next level are expanded
- Can be implemented by using a queue to store frontier nodes
 - put newly generated successors at end of queue
- Stop when node with goal state is reached
- Include check that state has not already been explored
 - Needs a new data structure for set of explored states
- Finds the shallowest goal first

Complexity of Breadth-first Search

- Does breadth-first search guarantee finding the shortest path?
- What happens on infinite graphs or on graphs with cycles if there is a solution?
- What is the time complexity as a function of the length of the path selected?
- What is the space complexity as a function of the length of the path selected?
- How does the goal affect the search?

Properties of breadth-first search

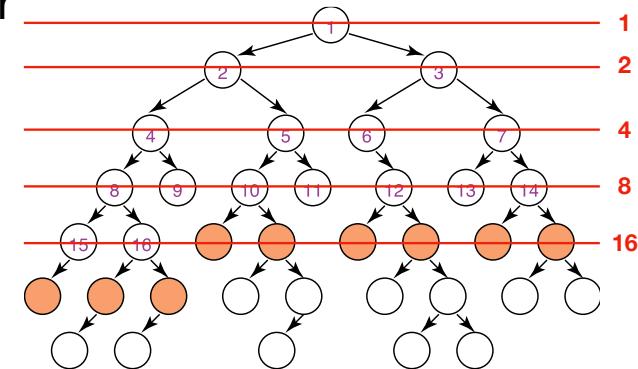
Complete? Yes (if breadth, b , is finite, the shallowest goal is at a fixed depth, d , and will be found before any deeper nodes are generated)

Time? $1 + b^2 + b^3 + \dots + b^d = \frac{b^{d+1} - 1}{b - 1} = O(b^d)$

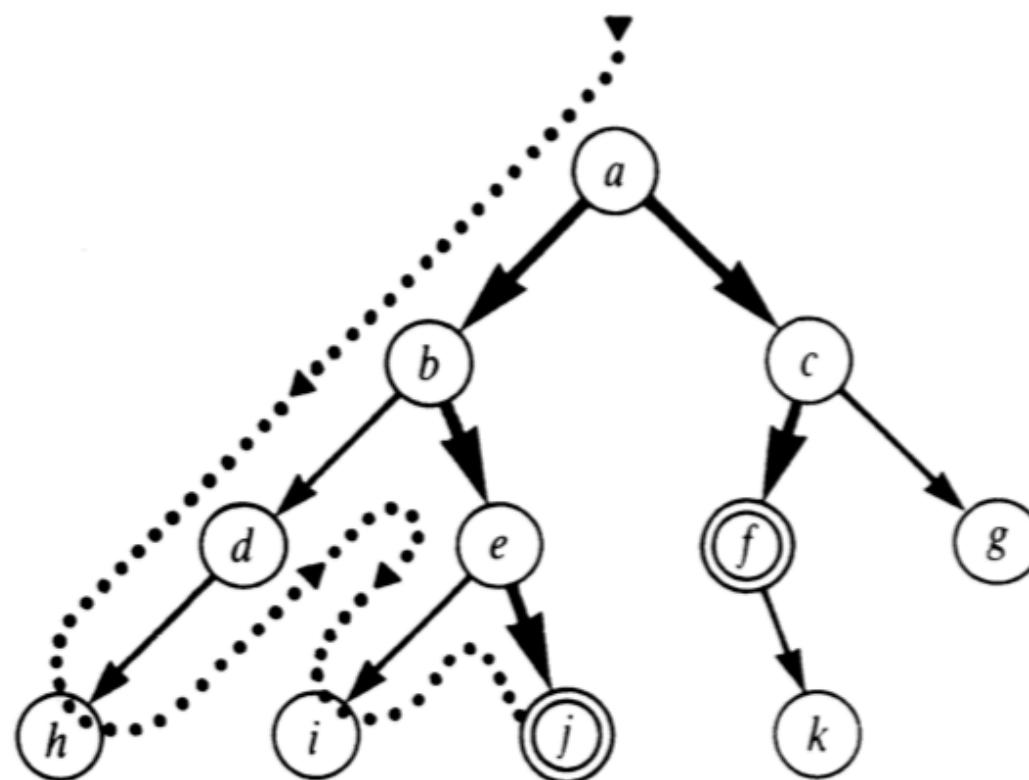
Space? $O(b^d)$ (keeps every node in memory; generate all nodes up to level d)

Optimal? Yes, but only if all actions have the same cost

Space is the big problem for BFS. It grows **exponentially** with depth



Depth-first Search - DFS

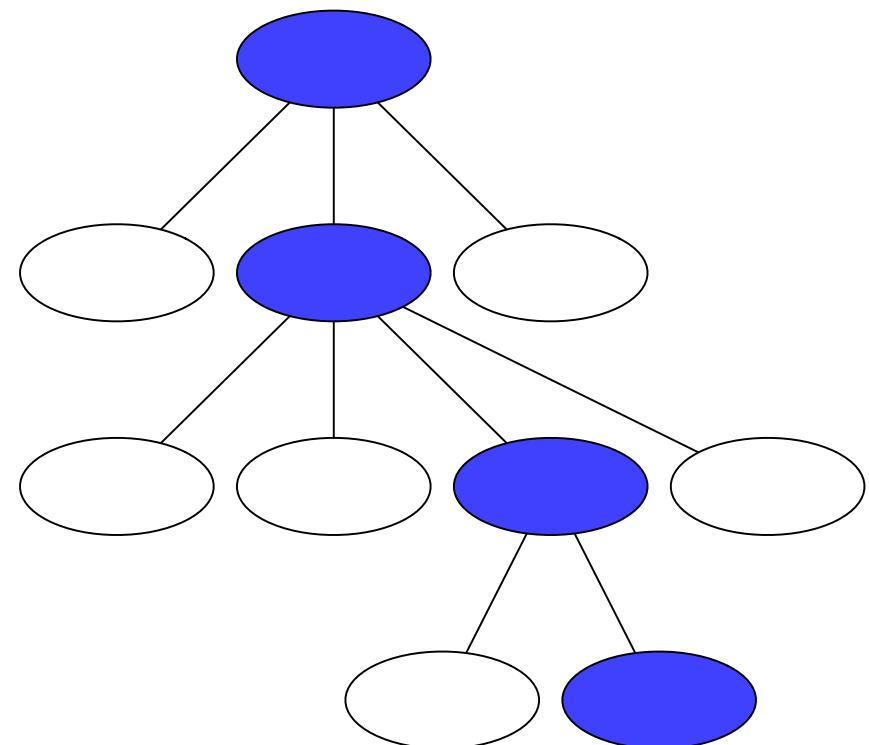


Depth First Search

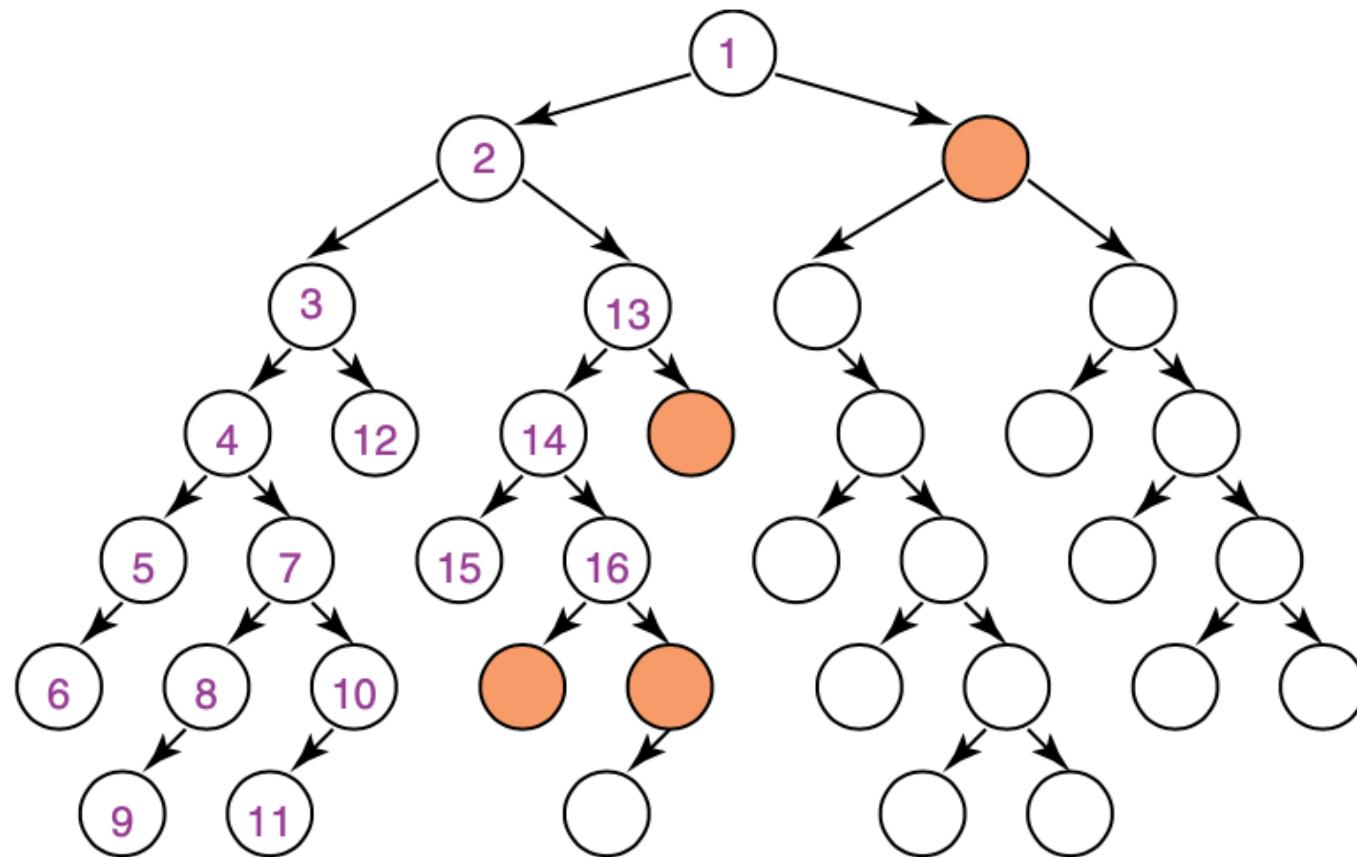
- Expand one node at the deepest level reached so far
- Implementation:
 - Implement the frontier as a stack, i.e. insert newly generated states at the front of the open list (frontier)
 - Can be implemented by recursive function calls, where call stack maintains open list
- In depth-first search, like breadth-first, the order in which the paths are expanded does not depend on the goal.

Depth First Search

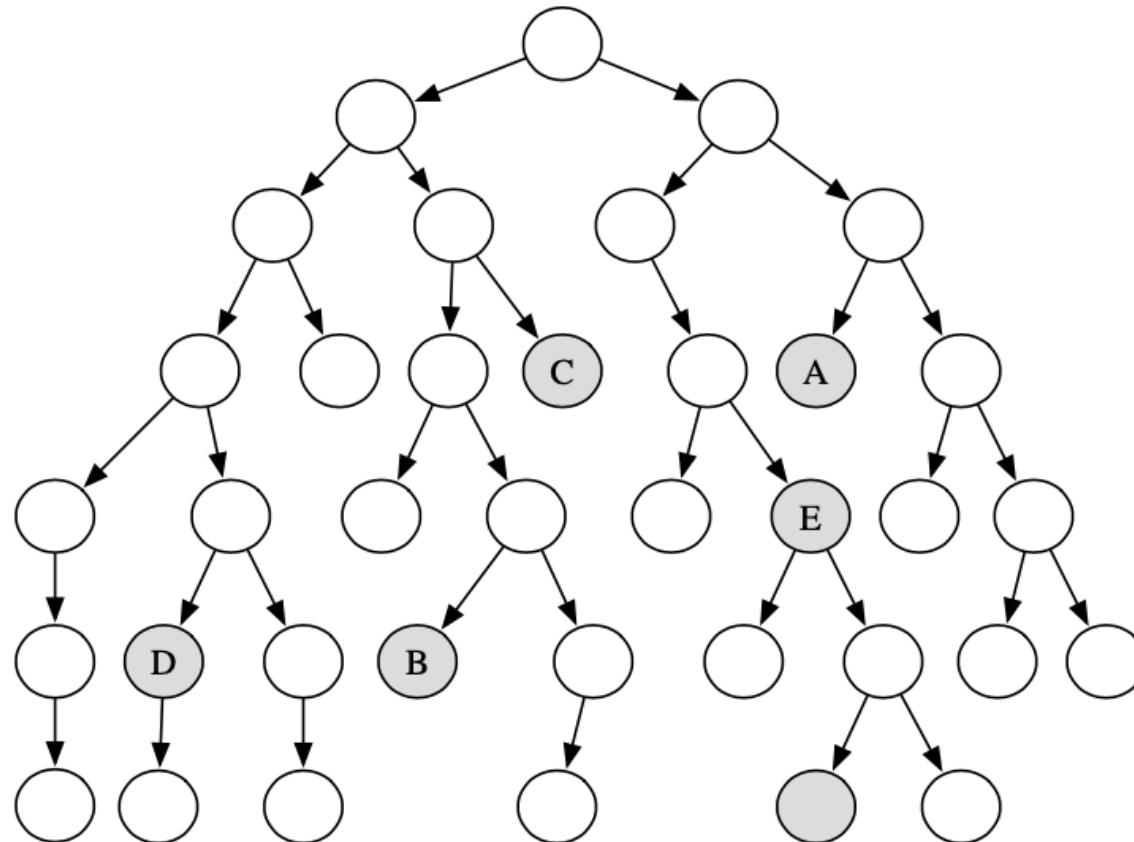
- At any point depth-first search stores single path from root to leaf, together with any remaining unexpanded siblings of nodes along path
- Stop when node with goal state is expanded
- Include check that state has not already been explored along a path – cycle checking



Depth-first Search Example



Which goal (shaded) will depth-first search find first?



Properties of depth-first search

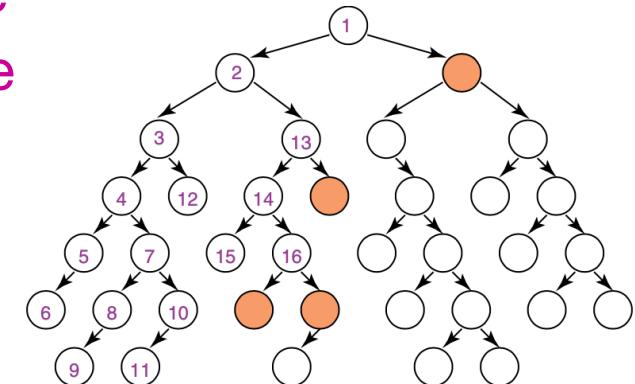
Complete? No! fails in infinite-depth spaces, spaces with loops

Modify to avoid repeated states along path → complete in finite spaces

Time? $O(b^m)$, m = maximum depth of search tree **terrible** if m is much larger than d , but if solutions are dense, may be much faster than breadth-first

Space? $O(bm)$, i.e., linear space

Optimal? No, can find suboptimal solutions first.



Depth-First Search Analysis

- In cases where problem has many solutions, depth-first search may outperform breadth-first search because there is a good chance it will find a solution after exploring only a small part of the space
- However, depth-first search may get stuck following a deep or infinite path even when a solution exists at a relatively shallow level
- Therefore, depth-first search is not complete and not optimal
 - Avoid depth-first search for problems with deep or infinite path

Lowest-cost-first Search

Uniform-Cost Search

- Sometimes transitions have a cost
- Cost of a path is the sum of the costs of its arcs:

$$cost(\langle n_0, \dots, n_k \rangle) = \sum_{i=1}^k cost(\langle n_{i-1}, n_i \rangle)$$

- An optimal solution has minimum cost
- **Delivery robot example:**
 - cost of arc may be resources (e.g., time, energy) required to execute action represented by the arc
 - aim is to reach goal using least resources

Lowest-cost-first Search

Uniform-Cost Search

- The simplest search method that is guaranteed to find a minimum cost path is **lowest-cost-first** search or **uniform-cost search**
 - similar to breadth-first search, but instead of expanding path with least number of arcs, select path with lowest cost
 - implemented by treating the frontier as a priority queue ordered by the cost function

$$cost(\langle n_0, \dots, n_k \rangle) = \sum_{i=1}^k cost(\langle n_{i-1}, n_i \rangle)$$

Lowest-Cost Search for Delivery Robot

- Edges are labelled with cost
 - e.g. distance to travel
- Sort queue by increasing cost of path to the node

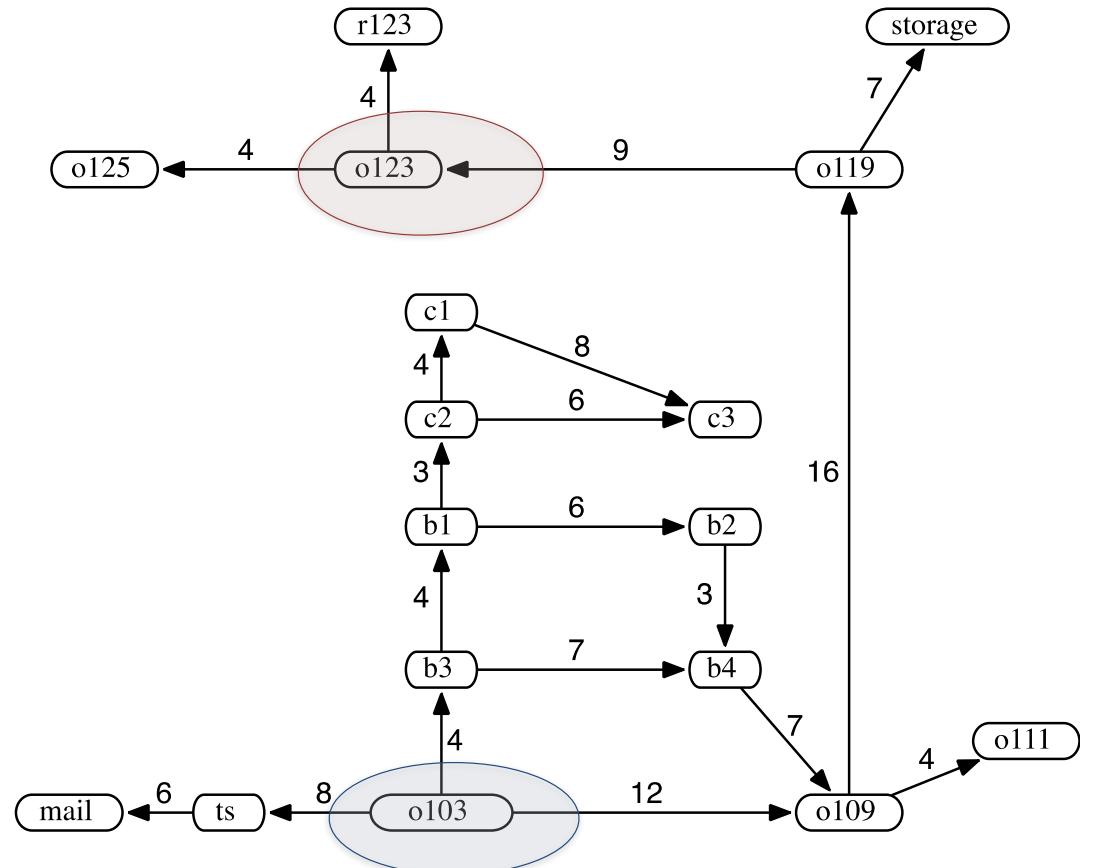
: $[o103_0]$

$[b3_4, ts_8, o109_{12}]$

$[b1_8, ts_8, b4_{11}, o109_{12}]$

$[ts_8, c2_{11}, b4_{11}, o109_{12}, b2_{14}]$

$[c2_{11}, b4_{11}, o109_{12}, mail_{14}, b2_{14}]$



Uniform-Cost Search

- Expand root first, then expand least-cost unexpanded node
- Implementation with priority queue
 - insert nodes in order of increasing path cost - lowest path cost is $g(n)$.
- Reduces to breadth-first search when all actions have same cost
- Finds the cheapest goal provided path cost is monotonically increasing along each path (i.e. no negative-cost steps)

Properties of Uniform-Cost Search

Complete? Yes, if b is finite and if transition $cost \geq \epsilon$ with $\epsilon > 0$

Time? Worst case, $O(b^{[C^*/\epsilon]})$ where C^* = cost of the optimal solution
every transition costs at least ϵ
 \therefore cost per step is $\frac{C^*}{\epsilon}$

Space? $O(b^{[C^*/\epsilon]})$, $b^{[C^*/\epsilon]} = b^d$ if all step costs are equal

Optimal? Yes – nodes expanded in increasing order of $g(n)$

Summary of Search Strategies

Strategy	Frontier Selection	Complete	Halts	Space
Depth-first	Last node added	No	No	Linear
Breadth-first	First node added	Yes	No	Exp
Lowest-cost-first	Minimal $cost(p)$	Yes	No	Exp

Complete: guaranteed to find a solution if there is one (for graphs with finite number of neighbours, even on infinite graphs)

Halts: on finite graph (perhaps with cycles).

Space: as a function of the length of current path

Depth Bounded Search

Expands nodes like Depth First Search but imposes a cutoff on the maximum depth of path.

Complete? Yes (no infinite loops anymore)

Time? $O(b^k)$ where k is the depth limit

Space? $O(bk)$, i.e., linear space similar to DFS

Optimal? No, can find suboptimal solutions first.

Problem: How to pick a good limit ?

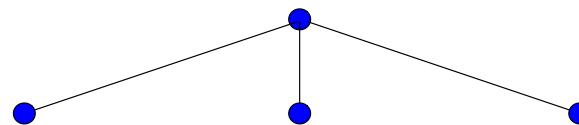
Iterative Deepening Search

- Depth-bounded search: hard to decide on a depth bound
- Iterative deepening: Try all possible depth bounds in turn
- Combines benefits of depth-first and breadth-first search

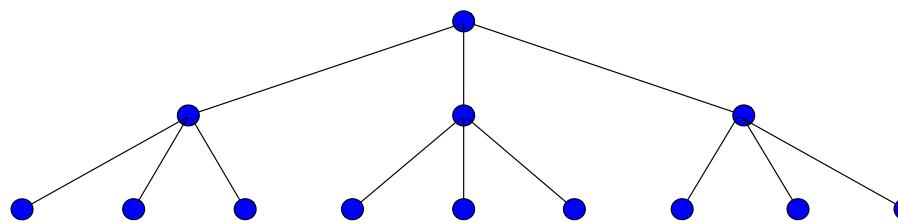
Iterative Deepening Search

- Tries to combine the benefits of depth-first (low memory) and breadth-first (optimal and complete)
- Does a series of depth-limited depth-first searches to depth 1, 2, 3, etc.
- Early states will be expanded multiple times, but that might not matter too much because most of the nodes are near the leaves.

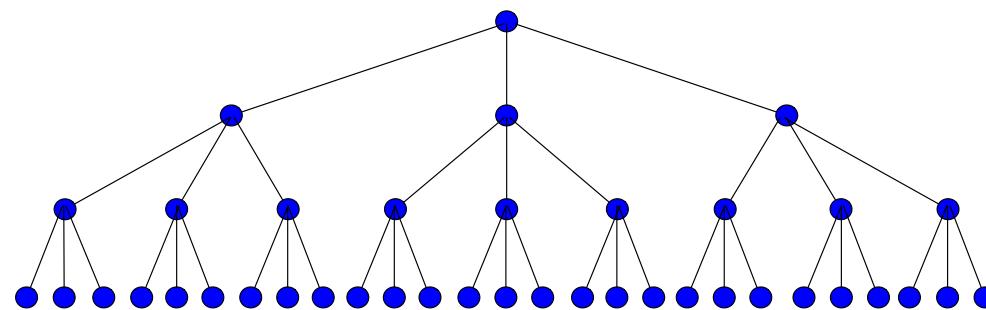
Iterative Deepening Search



Iterative Deepening Search



Iterative Deepening Search



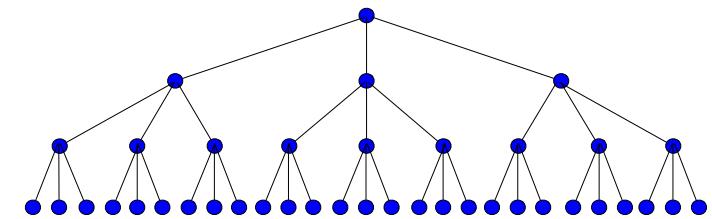
Properties of Iterative Deepening Search

- Complete? Yes.
- Time: nodes at the bottom level are expanded once, nodes at the next level up twice, and so on:

- depth-bounded: $1 + b^2 + b^3 + \dots + b^d = \frac{b^{d+1} - 1}{b - 1} = O(b^d)$
- Iterative deepening:

$$(d + 1)b^0 + db^1 + (d - 1)b^2 + \dots + 2 \cdot b^{d-1} + 1 \cdot b^d = O(b^d)$$

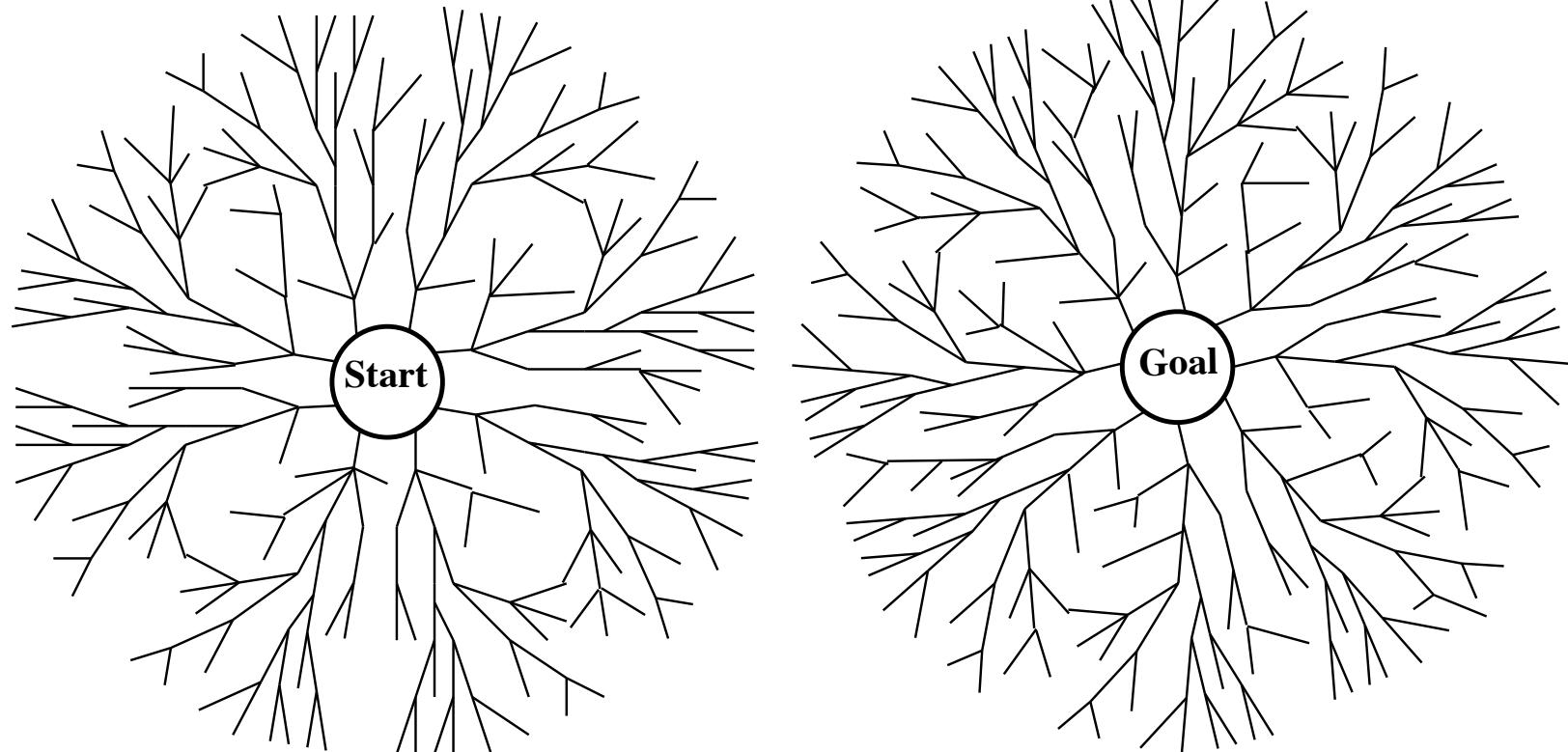
- Example b=10, d=5:
 - depth-bounded: $1 + 10 + 100 + 1,000 + 10,000 + 100,000 = 111,111$
 - iterative-deepening: $6 + 50 + 400 + 3,000 + 20,000 + 100,000 = 123,456$
 - only about 11% more nodes (for b = 10).



Properties of Iterative Deepening Search

- Complete? Yes.
- Time: $O(b^d)$
- Space? $O(bd)$
- Optimal? Yes, if step costs are identical.
- In general, iterative deepening is the preferred search strategy for a large search space where depth of solution is not known

Bidirectional Search



Bidirectional Search

- Search both forward from the initial state and backward from the goal
 - stop when the two searches meet in the middle.
- Need efficient way to check if a new node appears in the other half of the search.
 - Complexity analysis assumes this can be done in constant time, using a hash table.
- Assume branching factor = b in both directions and that there is a solution at depth = d:
 - Then bidirectional search finds a solution in $O(2b^{d/2}) = O(b^{d/2})$ time steps.

Bidirectional Search Analysis

- If solution exists at depth d then bidirectional search requires time

$$O(2b^{\frac{d}{2}}) = O(b^{\frac{d}{2}})$$

- (assuming constant time checking of intersection)
- To check for intersection must have all states from one of the searches in memory, therefore space complexity is $O(b^{\frac{d}{2}})$

Summary

- Problem formulation usually requires abstracting away real-world details to define a state space that can feasibly be explored.
- Variety of Uninformed search strategies
- Iterative Deepening Search uses only linear space and not much more time than other Uninformed algorithms.

Complexity Results for Uninformed Search

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Time	$O(b^d)$	$O(b^{\lceil C^*/\varepsilon \rceil})$	$O(b^m)$	$O(b^k)$	$O(b^d)$
Space	$O(b^d)$	$O(b^{\lceil C^*/\varepsilon \rceil})$	$O(bm)$	$O(bk)$	$O(bd)$
Complete?	Yes ¹	Yes ²	No	No	Yes ¹
Optimal ?	Yes ³	Yes	No	No	Yes ³

b = branching factor, d = depth of the shallowest solution,

m = maximum depth of the search tree, k = depth limit.

1 = complete if b is finite.

2 = complete if b is finite and step costs $\geq \varepsilon$ with $\varepsilon > 0$.

3 = optimal if actions all have the same cost.

Solving Problems by Searching

Informed Search

COMP3411/9814: Artificial Intelligence

Overview

- Heuristics
- Informed Search Methods
 - Best-first search
 - Greedy best-first search
 - A* search
 - Iterative Deepening A* Search

Informed (Heuristic) Search

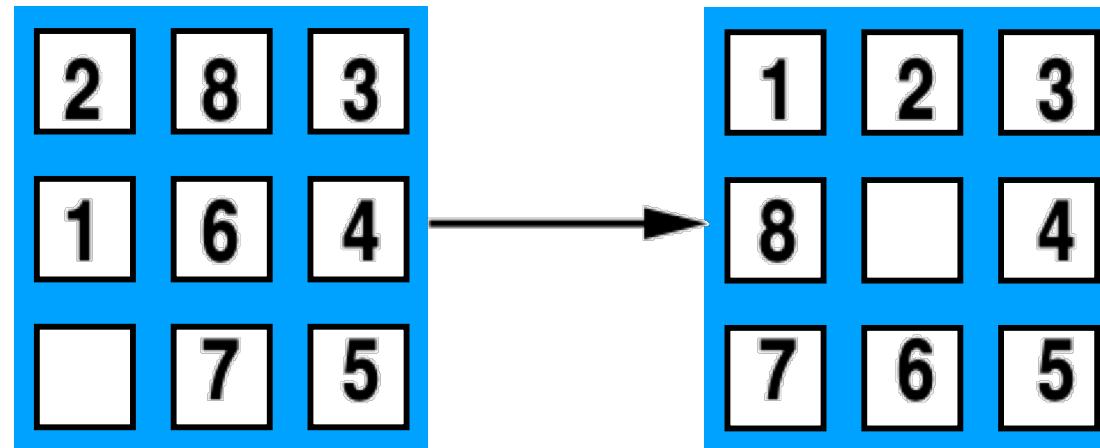
- Informed search strategy
 - use problem-specific knowledge
 - more efficiently than uninformed search
- Uninformed search algorithms have no information about problem other than its definition.
 - some can solve any solvable problem, none of them can do it efficiently
- Informed search algorithms can do well given guidance on where to look for solutions.
- Implemented using a **priority queue** to store frontier nodes

Heuristics

- Heuristics are “rules of thumb” for deciding which alternative is best
- Heuristic must **underestimate** actual cost to get from current node to goal
 - Called an **admissible heuristic**
 - Denoted **$h(n)$**
 - $h(n) = 0$ whenever n is a **goal** node

Heuristics – Example

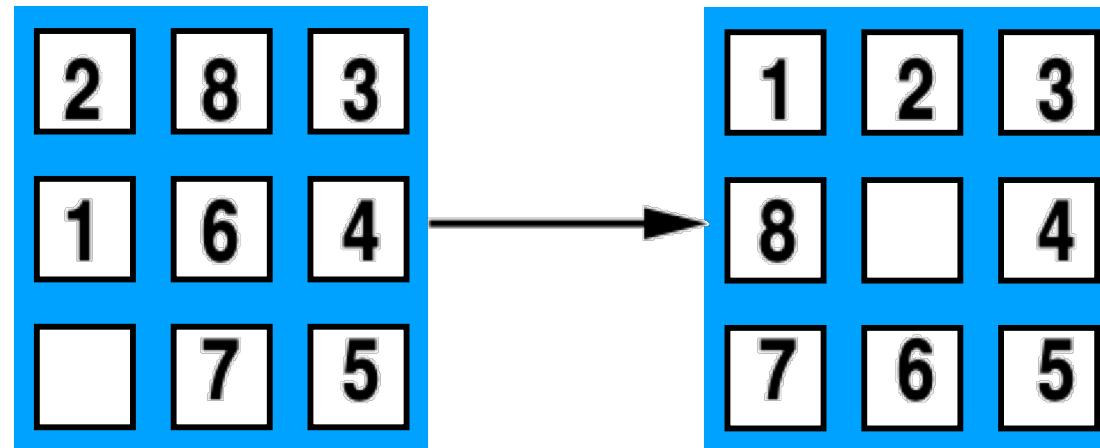
8-Puzzle – number of tiles out of place



$$h(n) = 5$$

Heuristics – Example

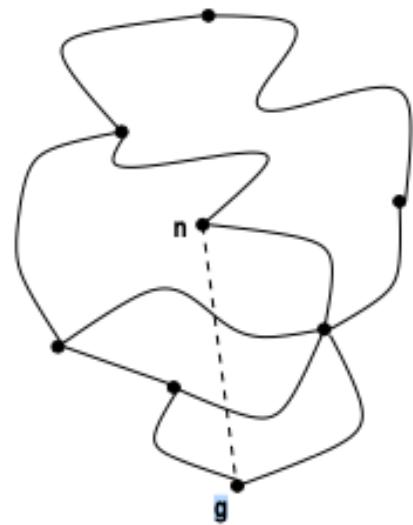
8-Puzzle – Manhattan distance (distance tile is out of place)



$$h(n) = 1 + 1 + 0 + 0 + 0 + 1 + 1 + 2 = 6$$

Heuristics – Example

Another common heuristic is the straight-line distance (“as the crow flies”) from node to goal



Therefore $h(n) = \text{distance from } n \text{ to } g$

Uninformed vs Informed Search

- Uninformed - keeps search until it stumbles on goal
 - No domain knowledge
- Informed - searches in direction of best guess to goal
 - Uses domain knowledge

Heuristic Search

- $h(n)$ estimates the cost of shortest path from node, n , to a goal node.
- $h(n)$ must be efficient to compute.
- h can be extended to paths: $h(\langle n_0, \dots, n_k \rangle) = h(n_k)$.
- $h(n)$ must be an **underestimate**
 - i.e. there is no path from n to a goal with cost less than $h(n)$.
 - An **admissible heuristic** is a non-negative function that is an **underestimate** of the actual cost of a path to a goal.

Example Heuristic Functions

- If nodes are points on a Euclidean plane and cost is distance, $h(n)$ can be straight-line distance from n to closest goal.
- If nodes are locations and cost is time, can use distance to goal divided by maximum speed.
 - If goal is to collect a bunch of coins and not run out of fuel, cost is an estimate of how many steps to collect rest of the coins, refuel when necessary, and return to goal.
- Heuristic function can be found by simplifying calculation of true cost

Search Strategies

General Search algorithm:

- add initial state to queue
- repeat:
 - take node from front of queue
 - test if it is a goal state; if so, terminate
 - “expand” it, i.e. generate successor nodes and add them to the queue

Search strategies are distinguished by the order in which new nodes are added to the queue of nodes awaiting expansion.

Search Strategies

- **BFS** and **DFS** treat all new nodes the same way:
 - **BFS** add all new nodes to the back of the queue
 - **DFS** add all new nodes to the front of the queue
- **Best First Search** uses an evaluation function $f()$ to order the nodes in the queue
 - Similar to uniform cost search
- **Informed or Heuristic:**
 - Greedy Search $f(n) = h(n)$ (estimates cost from node n to goal)
 - A* Search $f(n) = g(n) + h(n)$ (cost from start to n plus estimated cost to goal)

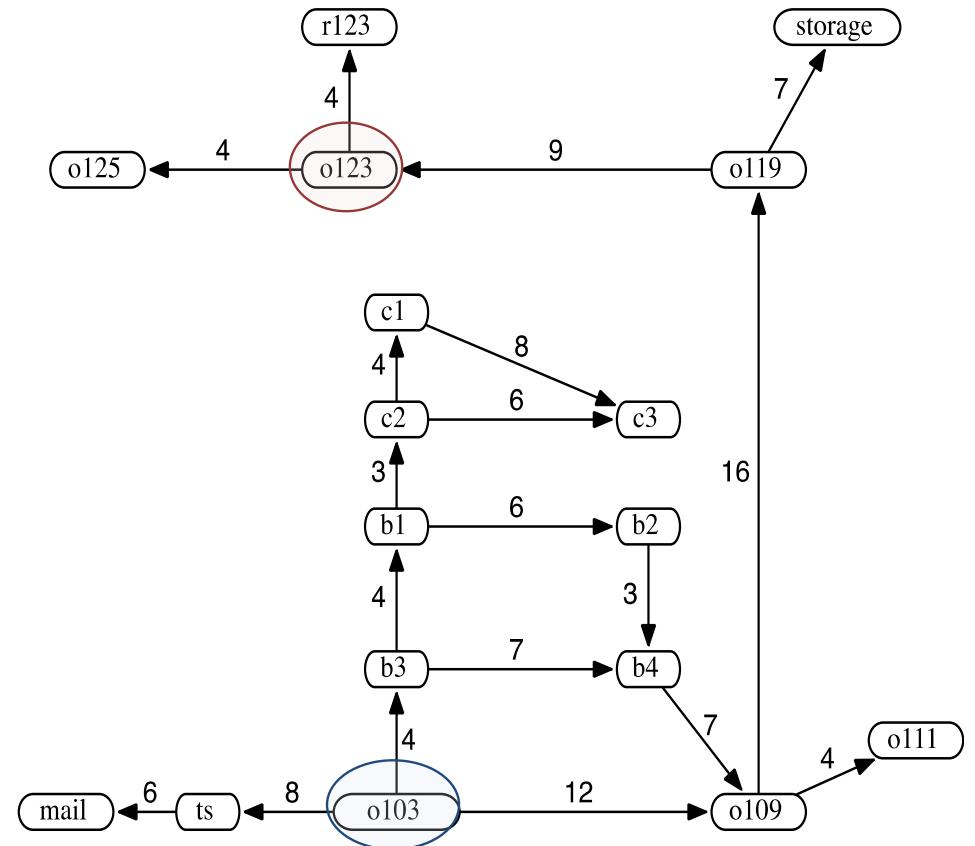
Delivery Robot Heuristic Function

Use straight-line distance as heuristic, and assume these values:

$$\begin{array}{lll} h(\text{mail}) = 26 & h(ts) = 23 & h(o103) = 21 \\ h(o109) = 24 & h(o111) = 27 & h(o119) = 11 \\ h(o123) = 4 & h(o125) = 6 & h(r123) = 0 \\ h(b1) = 13 & h(b2) = 15 & h(b3) = 17 \\ h(b4) = 18 & h(c1) = 6 & h(c2) = 10 \\ h(c3) = 12 & h(\text{storage}) = 12 \end{array}$$

$h(loc)$ = distance from loc to goal

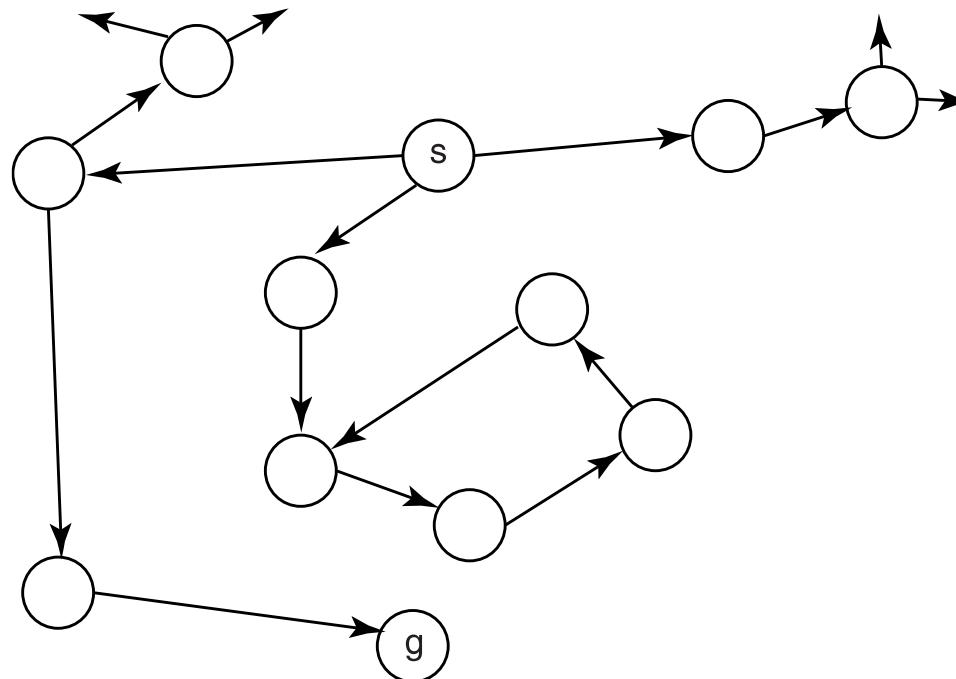
Heuristic function can be extended to paths by making heuristic value of path equal to heuristic value of node at the end of the path: $h(\langle n_0, \dots, n_k \rangle) = h(n_k)$.



Greedy Best-First Search

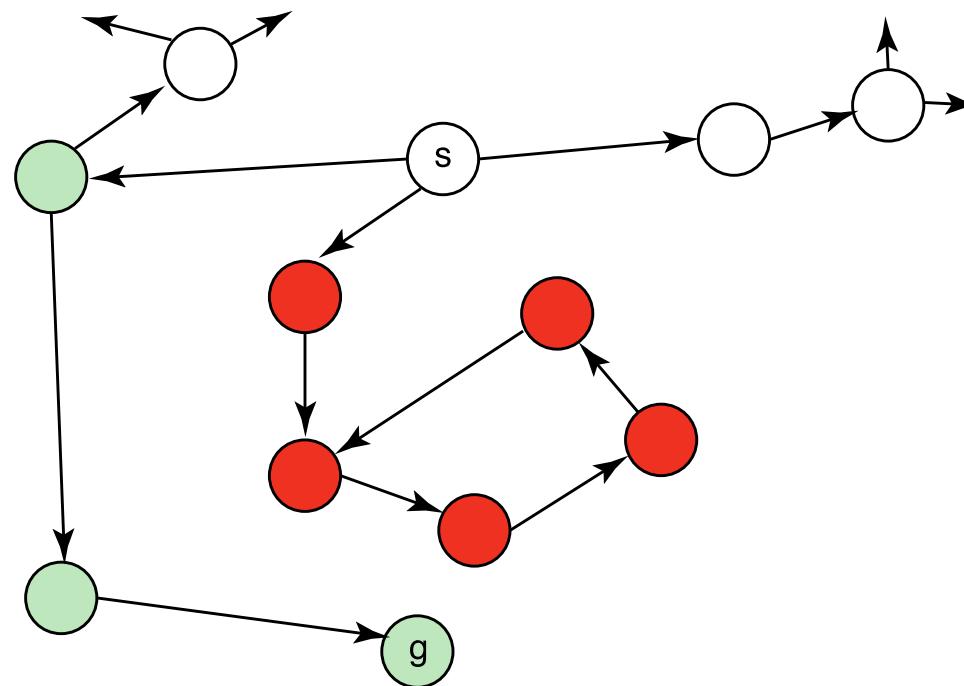
- Always select node closest to goal according to heuristic function
- $h(n)$ is estimated cost to goal
 - $h(n) = 0$ if n is a goal state
- Frontier is a priority queue ordered by h .
- “Greedy” algorithm takes “best” node first.
- Like depth-first search, except pick next node by $h(n)$

Greedy Best-first Search Example



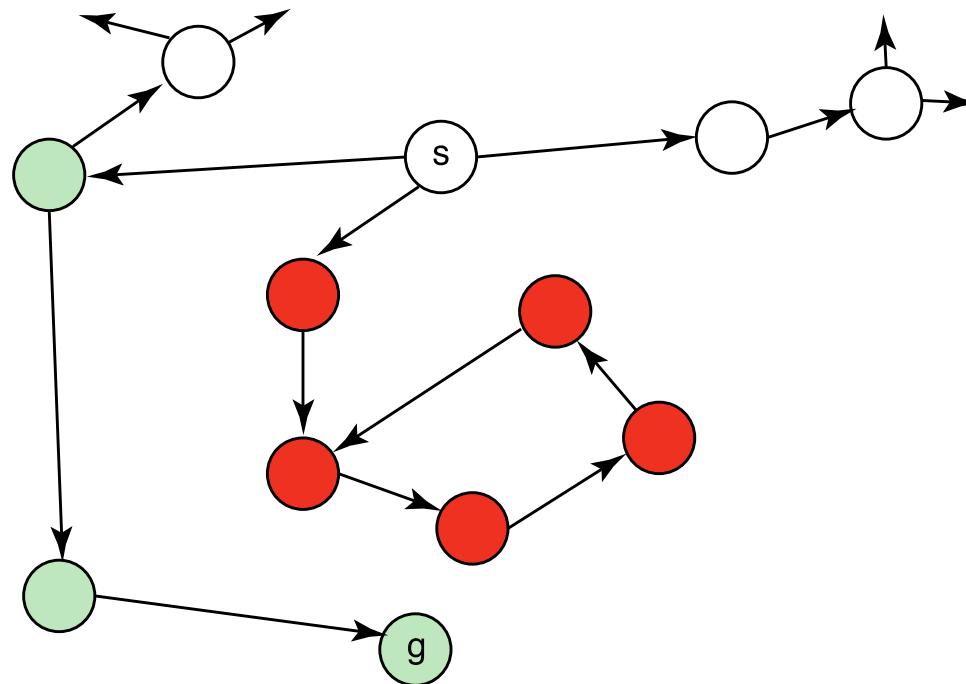
- Graph drawn to scale – cost of arc is its length
- Aim is to find shortest path from **s** to **g**.

Greedy Best-first Search Example



Straight-line distance to goal g is used as heuristic function.

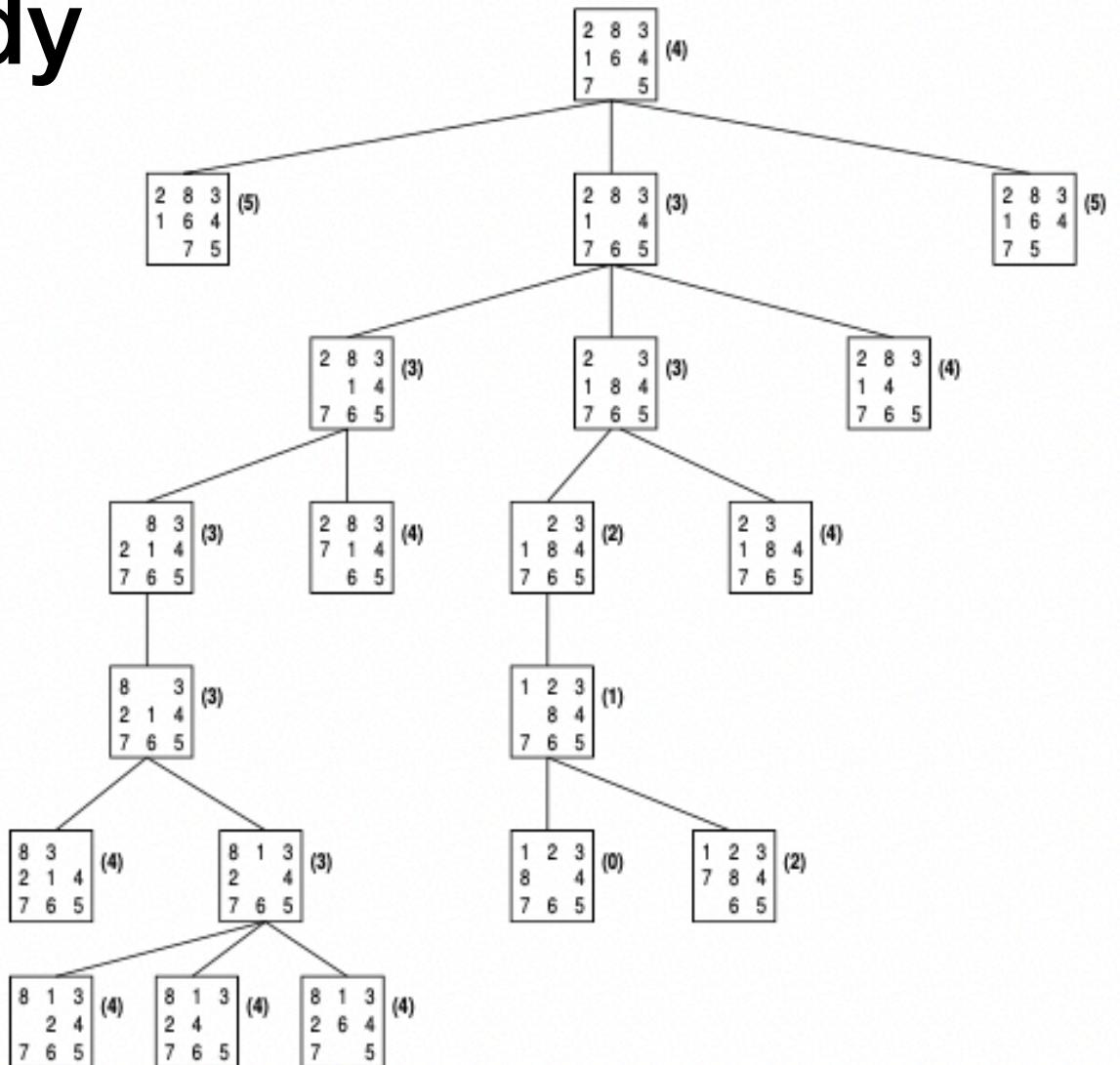
Greedy Best-first Search Example



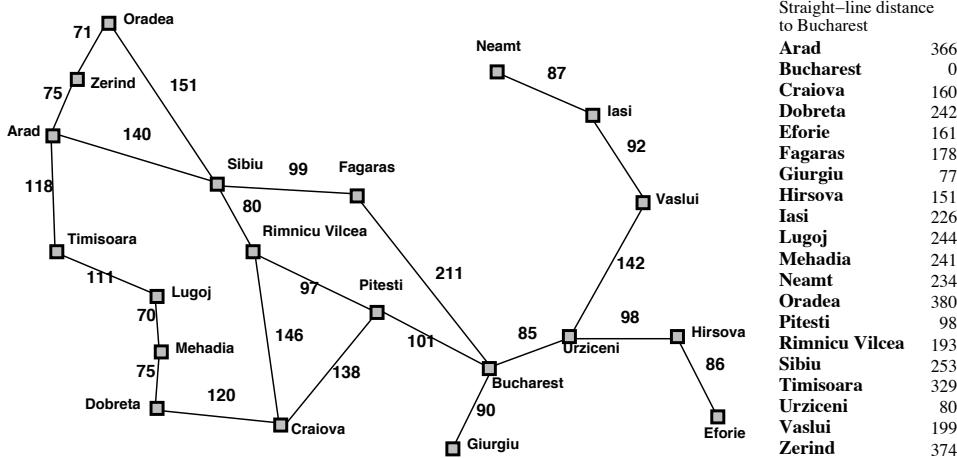
- Greedy depth-first search selects closest node to s and never terminates
- All nodes below s look good. Greedy best-first search cycles between them, never trying alternate route.

Examples of Greedy Best-First Search

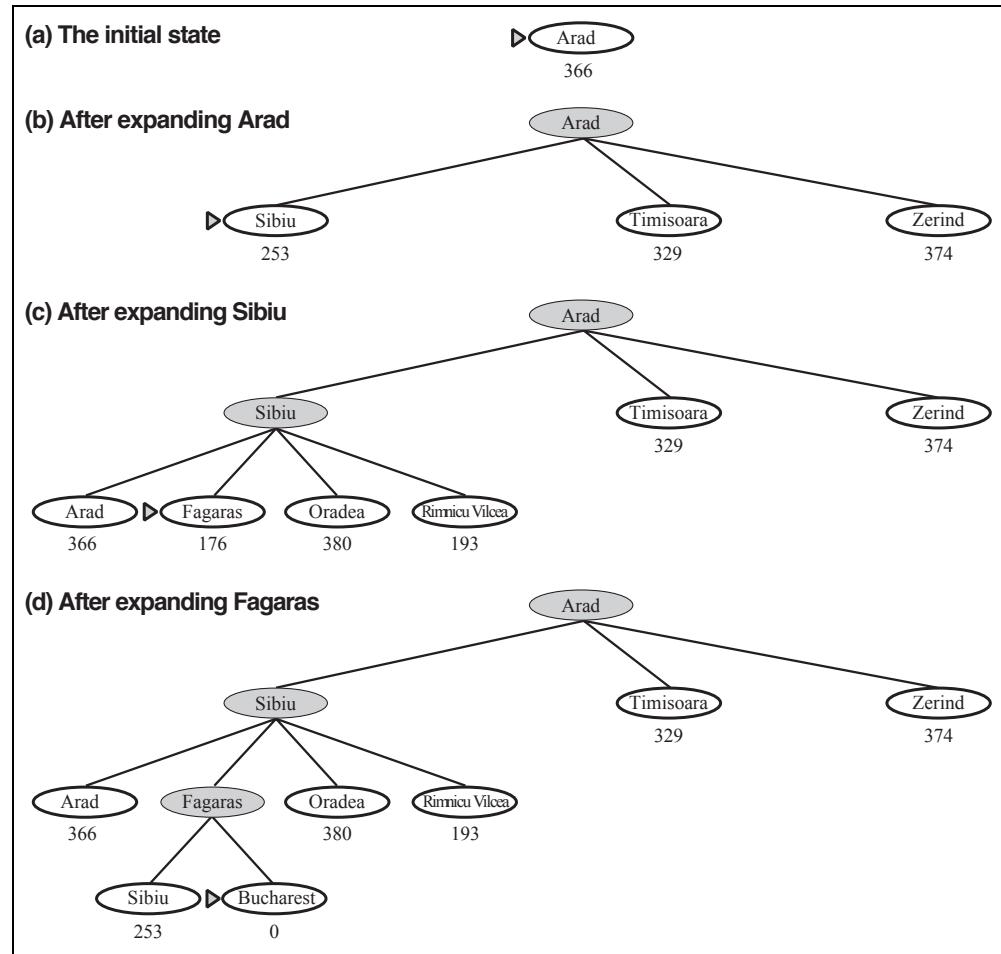
h is number of tiles out of place



Examples of Greedy Best-First Search



- Stages in a greedy best-first tree search for route from Arad to Bucharest with the straight-line distance heuristic.
- Note that **straight-line distances are less than actual distances** in map.



Properties of Greedy Best-First Search

Complete: No. Can get stuck in loops.
(Complete in finite space with repeated-state checking)

Time: $O(b^m)$, where m is the maximum depth in search space.

Space: $O(bm)$ (retains all nodes in memory)

Optimal: No.

- Greedy Search has the same deficits as Depth-First Search.
- However, a good heuristic can reduce time and memory costs substantially.

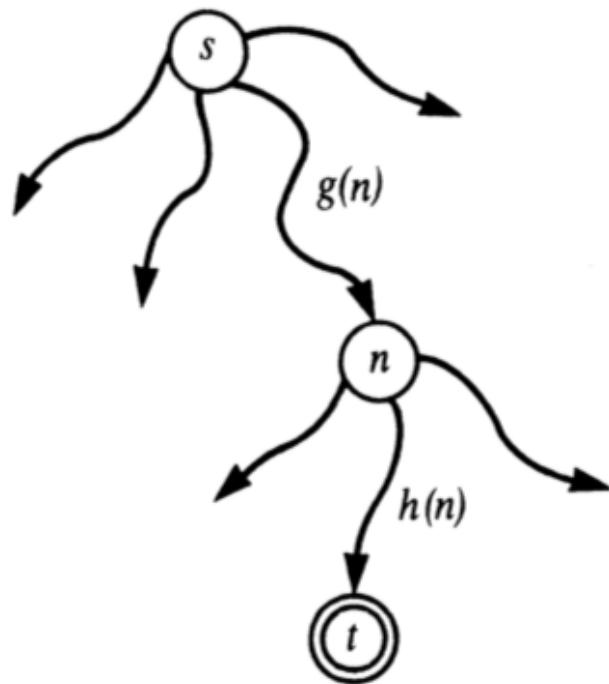
A* Search

- Use both cost of path generated and estimate to goal to order nodes on the frontier
 - $g(n)$ = cost of path from start to n
 - $h(n)$ = estimate from n to goal
- Order priority queue using function $f(n) = g(n) + h(n)$
- $f(n)$ is the estimated cost of the cheapest solution extending this path

A* Search

- Combines uniform-cost search and greedy search
- Greedy Search minimises $h(n)$
 - efficient but not optimal or complete
- Uniform Cost Search minimises $g(n)$
 - optimal and complete but not efficient

Heuristic Function

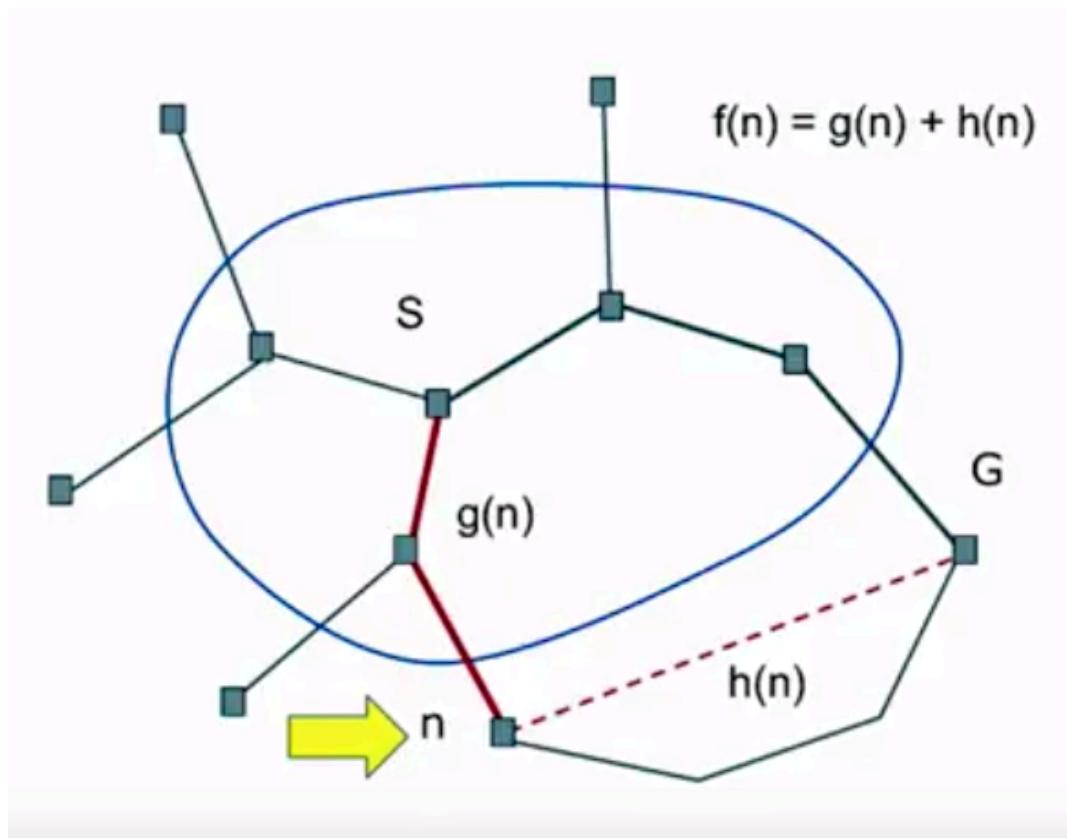


Heuristic estimate $f(n) = \text{cost of the cheapest path from } s \text{ to } t \text{ via } n: f(n) = g(n) + h(n)$

$g(n)$ is the cost the path from s to n

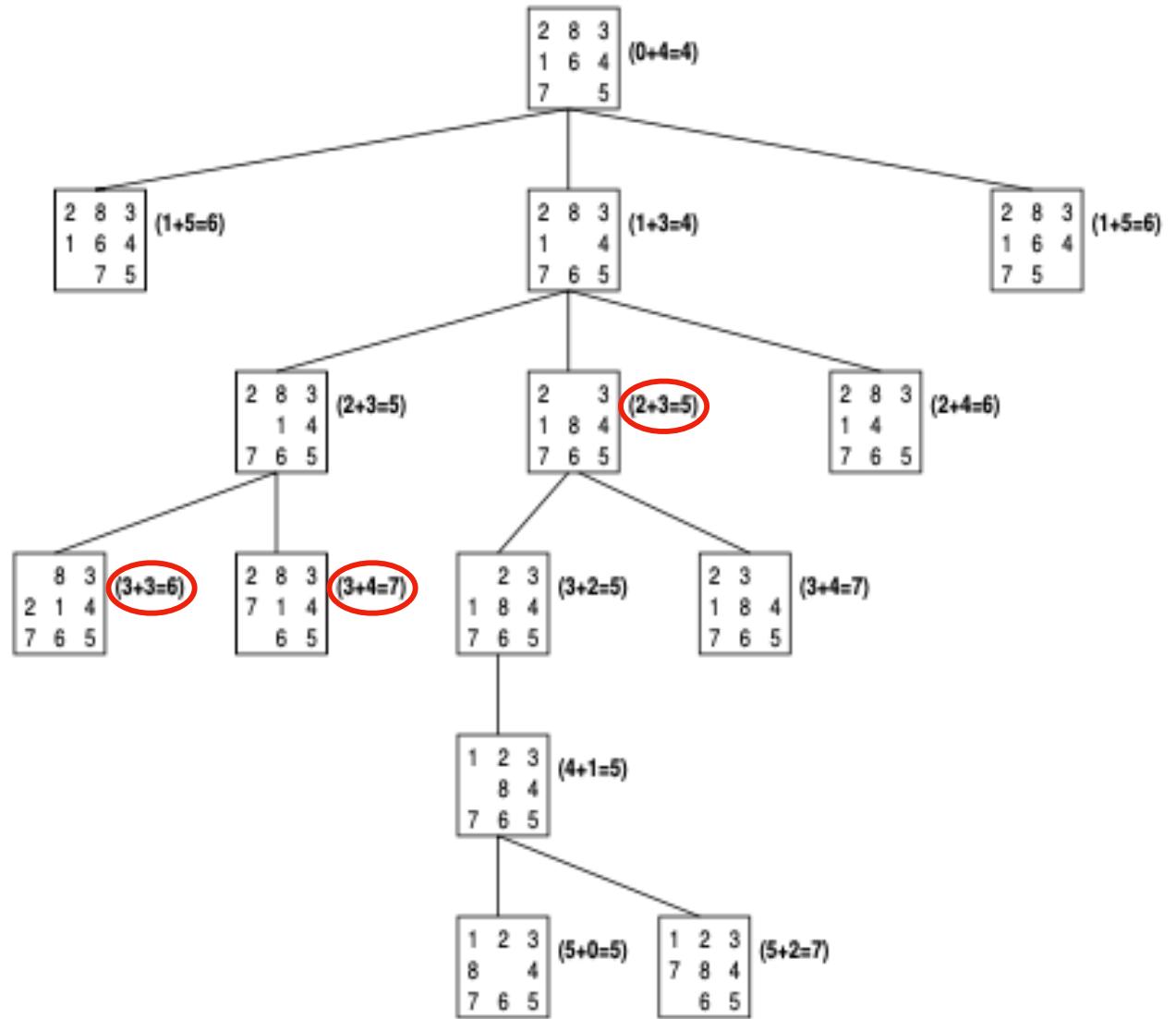
$h(n)$ is an estimate of the cost of an optimal path from n to t .

A* Search



- **S** = start
- **G** = Goal
- **n** = current node
- $g(n)$ = actual cost from **S** to **n**
- $h(n)$ = estimated distance from **n** to **G**

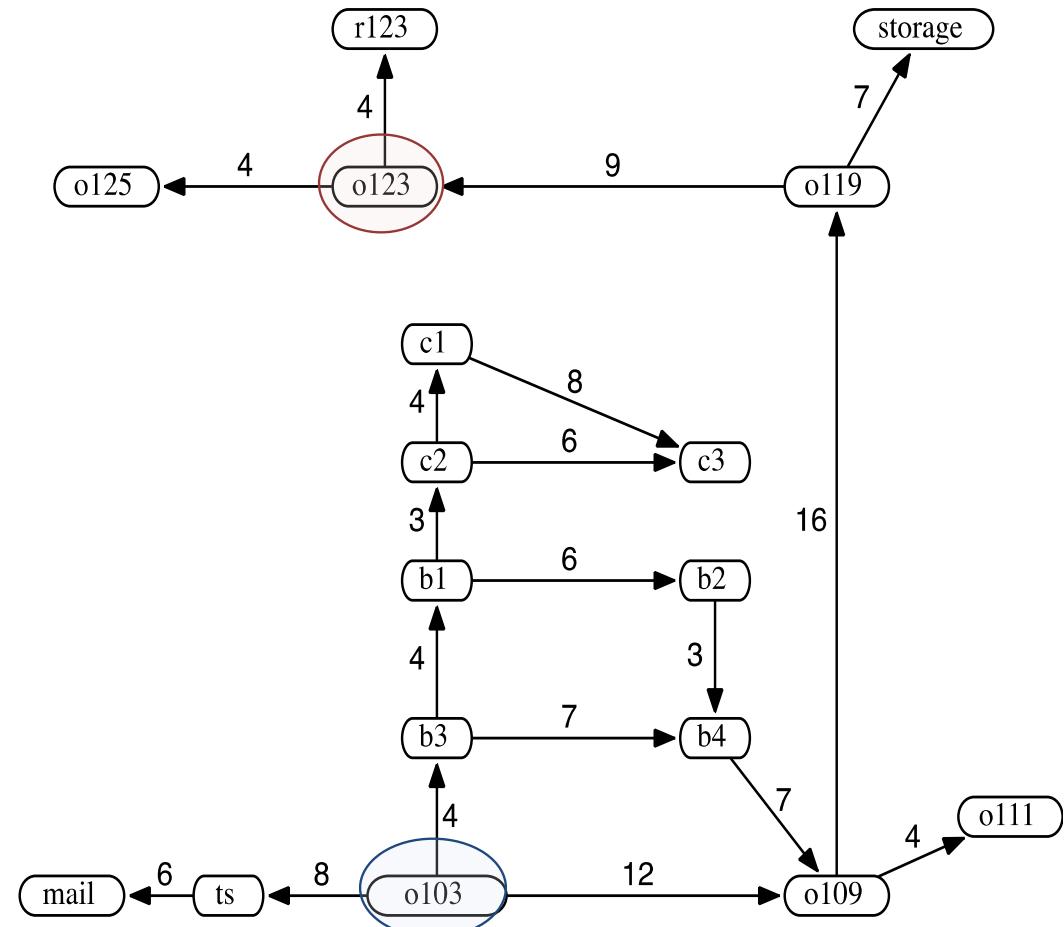
A* Search



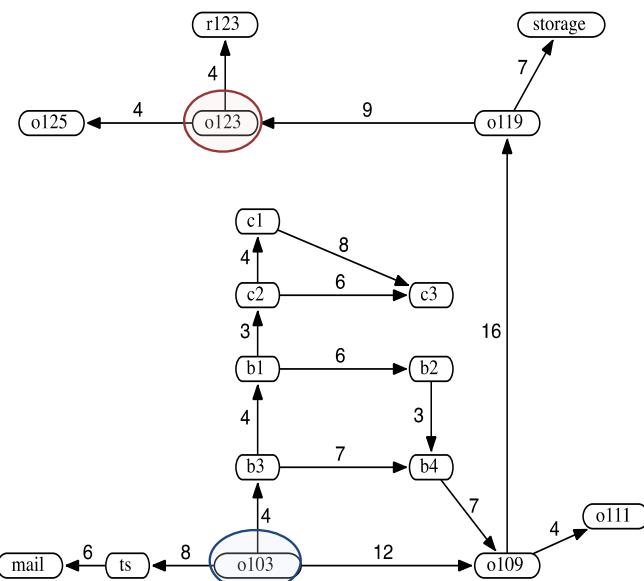
Delivery Robot Heuristic Function

Use straight-line distance as heuristic, and assume these values:

$$\begin{array}{lll}
 h(\text{mail}) = 26 & h(ts) = 23 & h(o103) = 21 \\
 h(o109) = 24 & h(o111) = 27 & h(o119) = 11 \\
 h(o123) = 4 & h(o125) = 6 & h(r123) = 0 \\
 h(b1) = 13 & h(b2) = 15 & h(b3) = 17 \\
 h(b4) = 18 & h(c1) = 6 & h(c2) = 10 \\
 h(c3) = 12 & h(\text{storage}) = 12
 \end{array}$$



A* Search - The Delivery Robot



$h(\text{mail}) = 26$ $h(\text{ts}) = 23$ $h(o103) = 21$
 $h(o109) = 24$ $h(o111) = 27$ $h(o119) = 11$
 $h(o123) = 4$ $h(o125) = 6$ $h(r123) = 0$
 $h(b1) = 13$ $h(b2) = 15$ $h(b3) = 17$
 $h(b4) = 18$ $h(c1) = 6$ $h(c2) = 10$
 $h(c3) = 12$ $h(\text{storage}) = 12$

1. **[o103₂₁]** $h(o103) = 21$
2. **[b3₂₁, ts₃₁, o109₃₆]** $f(\langle o103, b3 \rangle) = g(\langle o103, b3 \rangle) + h(b3) = 4 + 17 = 21$
3. **[b1₂₁, b4₂₉, ts₃₁, o109₃₆]**
4. **[c2₂₁, b2₂₉, b4₂₉, ts₃₁, o109₃₆]**
5. **[c1₂₁, b2₂₉, b4₂₉, c3₂₉, ts₃₁, o109₃₆]**
6. **[b2₂₉, b4₂₉, c3₂₉, ts₃₁, c3₃₅, o109₃₆]**
7. **[b4₂₉, ts₃₁, c3₃₅, o109₃₆]**
8. **[ts₃₁, c3₃₅, b4₃₅, o109₃₆, o109₄₂]**
-

- Lowest-cost path is eventually found.
- Forced to try many different paths, because some temporarily seem to have the lowest cost.
- Still does better than lowest-cost-first search and greedy best-first search.

Optimality of A*

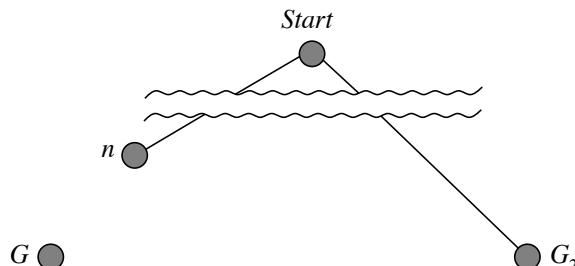
- Heuristic h is said to be **admissible** if

$$\forall n \ h(n) \leq h^*(n) \text{ where } h^*(n) \text{ is the true cost from } n \text{ to goal}$$

- If h is **admissible** then $f(n)$ never overestimates the actual cost of the best solution through n .
 - $f(n) = g(n) + h(n)$, where $g(n)$ is the actual cost to n and $h(n)$ is an underestimate
- Example: $h = \text{straight line distance}$ is admissible because the shortest path between any two points is a line.
- A^* is optimal if h is admissible.
- Admissible heuristics are by nature optimistic because they think the cost of solving the problem is less than it actually is.

Optimality of A* Search

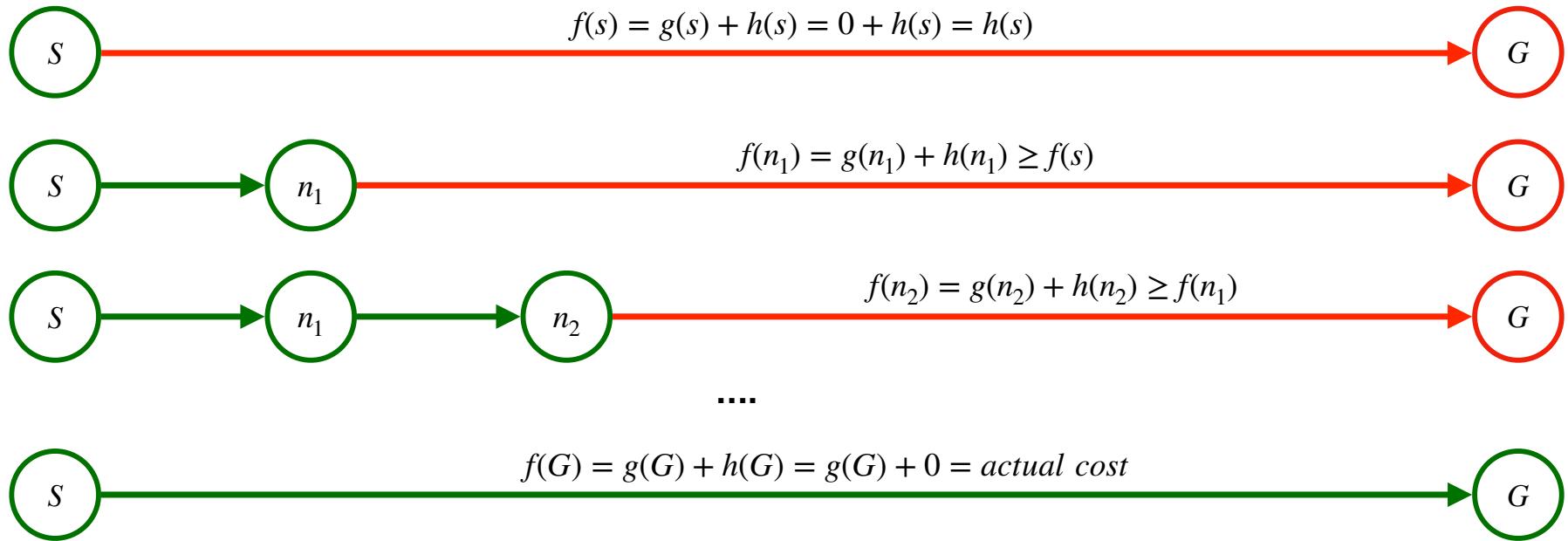
Suppose a sub-optimal goal node G_2 has been generated and is in the queue. Let n be the last unexpanded node on a shortest path to an optimal goal node G .



$$\begin{aligned} f(G_2) &= g(G_2) && \text{since } h(G_2) = 0 \\ &= g(G) && \text{since } G_2 \text{ is sub-optimal} \\ &\geq f(n) && \text{since } h \text{ is admissible} \end{aligned}$$

Hence $f(G_2) > f(n)$, and A* will never select G_2 for expansion because queue is always ordered, e.g. $[\dots, n, \dots, G_2]$.

Consistent Heuristics



If $h(n)$ is an underestimate of the actual cost, $h(n)$, and $f(n)$, can never decrease as search heads towards the goal

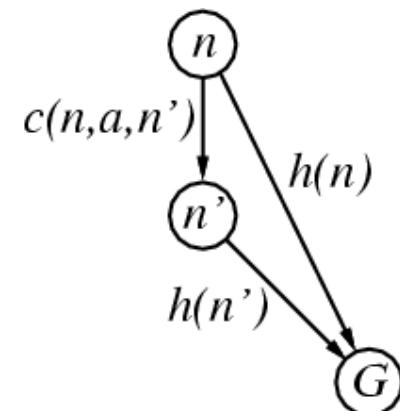
Consistent Heuristics

- A heuristic is consistent if $f(n)$ is nondecreasing along any path
- I.e. for every node n , every successor n' of n generated by any action a ,

$$h(n) \leq c(n, a, n') + h(n')$$

If h is consistent, we have

$$\begin{aligned} f(n') &= g(n') + h(n') \\ &= g(n) + c(n, a, n') + h(n') \\ &\geq g(n) + h(n) \\ &= f(n) \end{aligned}$$



Optimality of A* Search

Complete: Yes, unless infinitely many nodes with $f \leq$ cost of solution

Time: Exponential in *relative error in $h \times$ length of solution*

Space: Keeps all nodes in memory

Optimal: Yes (assuming h is admissible).

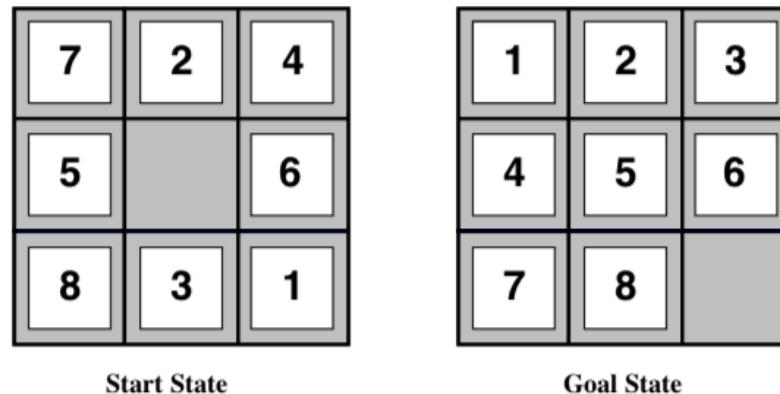
Iterative Deepening A* Search

- Iterative Deepening A* is a low-memory variant of A* that performs a series of depth-first searches but cuts off each search when the f exceeds current threshold, initially $f(start)$.
- The threshold is increased with each successive search.

Examples of Admissible Heuristics

$h_1(n)$ = total number of misplaced tiles

$h_2(n)$ = total Manhattan distance = \sum distance from goal position



$$h_1(\text{Start}) = 6$$

$$h_2(\text{Start}) = 4 + 0 + 3 + 3 + 1 + 0 + 2 + 1 = 14$$

How to Find Heuristic Functions ?

- Admissible heuristics can often be derived from the exact solution cost of a simplified or “relaxed” version of the problem. (i.e. with some of the constraints weakened or removed)
 - If the rules of the 8-puzzle are relaxed so that a tile can move anywhere, then $h_1(n)$ gives the shortest solution.
 - If the rules are relaxed so that a tile can move to any adjacent square, then $h_2(n)$ gives the shortest solution.

Dominance

- if $h_2(n) \geq h_1(n)$ for all n (both admissible)
 - h_2 **dominates** h_1 and is better for search.
 - Try make the heuristic h as large as possible, without exceeding h^* .
- typical search costs:

14-puzzle	IDS	= 3,473,941 nodes
	$A^*(h_1)$	= 539 nodes
	$A^*(h_2)$	= 113 nodes
24-puzzle	IDS	$\approx 54 \times 10^9$ nodes
	$A^*(h_1)$	= 39,135 nodes
	$A^*(h_2)$	= 1,641 nodes

Summary of Informed Search

- Heuristics can be applied to reduce search cost.
- Greedy Search tries to minimise cost from current node n to the goal.
- A* combines the advantages of Uniform-Cost Search and Greedy Search
- A* is complete, optimal and optimally efficient among all optimal search algorithms.
- Memory usage is still a concern for A*. IDA* is a low-memory variant.

Summary

- Informed search makes use of problem-specific knowledge to guide progress of search
- This can lead to a significant improvement in performance
- Much research has gone into admissible heuristics
 - Even on the automatic generation of admissible heuristics

Constraint Satisfaction Problems

COMP3411/9814: Artificial Intelligence

Constraint Satisfaction Problems

- Assignment problems (e.g. who teaches what class)
- Timetabling problems (e.g. which class is offered when and where?)
- Hardware configuration (e.g. minimise space for circuit layout)
- Transport scheduling (e.g. courier delivery, vehicle routing)
- Factory scheduling (optimise assignment of jobs to machines)
- Gate assignment (assign gates to aircraft to minimise transit)

Closely related to optimisation problems

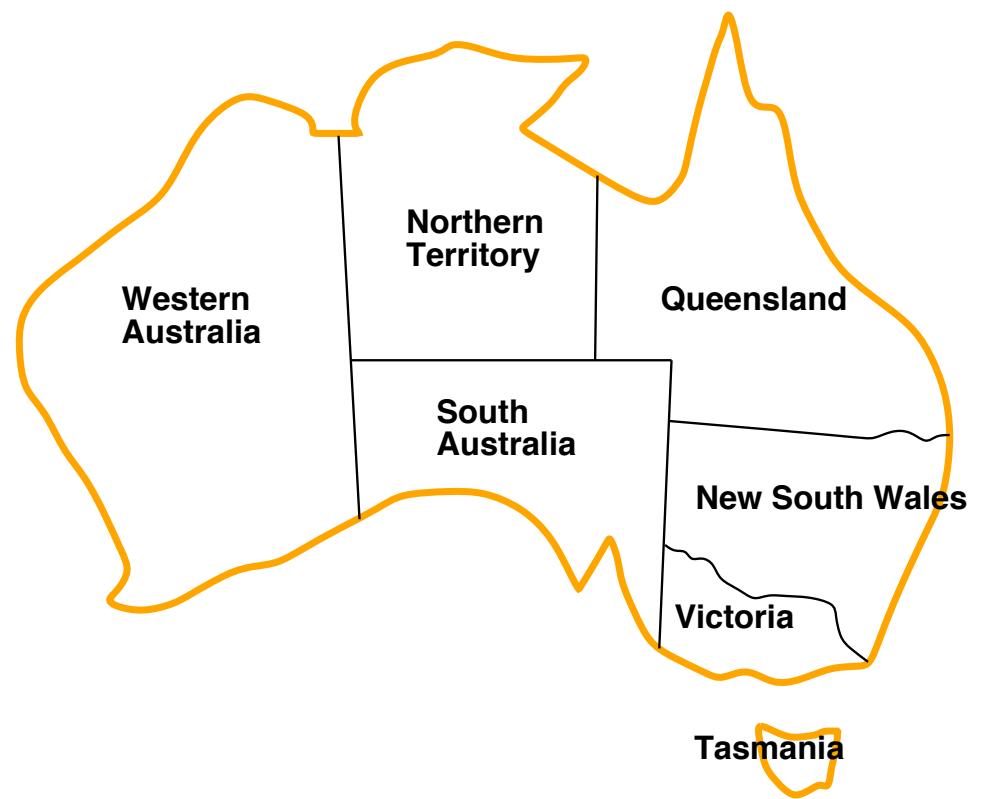
Lecture Overview

- Constraint Satisfaction Problems (CSPs)
- CSP examples
- Backtracking search and heuristics
- Forward checking and arc consistency
- Variable elimination
- Local search
 - Hill climbing
 - Simulated annealing

Constraint Satisfaction Problems (CSPs)

- Constraint Satisfaction Problems are defined by a set of variables X_i , each with a domain D_i of possible values, and a set of constraints C that specify allowable combinations of values.
- The aim is to find an assignment of the variables X_i from the domains D_i in such a way that none of the constraints C are violated.

Example: Map-Colouring



Variables: WA, NT, Q, NSW, V, SA, T

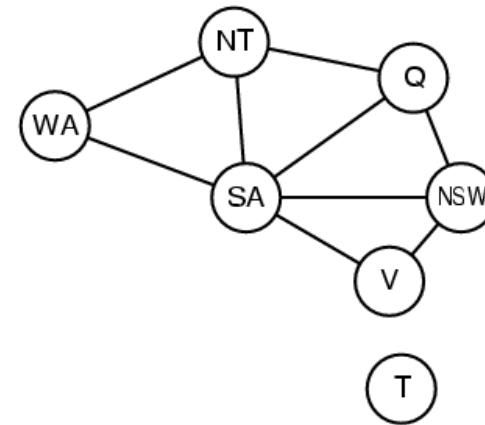
Domains: $D_i = \{\text{red, green, blue}\}$

Constraints: adjacent regions must have different colours

e.g. WA \neq NT, etc.

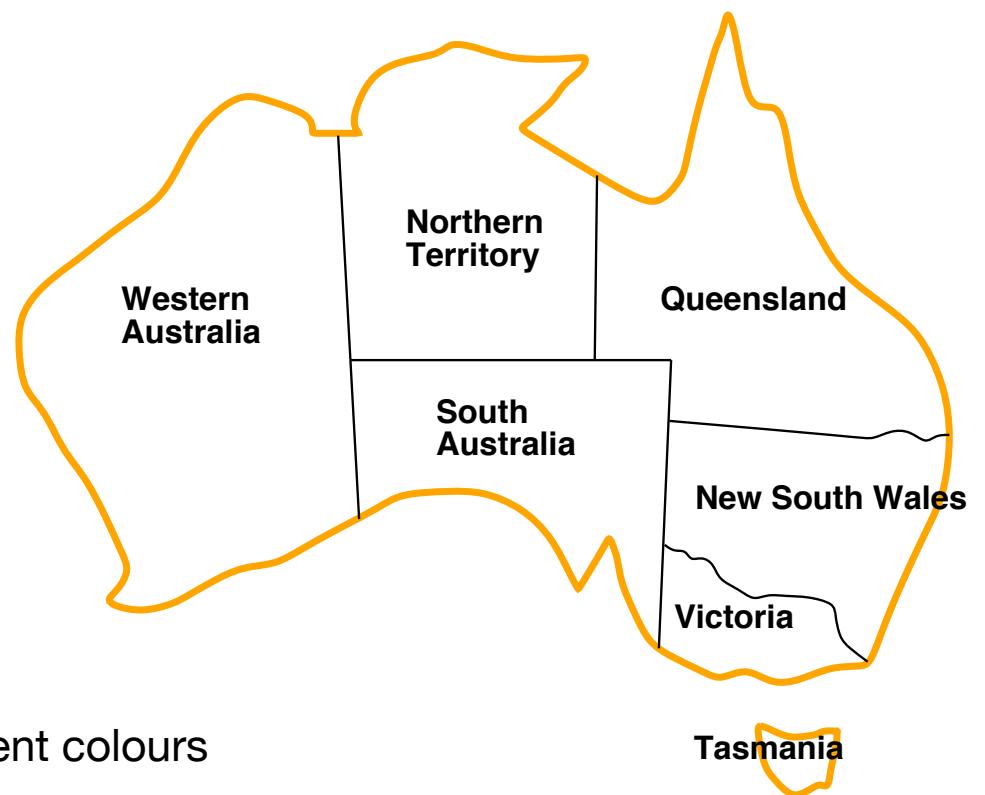
Constraint graph

Constraint graph: nodes are variables, arcs are constraints



Binary CSP: each constraint relates two variables

Example: Map-Colouring



Variables: WA, NT, Q, NSW, V, SA, T

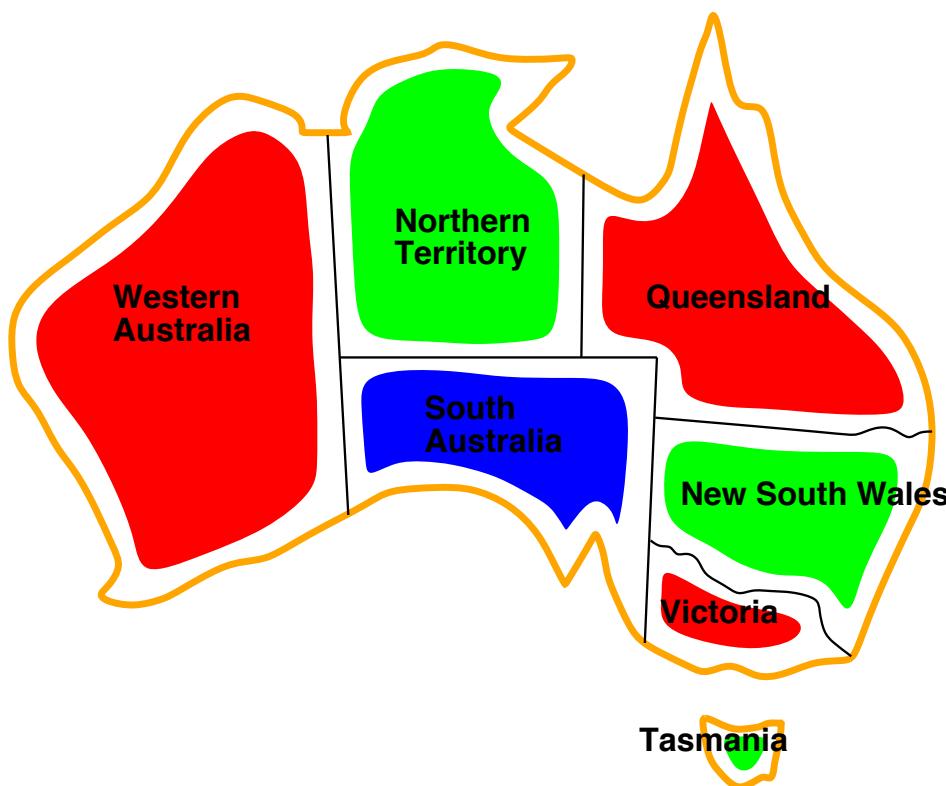
Domains: $D_i = \{\text{red, green, blue}\}$

Constraints: adjacent regions must have different colours

e.g. $WA \neq NT$, etc.

or $(WA,NT) \in \{(red,green), (red,blue), (green,red), (green,blue), (blue,red), (blue,green)\}$

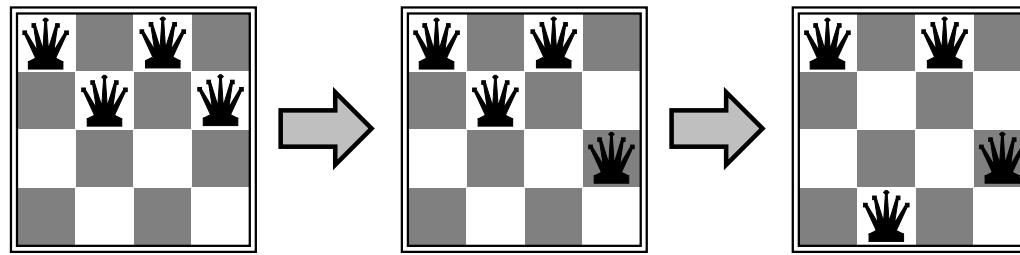
Example: Map-Colouring



{WA=red, NT=green, Q=red, NSW=green, V=red, SA=blue, T=green}

n-Queens Puzzle as a CSP

Assume one queen in each column. Domains are possible positions of queen in a column. Assignment is when each domain has one element. Which row does each one go in?



Variables: $Q1, Q2, Q3, Q4$

Domains: $D_i = \{1, 2, 3, 4\}$

Constraints:

$Q_i \neq Q_j$ (cannot be in same row)
 $|Q_i - Q_j| \neq |i - j|$ (or same diagonal)

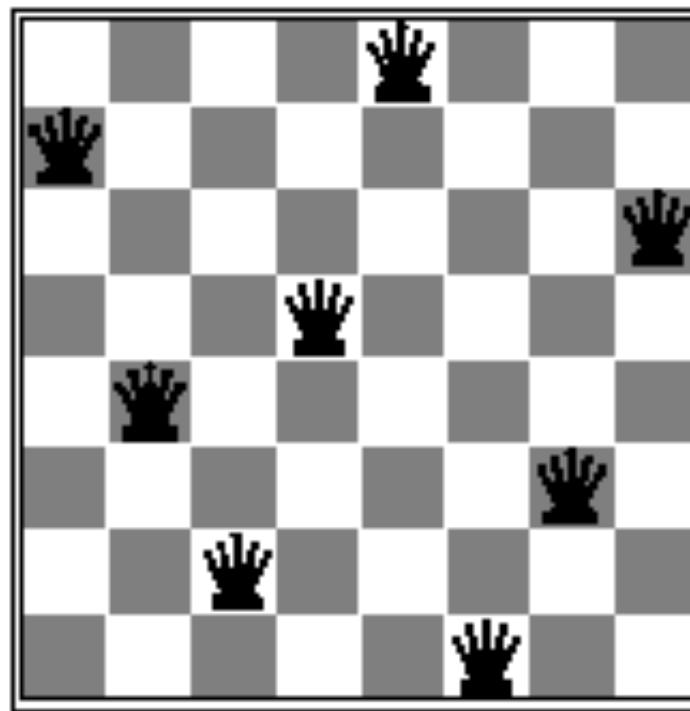
$\{1, 2, 1, 3\}$

Violates constraints because

$Q_1 = Q_3$ and

$|Q_1 - Q_2| = |i - j| = |1 - 2| = 1$

Example: n-Queens Puzzle



Put n queens on an n -by- n chess board so that no two queens are attacking each other.

Example: Cryptarithmetic

$$\begin{array}{r} \text{S} \quad \text{E} \quad \text{N} \quad \text{D} \\ + \quad \text{M} \quad \text{O} \quad \text{R} \quad \text{E} \\ \hline \text{M} \quad \text{O} \quad \text{N} \quad \text{E} \quad \text{Y} \end{array}$$

Variables:

D E M N O R S Y

Domains:

{0,1,2,3,4,5,6,7,8,9}

Constraints:

$M \neq 0, S \neq 0$ (unary constraints)

$Y = D+E$ or $Y = D+E - 10$, etc.

$D \neq E, D \neq M, D \neq N$, etc.

if there is a carry
over, have to add
to next column

Example: Cryptarithmetic

$$\begin{array}{r} \text{T} \quad \text{W} \quad \text{O} \\ + \quad \text{T} \quad \text{W} \quad \text{O} \\ \hline \text{F} \quad \text{O} \quad \text{U} \quad \text{R} \end{array}$$

Variables: F T U W R O C₁ C₂ C₃

Domains: {0,1,2,3,4,5,6,7,8,9}

Constraints: AllDifferent(F, T, U, W, R, O)

$$O + O = R + 10 \cdot C_1$$

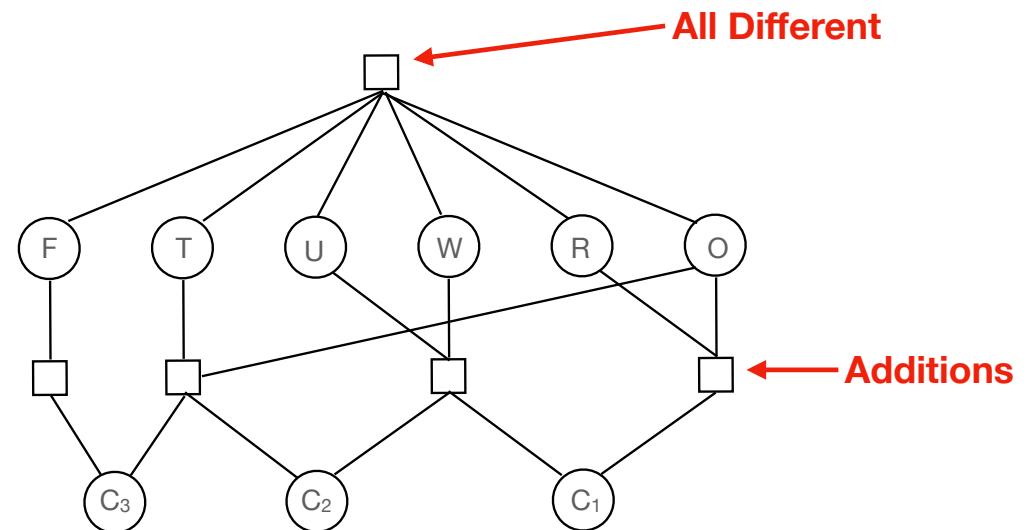
$$C_1 + W + W = U + 10 \cdot C_2$$

$$C_2 + T + T = O + 10 \cdot C_3$$

$$C_3 = F$$

Cryptarithmetic with Auxiliary Variables

$$\begin{array}{r} \text{T} \quad \text{W} \quad \text{O} \\ + \quad \text{T} \quad \text{W} \quad \text{O} \\ \hline \text{F} \quad \text{O} \quad \text{U} \quad \text{R} \end{array}$$



Constraints: AllDifferent(F, T , U , W , R , O)

$$O + O = R + 10 \cdot C_1$$

$$C_1 + W + W = U + 10 \cdot C_2$$

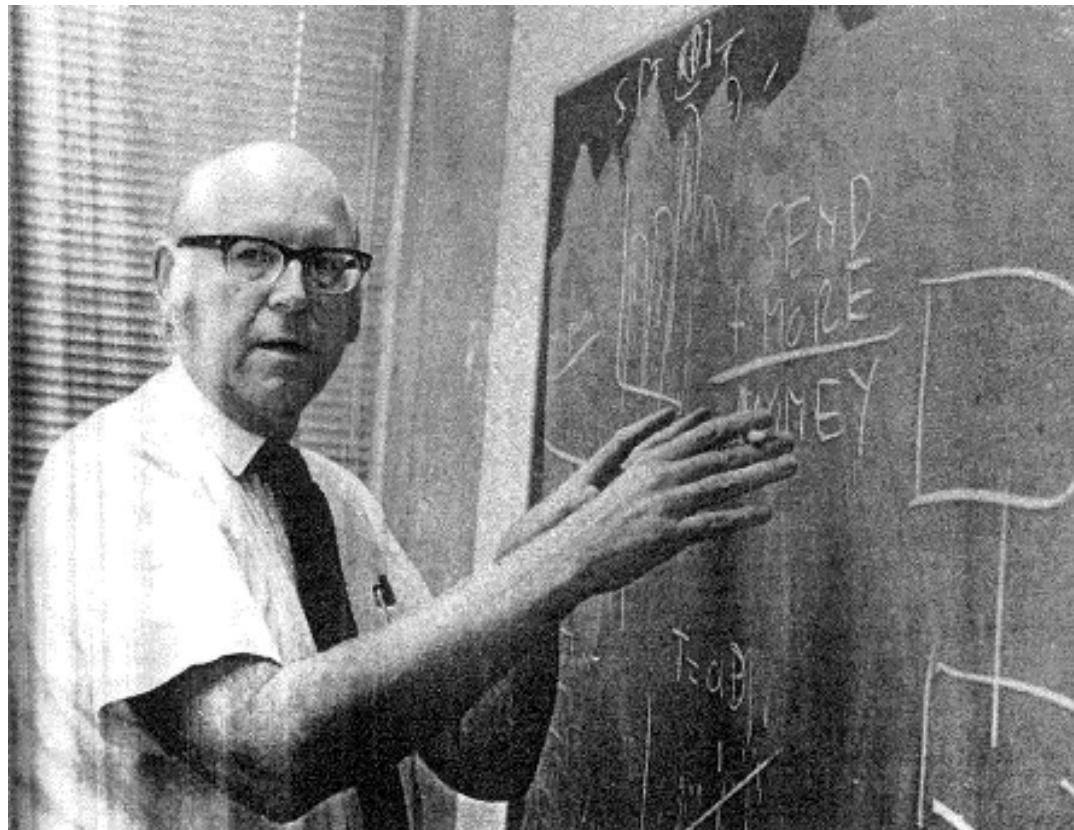
$$C_2 + T + T = O + 10 \cdot C_3$$

$$C_3 = F$$

Variables: F T U W R O X₁ X₂ X₃

Domains: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

Cryptarithmetic with Allen Newell



Book: Intended Rational Behavior

CSP Application - Factory scheduling

- An agent has to schedule a set of activities for a manufacturing process, involving casting, milling, drilling, and bolting.
- Each activity has a set of possible times at which it may start.
- The agent has to satisfy various constraints arising from prerequisite requirements and resource use limitations.
- For each activity there is a variable that represents the time that it starts:
 - B – start of bolding
 - D – start of drilling
 - C – start of casting

CSP Application - Factory scheduling

Constraints on the possible dates for three activities:

Variables: A, B, C - variables that represent the date of each activity

Domain of each variable is: $\{1, 2, 3, 4\}$

Constraint: $(A \leq B) \wedge (B < 3) \wedge (B < C) \wedge \neg(A = B \wedge C \leq 3)$

A starts on or before the same date as B and it cannot be that A and B start on the same date and C starts on or before day 3.

CSP Application - Factory scheduling

Constraint on the possible dates for three activities.

Variables: A, B, C - variables that represent the date of each activity

Domain of each variable is: $\{1, 2, 3, 4\}$

Constraint:

$$(A \leq B) \wedge (B < 3) \wedge (B < C) \wedge \neg(A = B \wedge C \leq 3)$$

A starts on or before the same date as B and it cannot be that A and B start on the same date and C starts on or before day 3.

Constraint defines its **extension**, e.g. table specifying the legal assignments:

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

Varieties of CSPs

- Discrete variables
 - Finite domains; size $d \Rightarrow O(d^n)$ complete assignments
 - e.g. Boolean CSPs, incl. Boolean satisfiability (NP-complete)
 - Infinite domains (integers, strings, etc.)
 - Job shop scheduling, variables are start/end days for each job
 - Need a constraint language, e.g. $\text{StartJob}_1 + 5 \leq \text{StartJob}_3$
 - Linear constraints solvable, nonlinear undecidable
- Continuous variables
 - e.g. start/end times for Hubble Telescope observations
 - Linear constraints solvable in polynomial time by *linear programming* methods

Types of constraints

- **Unary** constraints involve a single variable

$$M \neq 0$$

- **Binary** constraints involve pairs of variables

$$SA \neq WA$$

- **Higher-order** constraints involve 3 or more variables

$$Y = D + E \text{ or } Y = D + E - 10$$

- **Inequality** constraints on continuous variables

$$\text{EndJob1} + 5 \leq \text{StartJob3}$$

- **Soft** constraints (preferences)

11am lecture is better than 8am lecture!

Path Search vs Constraint Satisfaction

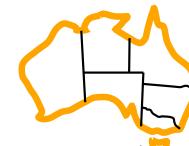
Difference between path search problems and CSPs

- Path Search Problems (e.g. Delivery Robot)
 - Knowing the final state is easy
 - Difficult part is how to get there
- Constraint Satisfaction Problems (e.g. n -Queens)
 - Difficult part is knowing the final state
 - How to get there is easy

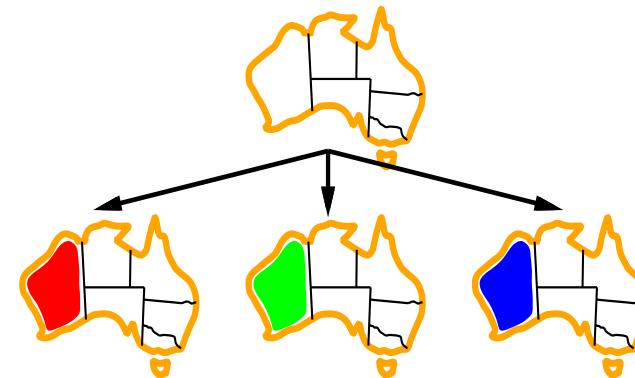
Backtracking Search

- CSPs can be solved by assigning values to variables one by one, in different combinations.
- Whenever a constraint is violated, go back to most recently assigned variable and assign it a new value.
- Can use Depth First Search, where states are defined by the values assigned so far:
 - Initial state: empty assignment.
 - Successor function:
 - assign a value to an unassigned variable that does not conflict with previously assigned values of other variables.
 - If no legal values remain, the successor function fails
 - Goal test: all variables have been assigned a value, and no constraints have been violated.

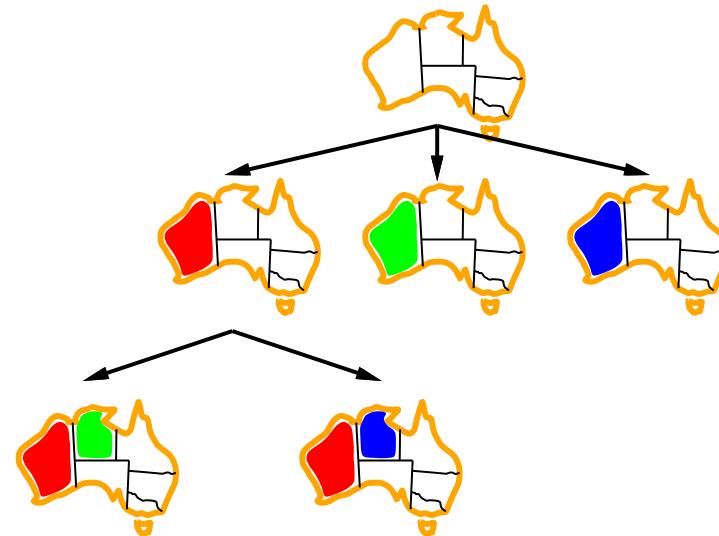
Backtracking example



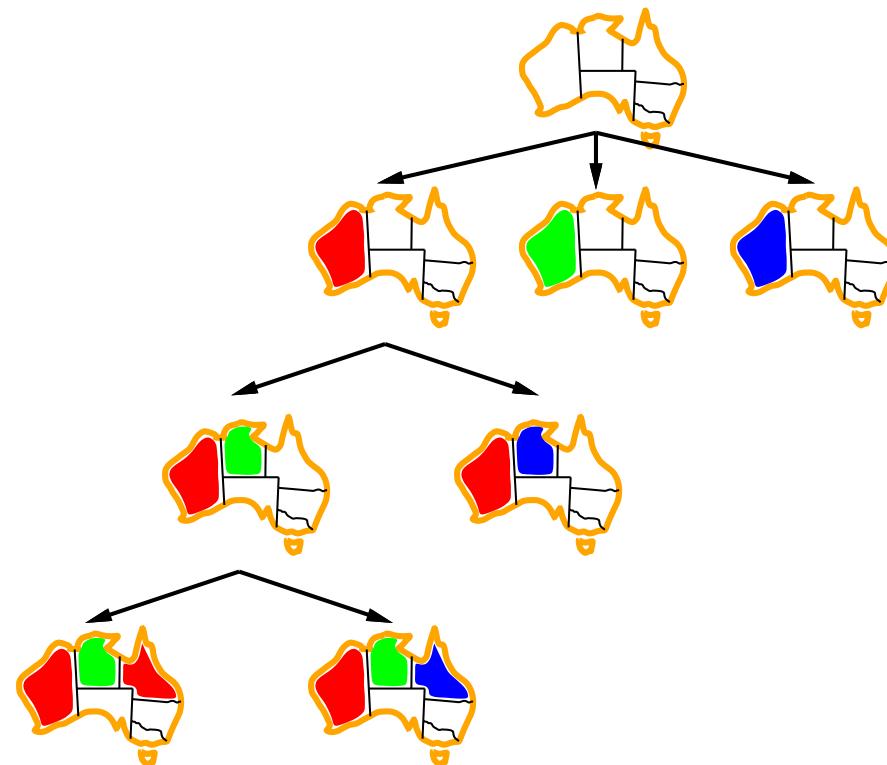
Backtracking example



Backtracking example



Backtracking example



Backtracking Search Properties

- If there are n variables, every solution will occur at exactly depth n .
- Variable assignments are **commutative**

[WA = red then NT = green] same as [NT = green then WA = red]

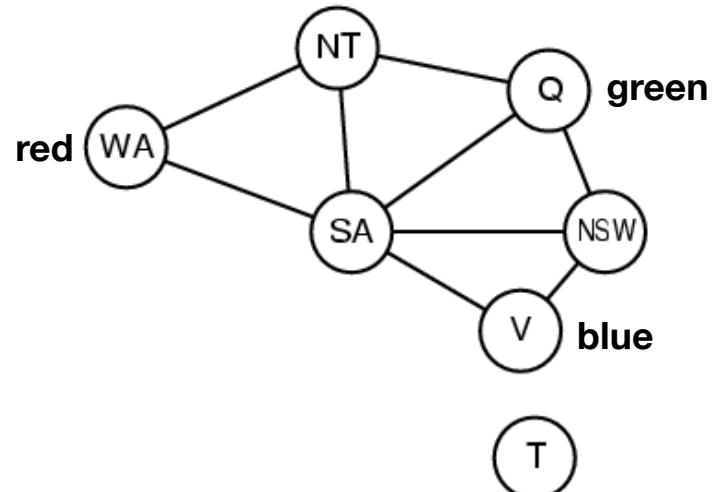
- Backtracking search can solve n-Queens for $n \approx 25$

Programming Constraints

```
variables([wa=_, nt=_, q=_, nsw=_, v=_, sa=_, t=_]).
```

```
domain(red).  
domain(green).  
domain(blue).  
  
connected(wa, nt).  
connected(wa, sa).  
connected(nt, q).  
connected(nt, sa).  
connected(sa, q).  
connected(sa, nsw).  
connected(sa, v).  
connected(q, nsw).  
connected(v, nsw).
```

```
adjacent(A, B) :- connected(A, B).  
adjacent(A, B) :- connected(B, A).
```



Programming Constraints

```
solve(V) :-  
    variables(V),  
    assign_all(V).  
  
assign_all([]).  
assign_all([State|OtherStates]) :-  
    assign_all(OtherStates),  
    assign_variable(State, OtherStates).  
  
assign_variable(Var = Colour, OtherStates) :-  
    domain(Colour),  
    constraint(Var = Colour, OtherStates).  
  
constraint(S1 = C, OtherStates) :-  
    \+ (adjacent(S1, S2), member(S2 = C, OtherStates)).
```

Prolog's *not* operator



All Solutions by Backtracking

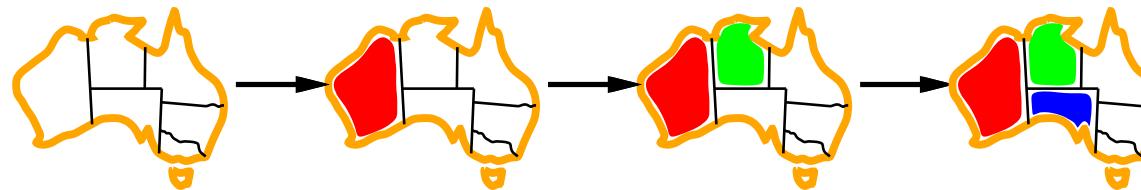
WA	NT	Q	NSW	V	SA	T
green	blue	green	blue	green	red	red
blue	green	blue	green	blue	red	red
red	blue	red	blue	red	green	red
blue	red	blue	red	blue	green	red
red	green	red	green	red	blue	red
green	red	green	red	green	blue	red
green	blue	green	blue	green	red	green
blue	green	blue	green	blue	red	green
red	blue	red	blue	red	green	green
blue	red	blue	red	blue	green	green
red	green	red	green	red	blue	green
green	red	green	red	green	blue	green
green	blue	green	blue	green	red	blue
blue	green	blue	green	blue	red	blue
red	blue	red	blue	red	green	blue
blue	red	blue	red	blue	green	blue
red	green	red	green	red	blue	blue
green	red	green	red	green	blue	blue

Improvements to Backtracking Search

- Which variable should be assigned next?
- In what order should its values be tried?
- Can we detect inevitable failure early?

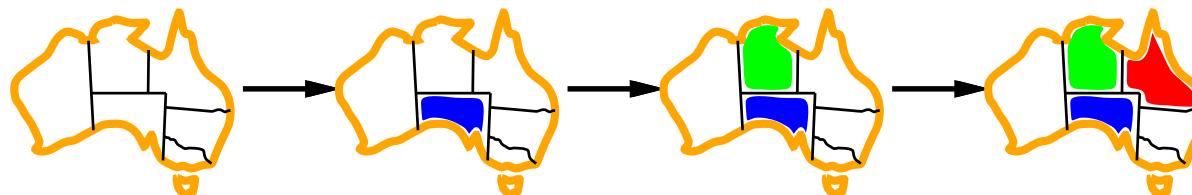
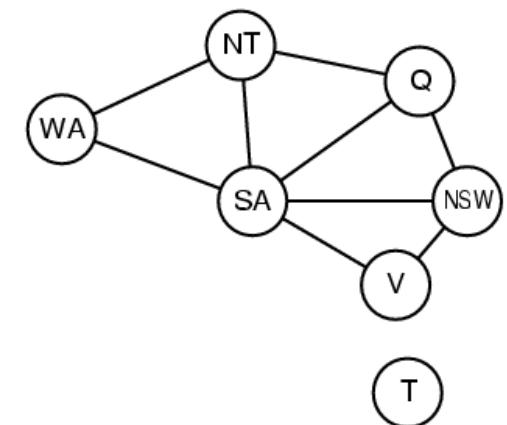
Minimum Remaining Values

- Minimum Remaining Values (MRV)
 - choose the variable with the fewest legal remaining values
 - Most constrained variable



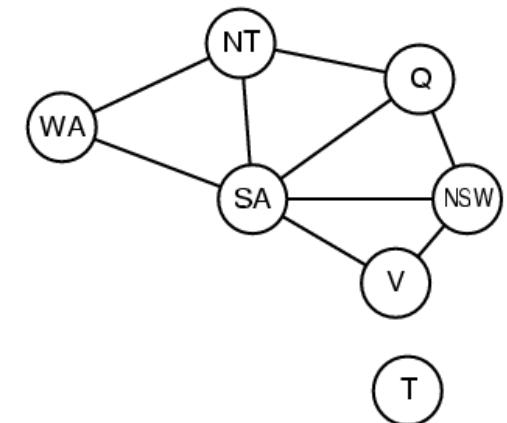
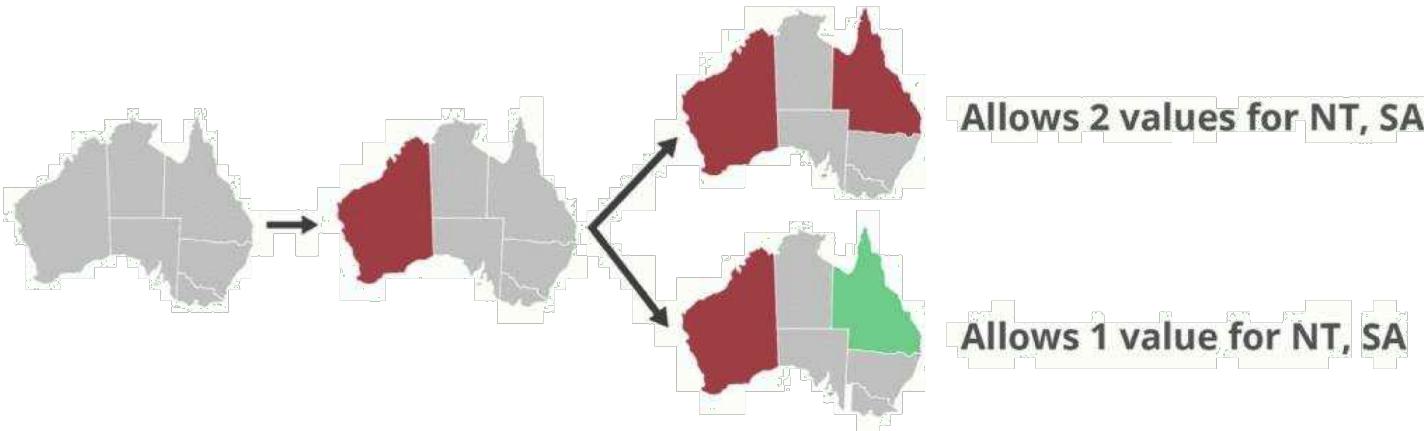
Degree Heuristic

- Tie-breaker among MRV variables
- Degree heuristic:
 - choose the variable with the most constraints on variables (i.e. most edges in graph)
 - If same degree, choose any one



Least Constraining Value

- Given a variable, choose the least constraining value:
the one that rules out the fewest values in the remaining variables



- More generally, 3 allowed values would be better than 2, etc.
Combining these heuristics makes 1000 queens feasible.

Forward Checking

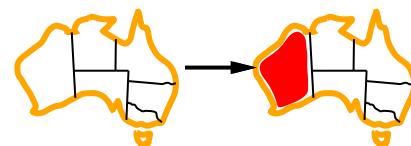
- Keep track of remaining legal values for unassigned variables
- Terminate search when any variable has no legal values
 - prune off that part of the search tree, and backtrack



Initially, all values are available.

Forward Checking

- Keep track of remaining legal values for unassigned variables
- Terminate search when any variable has no legal values
 - prune off that part of the search tree, and backtrack



WA	NT	Q	NSW	V	SA	T
█ R G B	█ R G B	█ R G B	█ R G B	█ R G B	█ R G B	█ R G B
█ R	█ G B	█ R G B	█ R G B	█ R G B	█ G B	█ R G B

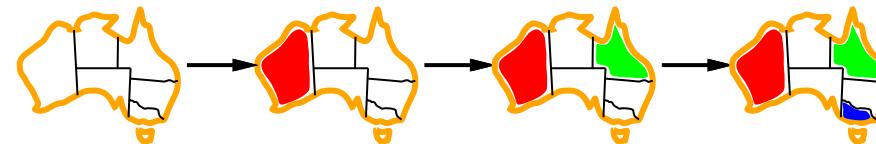
Forward Checking

- Keep track of remaining legal values for unassigned variables
 - Terminate search when any variable has no legal values
 - prune off that part of the search tree, and backtrack



Forward Checking

- Keep track of remaining legal values for unassigned variables
- Terminate search when any variable has no legal values
 - prune off that part of the search tree, and backtrack

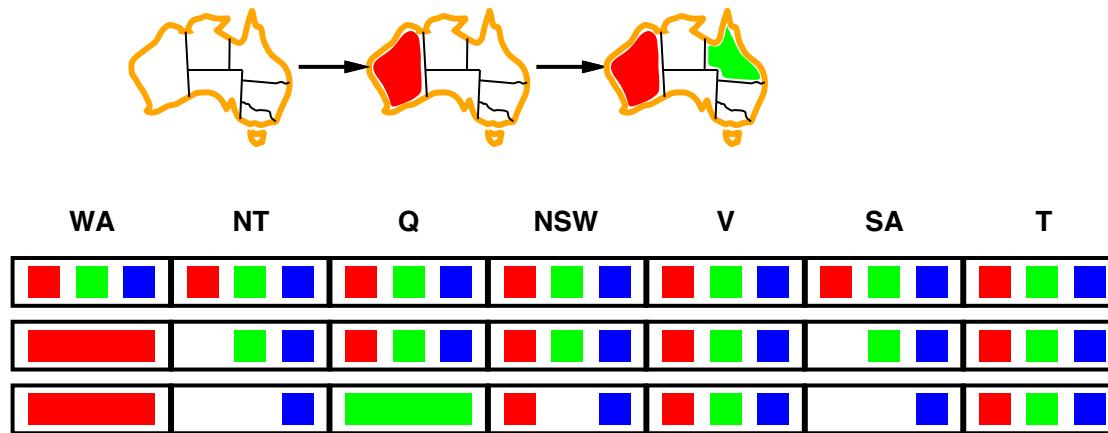


WA	NT	Q	NSW	V	SA	T
■■■	■■■	■■■	■■■	■■■	■■■	■■■
■■■		■■■	■■■	■■■	■■■	■■■
■■■			■■■	■■■	■■■	■■■
■■■				■■■		■■■

Fail

Constraint Propagation

Forward checking propagates information from assigned to unassigned variables, but doesn't provide early detection for all failures:



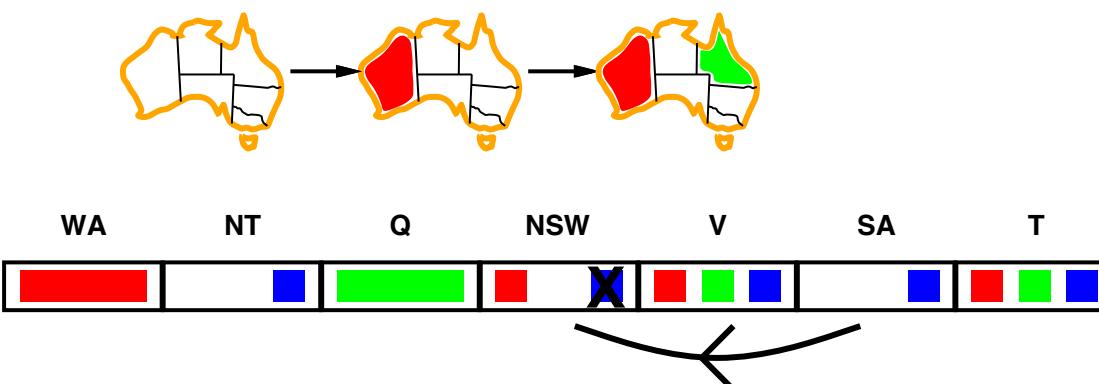
NT and SA cannot both be blue

Constraint propagation repeatedly enforces constraints locally

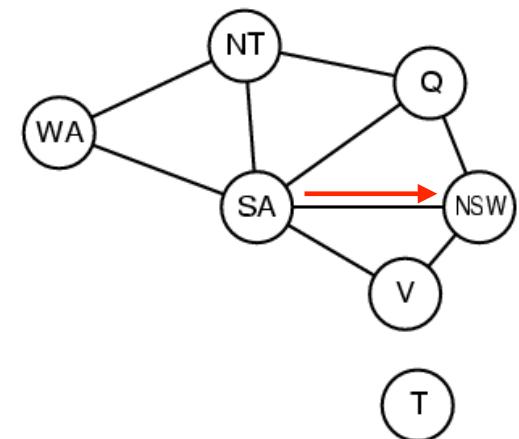
Arc Consistency

$X \rightarrow Y$ is consistent if

for **every** value x of X there is **some** allowed y



Only possible value for SA is blue, so NSW can't be blue

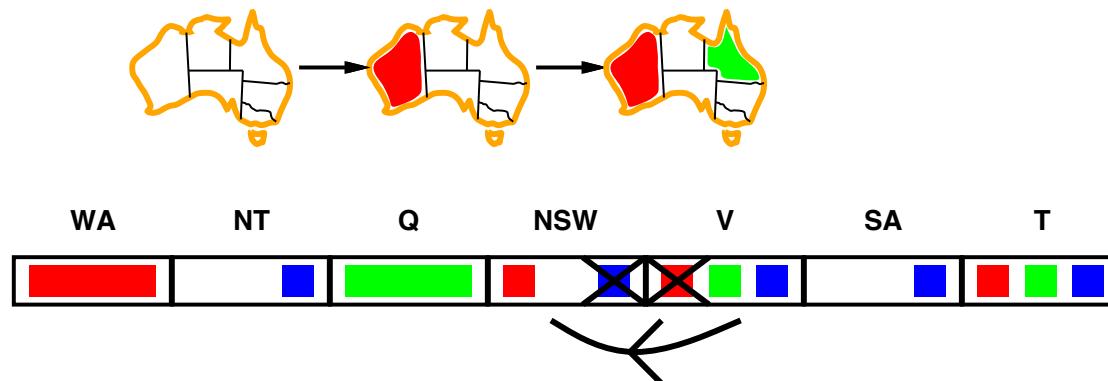


Propagate arc consistency on the graph

Arc Consistency

$X \rightarrow Y$ is consistent if

for **every** value x of X there is **some** allowed y

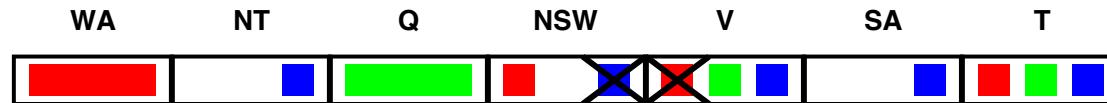


- If X loses a value, neighbours of X need to be rechecked.
- Since NSW can only be red now, V cannot be red

Arc Consistency

$X \rightarrow Y$ is consistent if

for **every** value x of X there is **some** allowed y

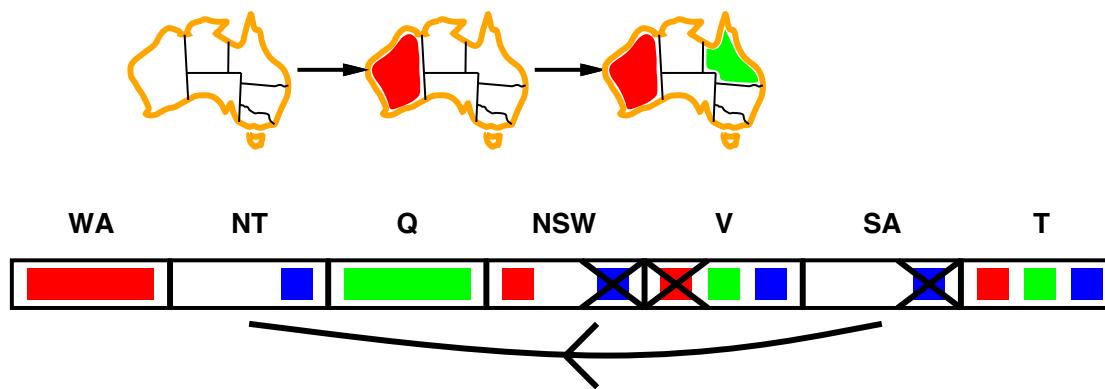


- If X loses a value, neighbours of X need to be rechecked.
- Arc consistency detects failure earlier than forward checking
- Can be run as a preprocessor after each assignment

Arc Consistency

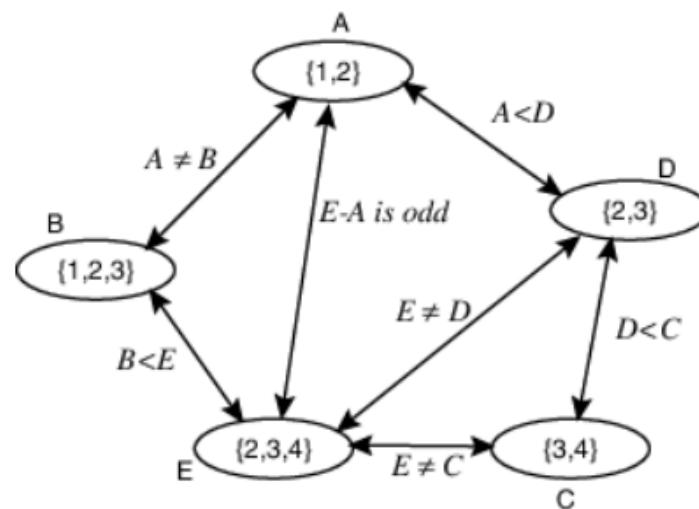
$X \rightarrow Y$ is consistent if

for every value x of X there is some allowed y



- Arc consistency detects failure earlier than forward checking.
- For some problems, it can speed up search enormously.
- For others, it may slow the search due to computational overheads.

Variable Elimination

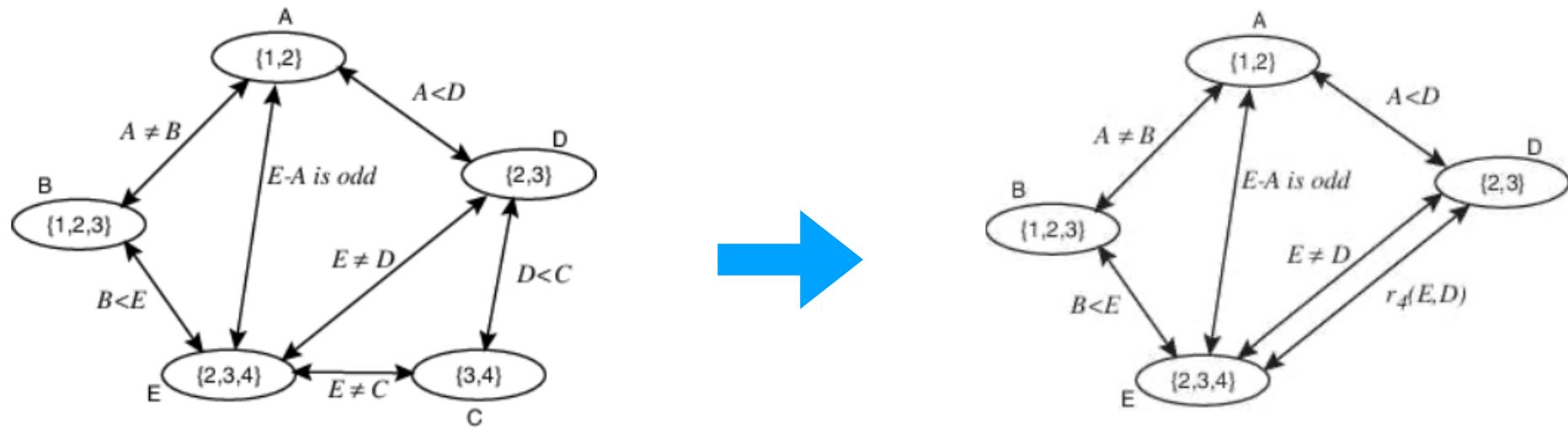


Variables: A, B, C, D, E

Domains: A = {1, 2}, B = {1, 2, 3}, C = {3, 4}, D = {2, 3}, E = {2, 3, 4}

Constraints: A ≠ B, E ≠ C, E ≠ D, A < D, B < E, D < C, E-A is odd

Variable Elimination



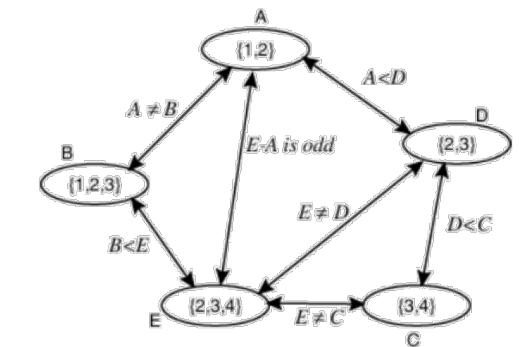
- Eliminates variable, one by one
- Replace them with constraints on adjacent variables

Variable Elimination Example

1. Select a variable X

$r_1 : C \neq E$	C	E
	3	2
	3	4
	4	2
	4	3

2. Enumerate constraints

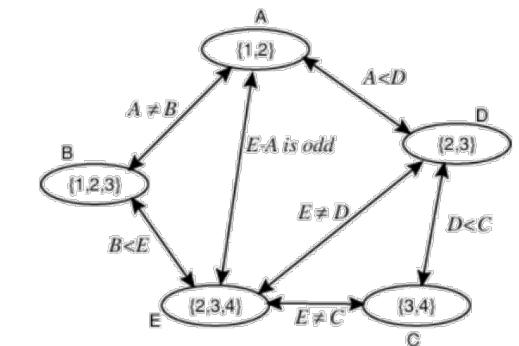


Variable Elimination Example

1. Select a variable X

$r_1 : C \neq E$		$r_2 : C > D$	
C	E	C	D
3	2	3	2
3	4	4	2
4	2	4	3
4	3		

2. Enumerate constraints



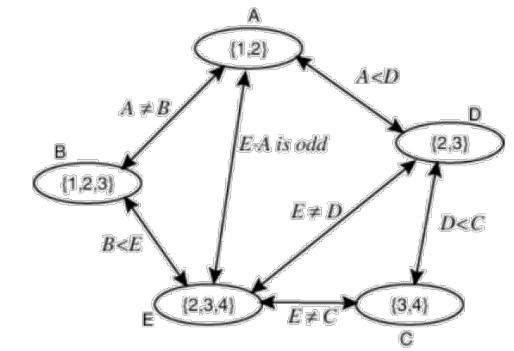
Variable Elimination Example

1. Select a variable X

2. Join the constraints in which X appears

$r_1 : C \neq E$	C	E	$r_2 : C > D$	C	D
	3	2		3	2
	3	4		4	2
	4	2		4	3
	4	3			

$r_3 : r_1 \bowtie r_2$	C	D	E
	3	2	2
	3	2	4
	4	2	2
	4	2	3
	4	3	2
	4	3	3



Variable Elimination Example

1. Select a variable X
2. Join the constraints in which X appears
3. Project join onto its variables other than X (forming r_4)

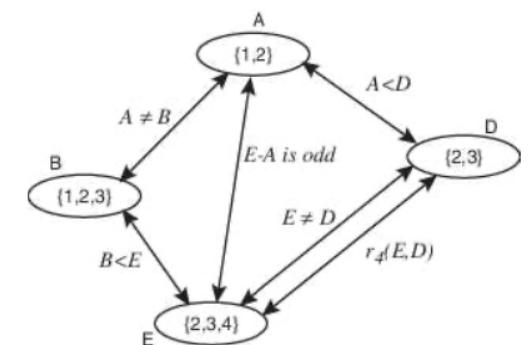
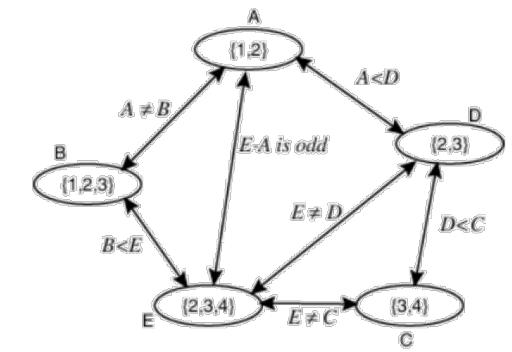
$r_1 : C \neq E$	C	E
	3	2
	3	4
	4	2
	4	3

$r_2 : C > D$	C	D
	3	2
	4	2
	4	3

$r_3 : r_1 \bowtie r_2$	C	D	E
	3	2	2
	3	2	4
	4	2	2
	4	2	3
	4	3	2
	4	3	3

$r_4 : \pi_{\{D,E\}} r_3$	D	E
	2	2
	2	3
	2	4
	3	2
	3	3

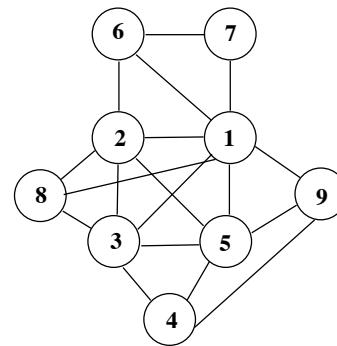
➡ new constraint



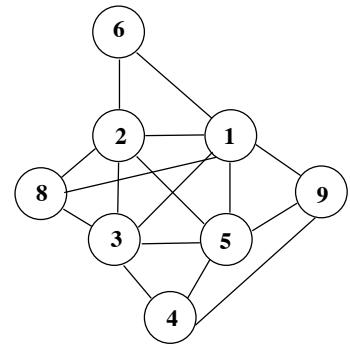
To generate one or all solutions, the algorithm remembers the joined relation C, D, E to construct a solution that involves C from a solution to the reduced network.

Variable Elimination

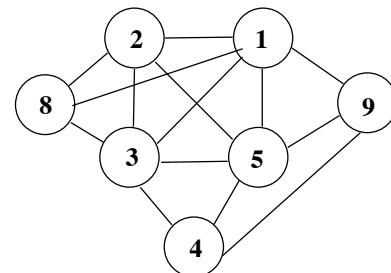
1. Select a variable X
2. Join the constraints in which X appears, forming constraint R1
3. Project R1 onto its variables other than X, forming R2
4. Replace all of the constraints in which X appears by R2
5. Recursively solve the simplified problem, forming R3
6. Return R1 joined with R3



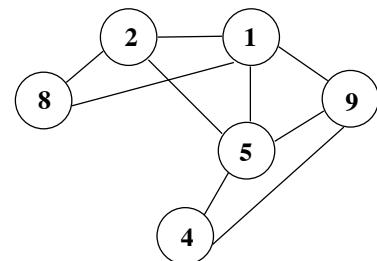
a) Initial constraint graph



b) After eliminating X7



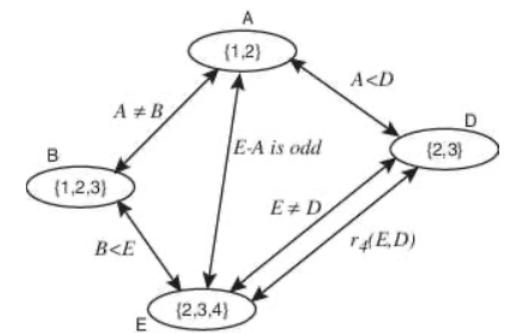
c) After eliminating X6



d) After branching in X3

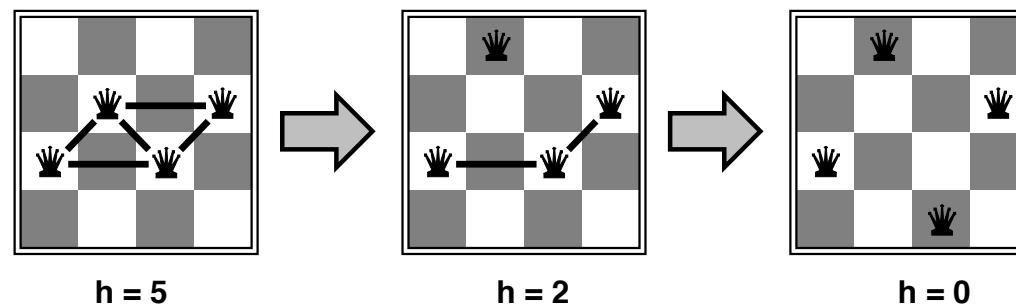
Variable Elimination

1. If there is only one variable,
return the join of all the relations in the constraints
2. Otherwise
 - 2.1. Select a variable X
 - 2.2. Join the constraints in which X appears, forming constraint R1
 - 2.3. Project R1 onto its variables other than X, forming R2
 - 2.4. Replace all of the constraints in which X appears by R2
 - 2.5. Recursively solve the simplified problem, forming R3
 - 2.6. Return R1 joined with R3



Local Search

There is another class of algorithms for solving CSP's, called “[Iterative Improvement](#)” or “Local Search”.

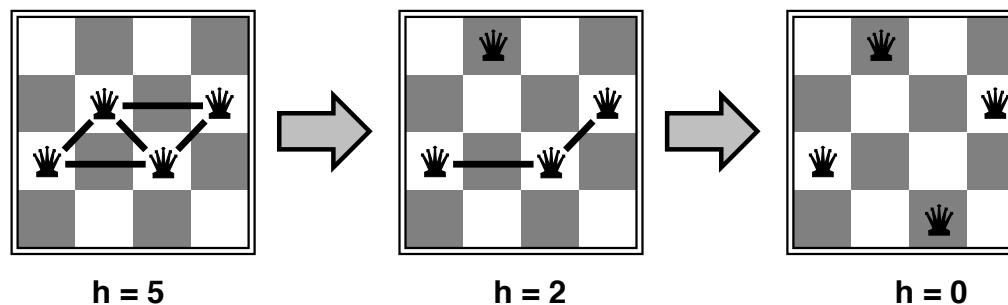


Local Search

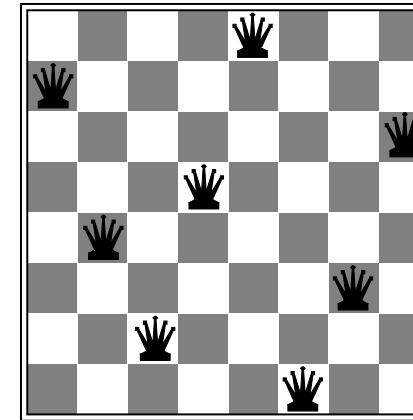
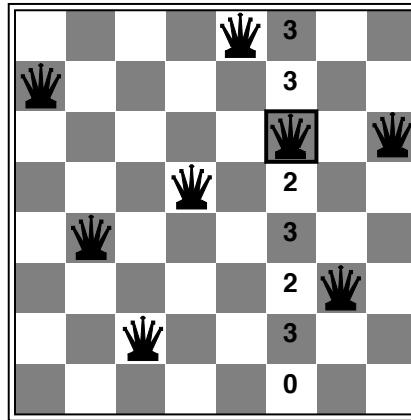
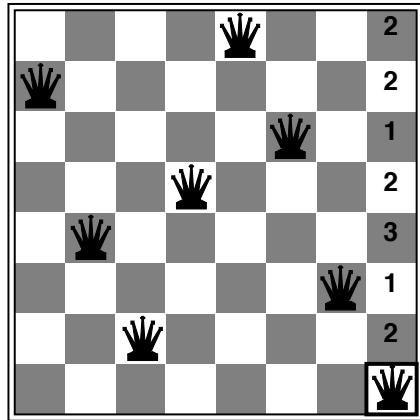
There is another class of algorithms for solving CSP's, called “[Iterative Improvement](#)” or “Local Search”.

- [Iterative Improvement](#)

- assign all variables randomly in the beginning (thus violating several constraints),
- change one variable at a time, trying to reduce the number of violations at each step.
- Greedy Search with $h = \text{number of constraints violated}$

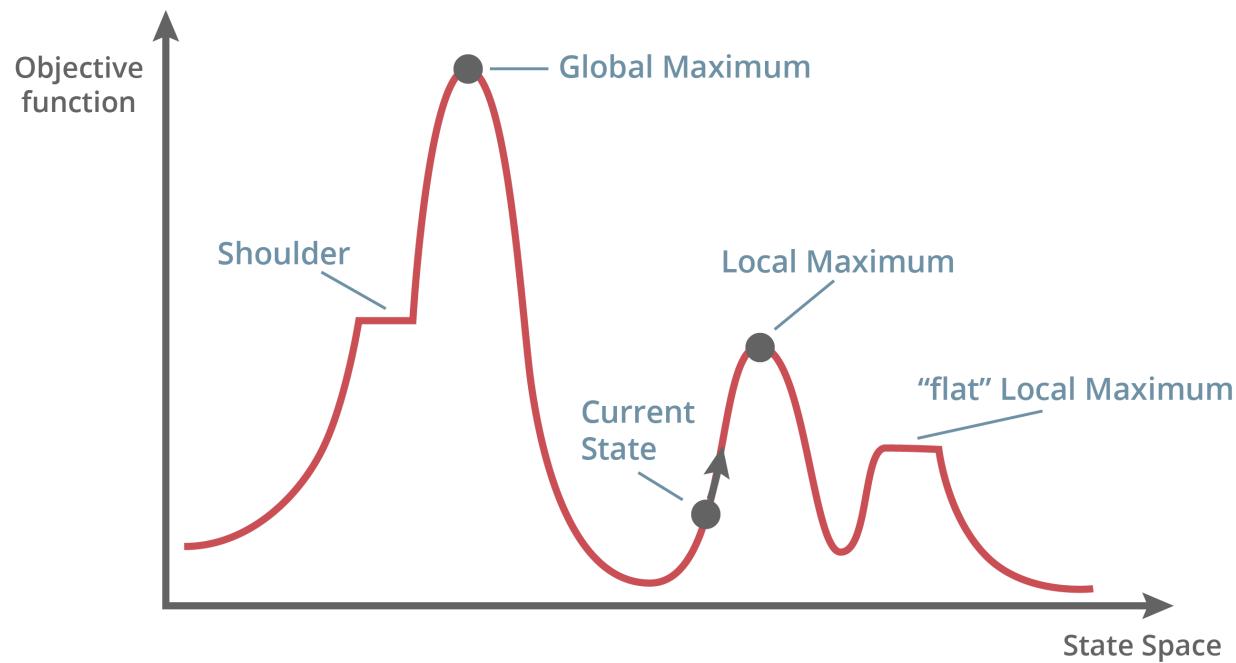


Hill-climbing by min-conflicts



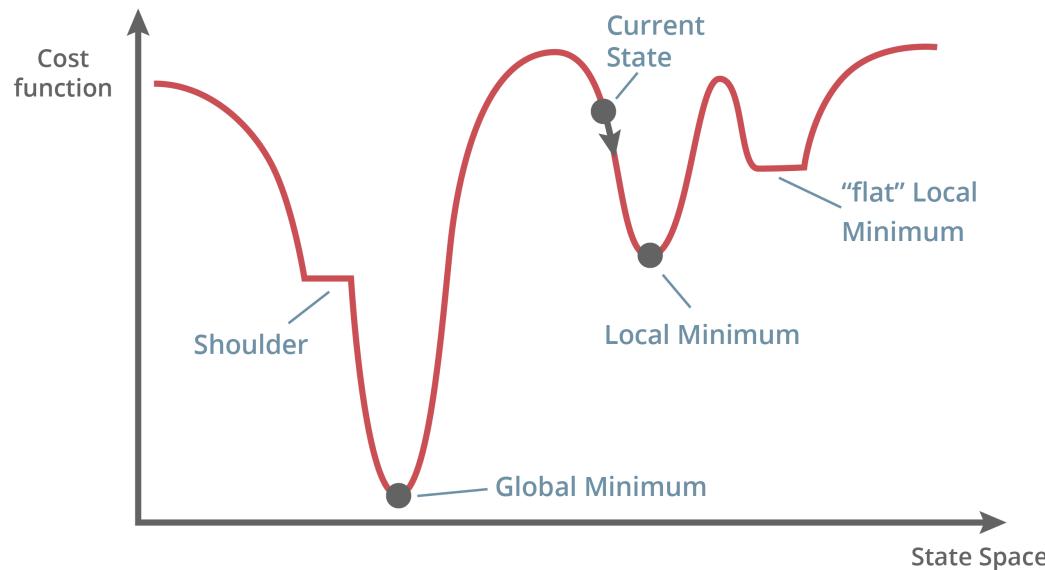
- Variable selection: randomly select any conflicted variable
- Value selection by min-conflicts heuristic
 - choose value that violates the fewest constraints

Flat regions and local optima



- May have to go sideways or backwards to make progress towards the solution
- Exploration vs Exploitation

Inverted View



When we are minimising violated constraints, it makes sense to think of starting at the top of a ridge and climbing down into the valleys.

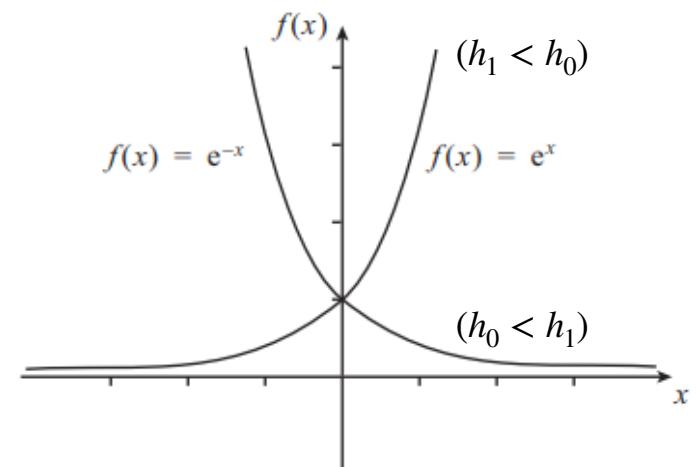
Simulated Annealing

- Stochastic hill climbing based on difference between evaluation of previous state (h_0) and new state (h_1).
 - If $h_1 < h_0$, definitely make the change (smaller is better)
 - Otherwise, make the change with probability

$$e^{-(h_1 - h_0)/T}$$

where T is a “temperature” parameter.

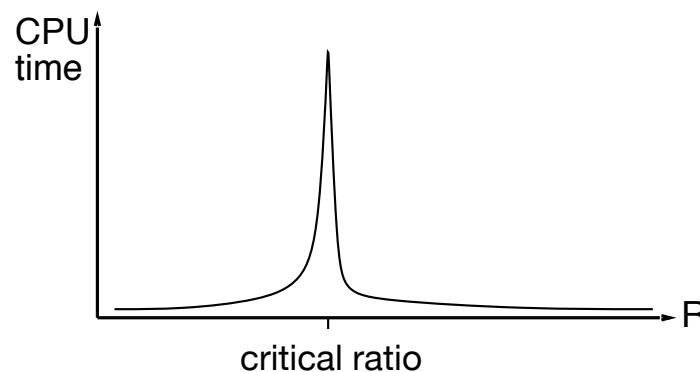
- Reduces to ordinary hill climbing as $T \rightarrow 0$
- Becomes totally random search as $T \rightarrow \infty$
- Sometimes, we gradually decrease the value of T during the search



Phase Transition in CSP's

- Given random initial state, hill climbing by min-conflicts with random restarts can solve n-queens in almost constant time for arbitrary n with high probability (e.g., n = 10,000,000).
- In general, randomly-generated CSP's tend to be easy if there are very few or very many constraints. They become extra hard in a narrow range of the ratio

$$R = \frac{\text{number of constraints}}{\text{number of variables}}$$

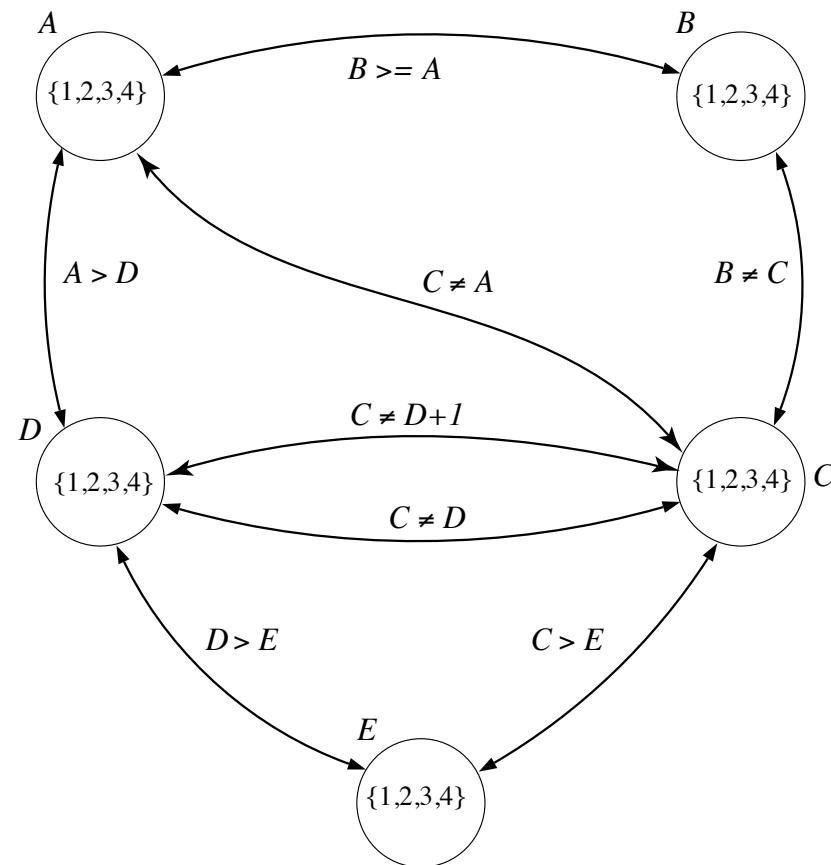


Summary

- CSPs are a special kind of search problem:
 - states defined by values of a fixed set of variables
 - goal test defined by constraints on variable values
- Backtracking = depth-first search with one variable assigned per node
- Variable ordering and value selection heuristics help significantly
- Forward checking prevents assignments that guarantee later failure
- Constraint propagation (e.g., arc consistency) does additional work to constrain values and detect inconsistencies
- Iterative min-conflicts is usually effective in practice
- Simulated Annealing can help to escape from local optima

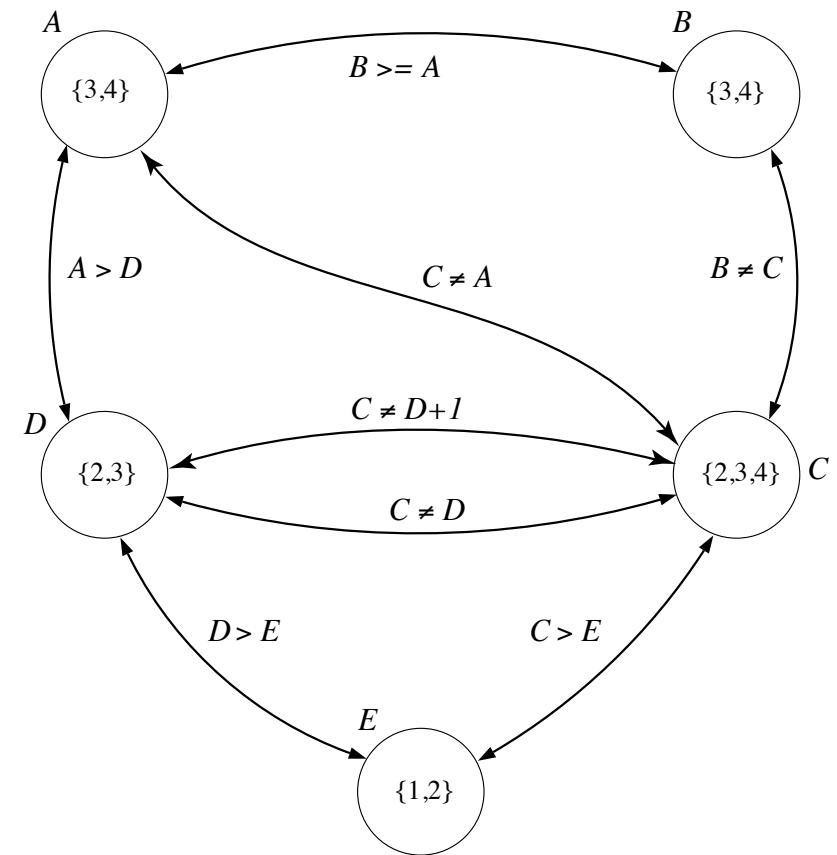
Constraint Programming

Finite Domain



Arc Consistency

Arc	Relation	Value(s) Removed
$\langle D, E \rangle$	$D > E$	$D = 1$
$\langle E, D \rangle$	$D > E$	$E = 4$
$\langle C, E \rangle$	$C > E$	$C = 1$
$\langle D, A \rangle$	$A > D$	$D = 4$
$\langle A, D \rangle$	$A > D$	$A = 1 \& A = 2$
$\langle B, A \rangle$	$B \geq A$	$B = 1 \& B = 2$
$\langle E, D \rangle$	$D > E$	$E = 3$



Constraint Ordering is Important

```
solve(A, B, C, D, E) :-  
    domain(C),  
    domain(D),  
    domain(A),  
    domain(B),  
    domain(E),  
    A > D,  
    D > E,  
    C =\= A,  
    C > E,  
    C =\= D,  
    B >= A,  
    B =\= C,  
    C =\= D + 1.
```

```
solve(A, B, C, D, E) :-  
    domain(C),  
    domain(D),  
    C =\= D,  
    C =\= D + 1,  
    domain(A),  
    A > D,  
    C =\= A,  
    domain(B),  
    B >= A,  
    B =\= C,  
    domain(E),  
    C > E,  
    D > E.
```

Much faster !

```
domain(1).  
domain(2).  
domain(3).  
domain(4).
```

CLP(FD)

- SWI Prolog (and others) include constraint programming libraries
 - Others: ECLIPSe, YAP, GNU-Prolog, Ciao, ...
- Non-standard extensions, so beware!
- They change Prolog's normal depth-first search for variable bindings to incorporate constraint solving methods (including arc consistency, etc).

Example

```
:  
- use_module(library(clpfd)).  
  
solve(A, B, C, D, E) :-  
    [A, B, C, D, E] ins 1..4,  
    A #> D,  
    D #> E,  
    C #\= A,  
    C #> E,  
    C #\= D,  
    B #>= A,  
    B #\= C,  
    C #\= D + 1,  
    labeling([], [A, B, C, D, E]).
```

Annotations:

- [A, B, C, D, E] ins 1..4, ← Declare domain
- A #> D,
D #> E,
C #\= A,
C #> E,
C #\= D,
B #>= A,
B #\= C,
C #\= D + 1, ← '#' means operator is a constraint,
satisfied by constraint solving
rather than depth-first search
- labeling([], [A, B, C, D, E]). ← Assign values

Solution to FD Problem

```
?- solve(A, B, C, D, E).
```

A = 3,

B = 3,

C = 4,

D = 2,

E = 1 ;

A = 4,

B = 4,

C = 2,

D = 3,

E = 1

Consistency Check

?- constraints(A, B, C, D, E).

A in 3..4,
C #\= A,
B #>= A,
D #=< A + -1,
C in 2..4,
C #\= D+1,
B #\= C,
C #\= D,
E #=< C + -1,
D in 2..3,
E #=< D + -1,
E in 1..2,
B in 3..4

Cryptarithmetic

$$\begin{array}{r} \text{D O N A L D} \\ + \text{G E R A L D} \\ \hline \text{R O B E R T} \end{array}$$

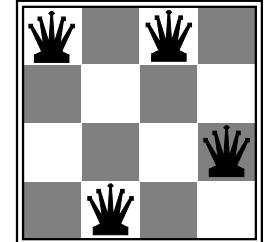
Cryptarithmetic

```
% Cryptarithmetic puzzle DONALD + GERALD = ROBERT in CLP(FD)

:- use_module(library(clpfd)).

solve([D,O,N,A,L,D],[G,E,R,A,L,D],[R,O,B,E,R,T]) :-  
    Vars = [D,O,N,A,L,G,E,R,B,T],  
    Vars ins 0..9,  
    all_different(Vars),  
    100000*D + 10000*O + 1000*N + 100*A + 10*L + D +  
    100000*G + 10000*E + 1000*R + 100*A + 10*L + D #=  
    100000*R + 10000*O + 1000*B + 100*E + 10*R + T,  
    labeling([], Vars).  
  
% All variables in the puzzle  
% They are all decimal digits  
% They are all different  
  
?- solve(X, Y, Z).  
  
X = [ 5, 2, 6, 4, 8, 5 ],  
Y = [ 1, 9, 7, 4, 8, 5 ],  
Z = [ 7, 2, 3, 9, 7, 0 ]
```

N-Queens



```
% The k-th element of Cols is the column number of the queen in row k.
```

[1, 4, 1, 3]

```
:  
- use_module(library(clpf)).
```

```
n_queens(N, Qs) :-  
    length(Qs, N),  
    Qs ins 1..N,  
    safe_queens(Qs).
```

```
safe_queens([]).  
safe_queens([Q|Qs]) :-  
    safe_queens(Qs, Q, 1),  
    safe_queens(Qs).
```

```
safe_queens([], _, _).  
safe_queens([Q|Qs], Q0, D0) :-  
    Q0 #\= Q,  
    abs(Q0 - Q) #\= D0,  
    D1 #= D0 + 1,  
    safe_queens(Qs, Q0, D1).  
?- n_queens(8, Qs), labeling([ff], Qs).
```

CLP(R) - constraints over reals

```
:– use_module(library(clpr)).
```

```
mg(P, T, I, B, MP) :-  
{ T = 1,  
  B + MP = P * (1 + I)  
 }.
```

```
mg(P, T, I, B, MP) :-  
{ T > 1,  
  P1 = P * (1 + I) - MP,  
  T1 = T - 1  
 },  
 mg(P1, T1, I, B, MP).
```

```
?– mg(1000, 30, 5/100, B, 0).
```

B = 4321.9423751506665

Mortgage relation between the following arguments:

- P is the balance at T_0
- T is the number of interest periods (e.g., years)
- I is the interest ratio where e.g., 0.1 means 10%
- B is the balance at the end of the period
- MP is the withdrawal amount for each interest period.

Back to Standard Prolog:

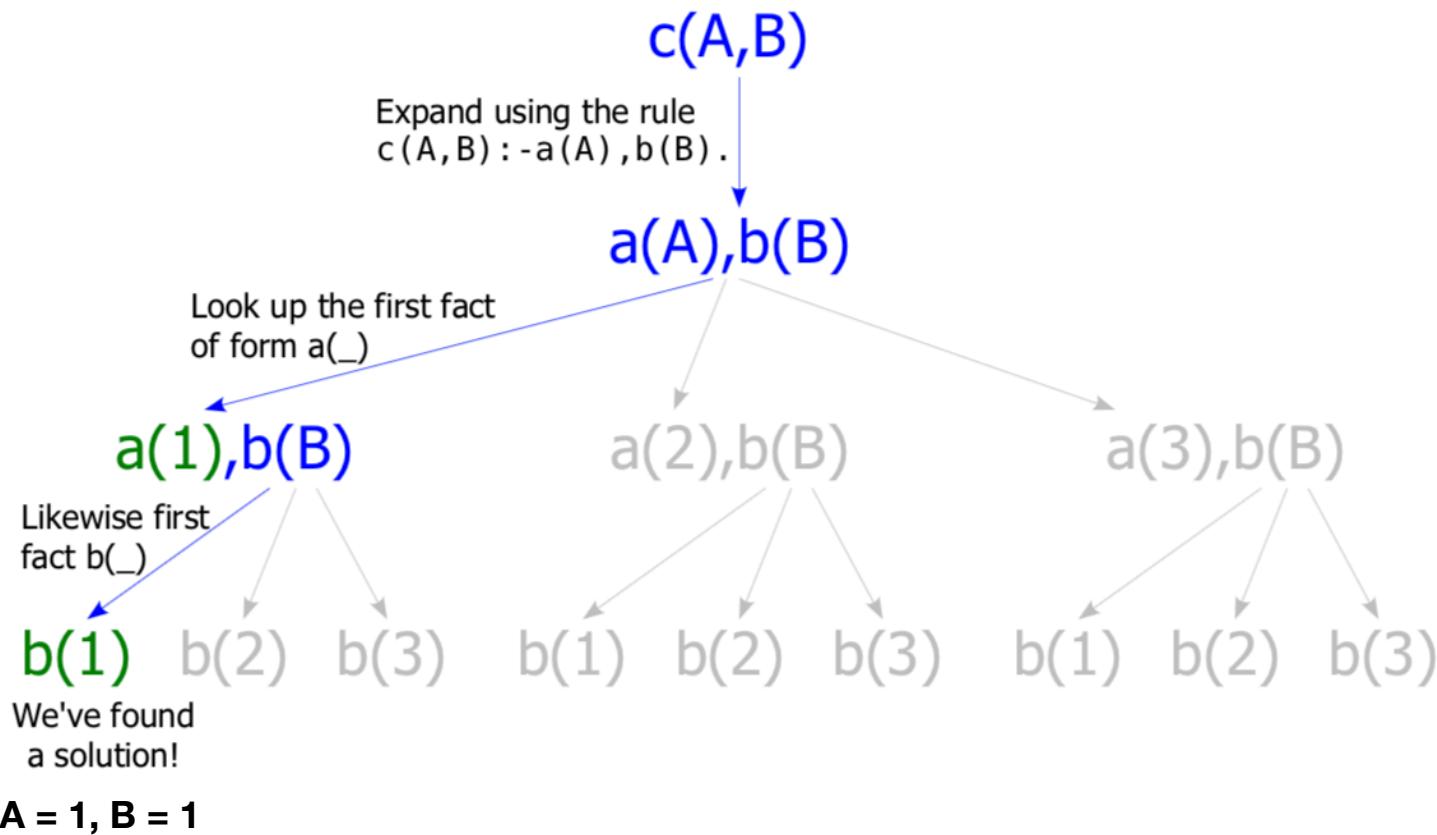
Controlling Execution

Prolog – Finding Answers

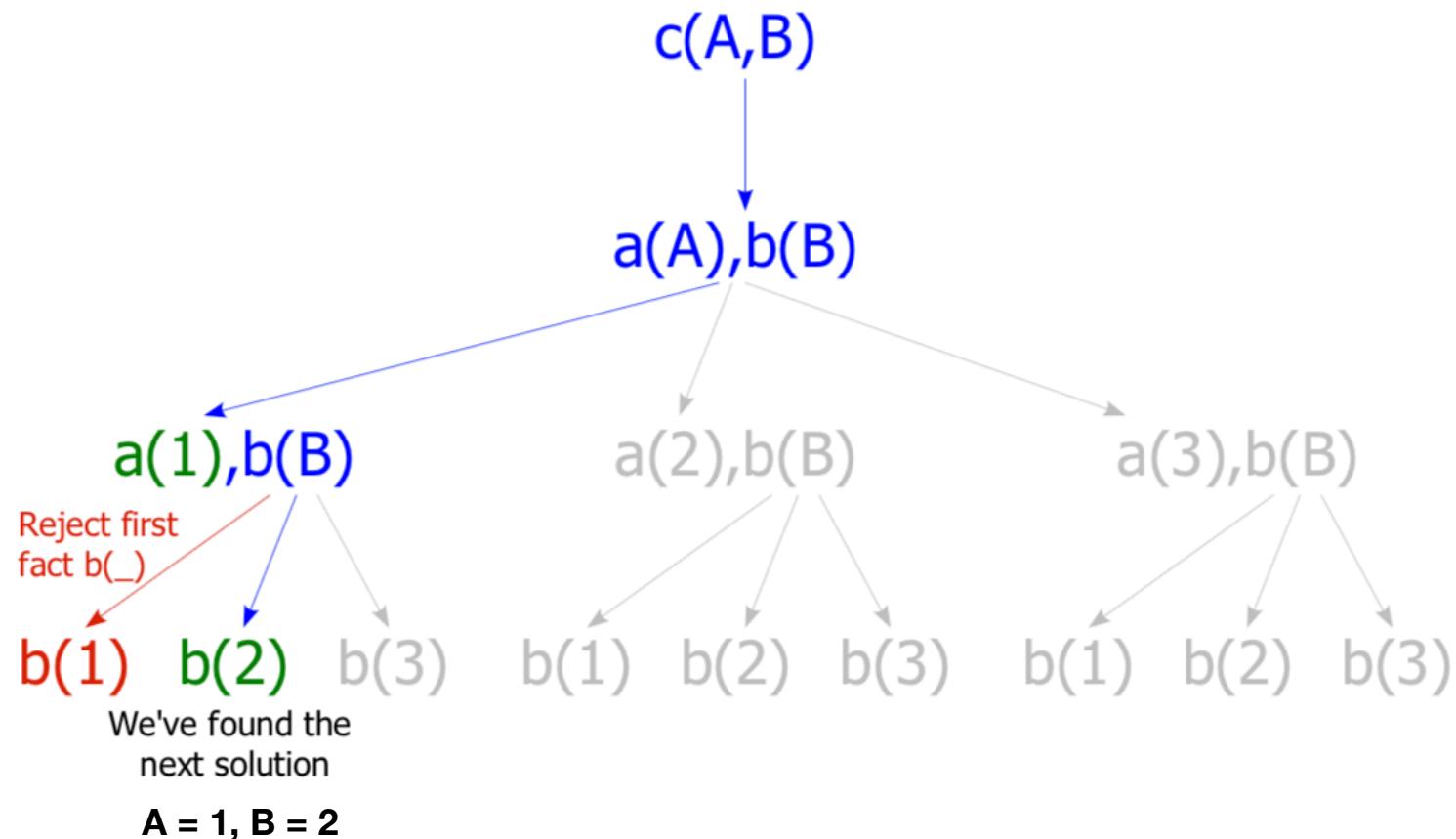
Prolog uses depth first search to find answers

```
a(1).  
a(2).  
a(3).  
b(1).  
b(2).  
b(3).  
  
c(A, B) :- a(A), b(B).
```

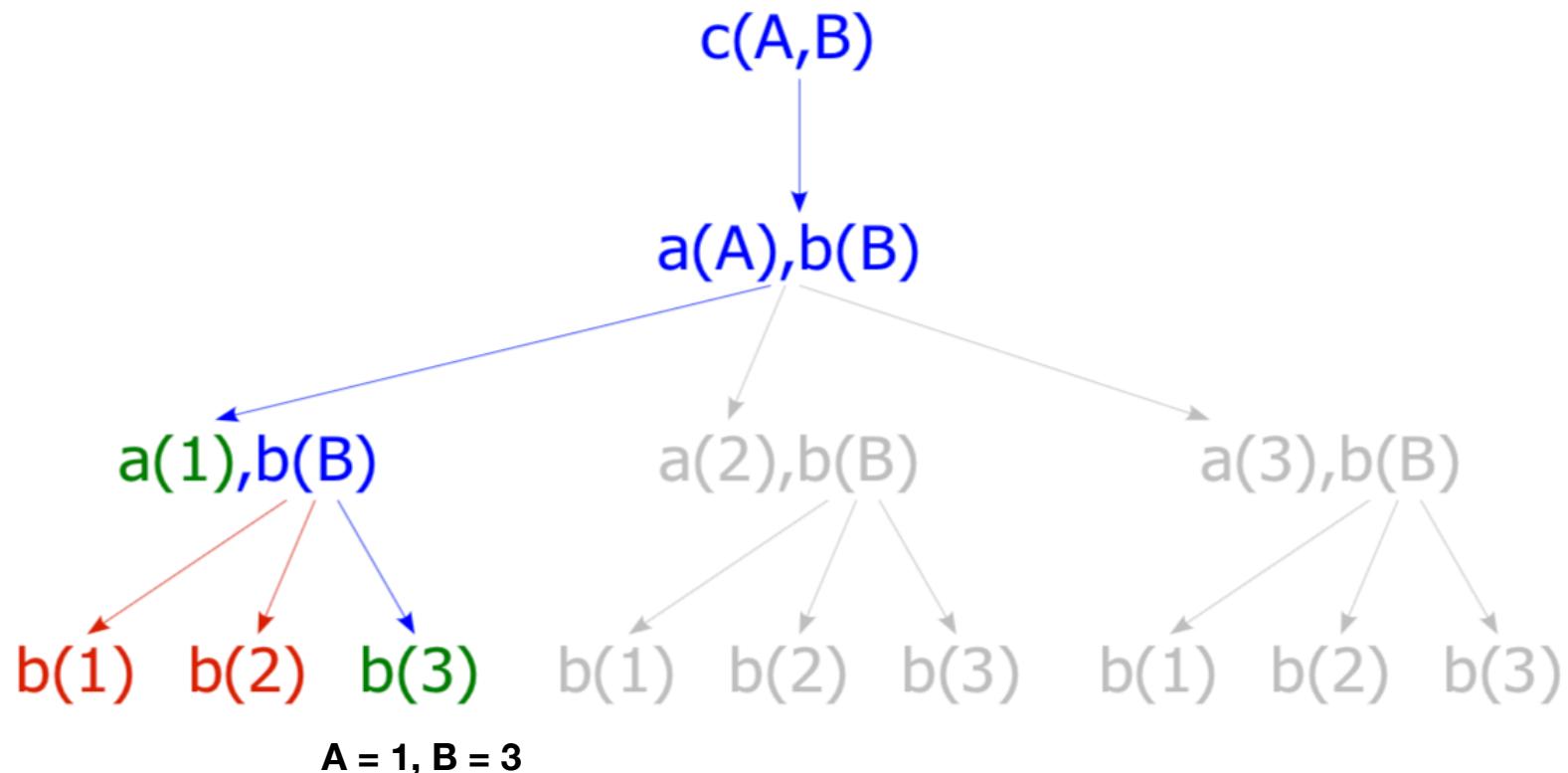
Depth-first solution of query $c(A,B)$



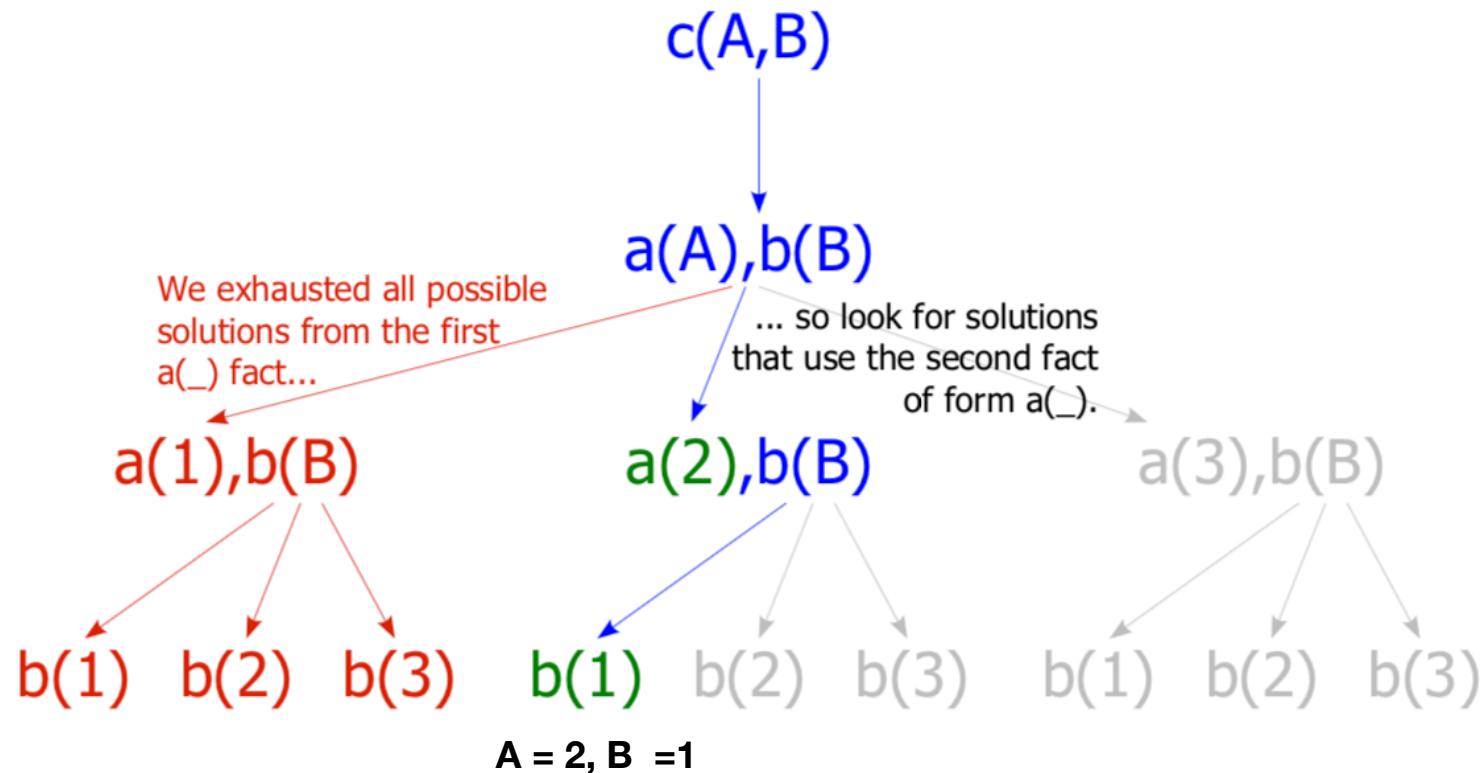
Backtrack to find another solution



Backtrack to find another solution



Backtrack to find another solution



The Cut (!)

- Sometimes we need a way of preventing Prolog from finding all solutions
- The *cut* operator is a built-in predicate that prevents backtracking
- It violates the declarative reading of a Prolog programming
- Use it *VERY sparingly!!!*

Backtracking

```
lectures(maurice, Subject), studies(Student, Subject)?
```

```
Subject = 1021
```

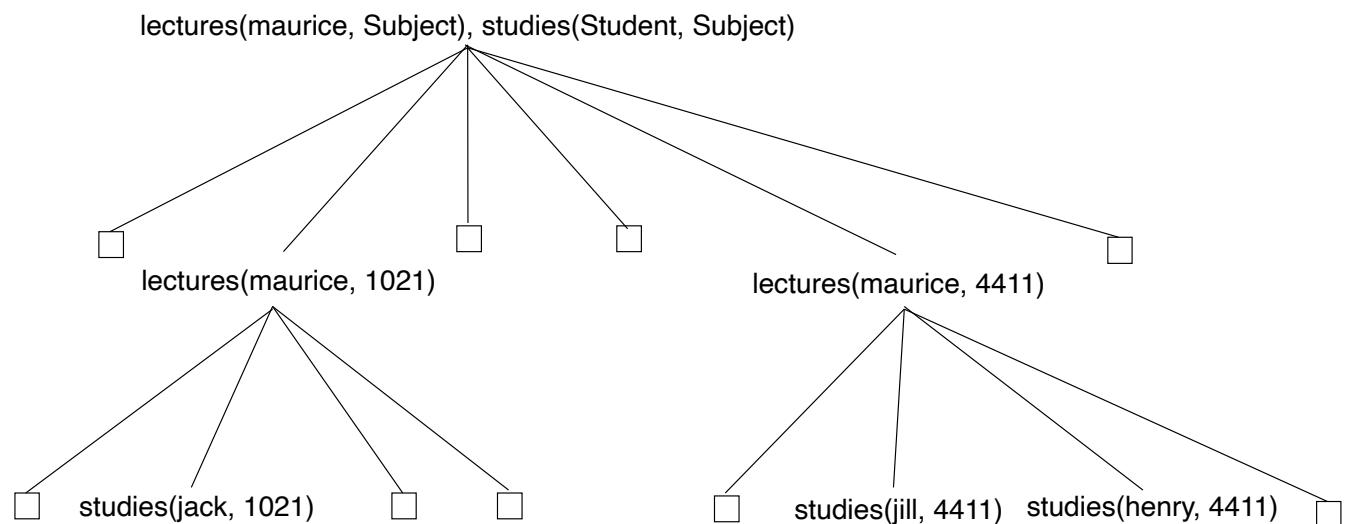
```
Student = jack ;
```

```
Subject = 4411
```

```
Student = Jill ;
```

```
Subject = 4411
```

```
Student = Henry
```



Cut prunes the search

```
lectures(maurice, Subject), !, studies(Student, Subject)?
```

```
Subject = 1021
```

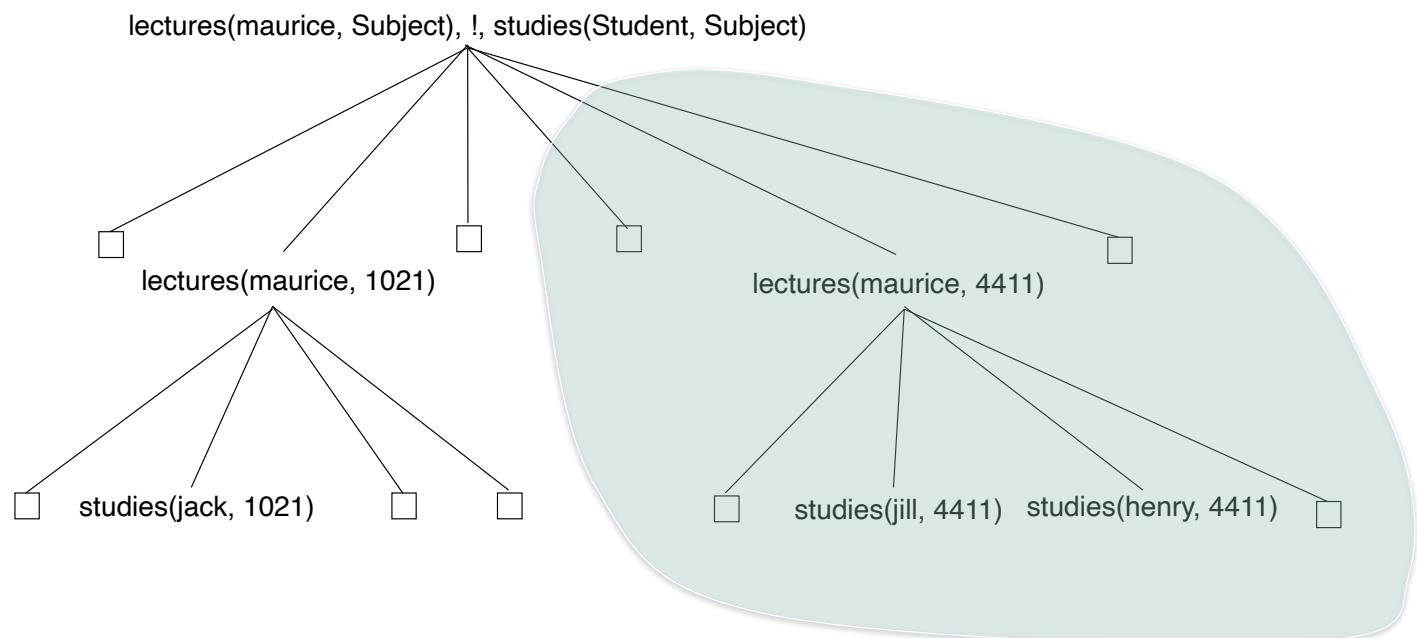
```
Student = jack ;
```

```
Subject = 4411
```

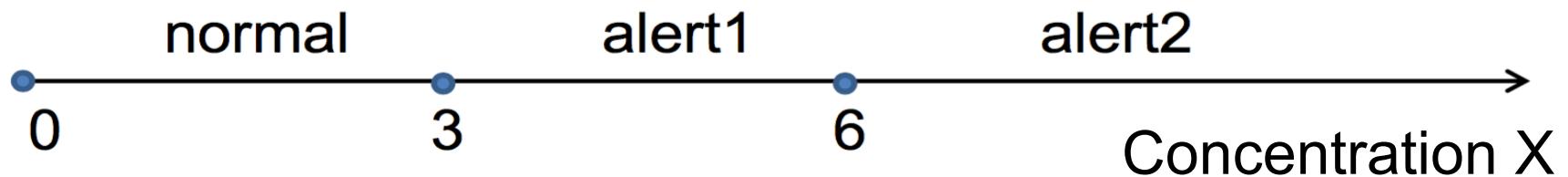
```
Student = Jill ;
```

```
Subject = 4411
```

```
Student = Henry
```



Example



Rules for determining the degree of pollution

Rule 1: if $X < 3$ then $Y = \text{normal}$

Rule 2: if $3 \leq X$ and $X < 6$ then $Y = \text{alert1}$

Rule 3: if $6 \leq X$ then $Y = \text{alert2}$

In Prolog: **f(Concentration, Pollution_Alert)**

```
f(X, normal) :- X < 3.                      % Rule1
f(X, alert1) :- 3 =  
= X, X < 6.                    % Rule2
f(X, alert2) :- 6 =  
= X.                      % Rule3
```

Alternative Version

```
f(X, normal) :- X < 3, !.          % Rule1
f(X, alert1) :- X < 6, !.          % Rule2
f(X, alert2).                      % Rule3
```

Which version is easier to read?

Operators

Operator Notation

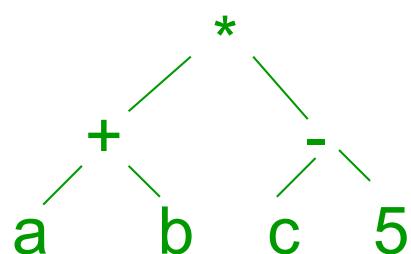
- Operators are just compound (i.e. functional) terms

$$2*a + b*c = +(* (2 , a) , * (b , c))$$

- $+$, $*$ are infix operators in Prolog
 - They are only interpreted as arithmetic expressions when they appear on the right-hand side of the *is* operator.

Operator Expressions are also Trees

- For example: $(a + b) * (c - 5)$
- Written as an expression with the functors:

$$*(+ (a, b), - (c, 5))$$


Operators in Prolog

- You can define your own operators.

```
:‐ op(Precedence, Type, Name).
```

- Precedence is a number between 0 and 1200.

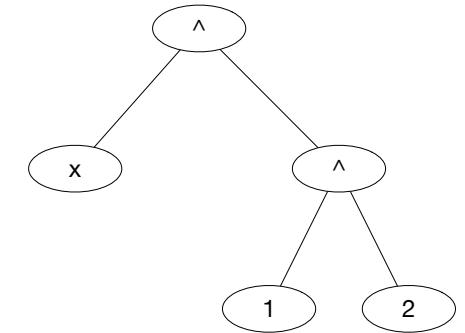
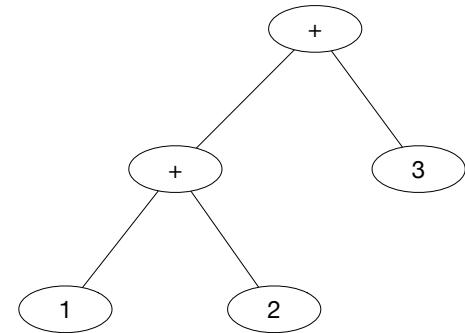
For example,

- the precedence of “ = ” is 700,
- the precedence of “ + ” is 500,
- the precedence of “ * ” is 400.

Operators in Prolog

```
:  
- op(Precedence, Type, Name).
```

- Type is an atom specifying the associativity of the operator.
- Infix operators:
 - yfx - left associate (e.g. $1 + 2 + 3 = ((1 + 2) + 3)$)
 - xfy - right associative (e.g. $x \wedge 2 \wedge 2 = (x \wedge (2 \wedge 2))$)
 - xfx - non-associative (e.g. wa = green; a = b = c is not valid)
- Prefix operators
 - fy, fx (associative, non-associative)
- Postfix operators
 - yf, xf (associative, non-associative)



Predefined Operators

- Operators with the same properties can be specified in one statement by giving a list of their names instead of a single name as third argument of op.
- Operator definitions don't specify the meaning of an operator, only how it can be used syntactically.
- Operator definition doesn't say how a query involving operator is evaluated to true.

```
:- op(1200, xfx, [:-, ->]).  
:- op(1200, fx, [:-, ?-]).  
:- op(1100, xfy, [ ; ]).  
:- op(1000, xfy, [',']).  
:- op( 700, xfx, [=, is, =.., ==. \==, \==, =:=, =\=, <, >, =<, >=] ).  
:- op( 500, yfx, [+,-]).  
:- op( 500, fx, [+,-]).  
:- op( 300, xfx, [ mod ]).  
:- op( 200, xfy, [ ^ ] ).
```

User Defined Operators

Relations can be defined as operators, e.g.

```
has(peter, information).  
supports(floor, table).
```

can be written with operators:

```
:- op(600, xfx, has).  
:- op(600, xfx, supports).
```

```
peter has information.  
floor supports table.
```

Example - IF statement

```
% Operator declarations
:- op(500, fx, if).
:- op(400, xfx, then).
:- op(300, xfx, else).

% Interpreter

if Condition then S1 else S2 :-
    Condition, !, S1.                      % Don't allow backtracking if Condition is true
if Condition then S1 else S2 :-
    S2.
```

Built-in Predicates

- Testing the type of terms
- Construction and decomposition of terms: =. . . , functor, arg, name
- Comparison
- *bagof*, *setof* and *findall*
- Input, output

Testing the type of terms

var(X)	true if X is unbound or instantiated to an unbound variable
nonvar(X)	X is not a variable or instantiated to an unbound variable
atom(X)	true if X is an atom
integer(X)	true if X is an integer
float(X)	true if X is a real number
number(X)	true if X is a number
atomic(X)	true if X is a number or an atom
compound(X)	true if X is a compound term (a structure)
is_list(X)	true if X is [] or a non-empty list

Example: Arithmetic Operations

...,

number(X), % Value of X number?

number(Y), % Value of Y number?

Z is X + Y, % Then addition is possible

...

Construction and decomposition of terms:

$=.. , functor, arg, name$

Term =.. [Functor, Arg1, Arg2, Arg3, ...] % “univ”

Term =.. L

true if L is a list that contains the principal functor of **Term**, followed by its arguments.

Example:

?- f(a, b) =.. L.

L = [f, a, b]

?- T =.. [rectangle, 3,5].

T = rectangle(3, 5)

Construction and decomposition of terms: $=..$, functor, arg, name

```
?- functor(a(), N, A).  
N = a, A = 0.
```

```
?- functor(T, a, 0).  
T = a.
```

```
?- arg(2, f(a, b), X).  
X = b.
```

```
?- arg(N, f(a, b), V).  
N = 1, V = a ;  
N = 2, V = b.
```

Substitute

```
substitute(Subterm, Term, Subterm1, Term1)
```

Find all occurrences of Subterm in Term and substitute with Subterm1 to get Term1.

```
?- substitute(sin(x), 2*sin(x)*f(sin(x)), t, F).
```

```
F = 2*t*f(t)
```

Substitute

```
% Case 1: Substitute whole term
substitute(Term, Term, Terml, Terml) :- !.

% Case 2: Nothing to substitute if Term atomic
substitute(_, Term, _, Term) :-
    atomic(Term), !.                                % Term is a constant

% Case 3: Do substitution on arguments
substitute(Sub, Term, Sub1, Terml) :-
    Term =.. [F | Args],                          % Get arguments
    substlist(Sub, Args, Sub1, Args1),            % Perform substitution on them
    Terml =.. [F | Args1].                      % Construct Terml

% substlist(SubTerm, Term_List, NewSubTerm, NewTerm_List)
```

Substitute

```
% substlist(SubTerm, Term_List, NewSubTerm, NewTerm_List)

% End of list, nothing to do
substlist(_, [], _, []).

% Otherwise, try substituting recursively
substlist(A, [X|List], B, [Y|List1]) :-
    substitute(A, X, B, Y),
    substlist(A, List, B, List1).
```

Example - Use of *substitute* / 4

```
?- E0 = (a+b) * (a-b),  
       substitute(a, E0, 6, E1),  
       substitute(b, E1, 3, E2),  
       Value is E2.
```

```
E1 = (6+b) * (6-b)  
E2 = (6+3) * (6-3)  
Value = 27
```

Comparison

$X = Y$ true if X and Y match

$X == Y$ if X and Y are identical

$X \== Y$ if X and Y are not identical

$X @< Y$ X is lexicographically smaller than Y, term X precedes term Y
by alphabetical or numerical ordering
(e.g. paul @< peter)

findall, bagof, setof

```
% Find all values of Object that satisfy Condition and collect in List
findall(Object, Condition, List)

% Same as findall except only stores unique values and fails if no solution found
bagof(Object, Condition, List)

% Find all values of Object that satisfy Condition and collect in sorted List
setof(Object, Condition, List)
```

Example: robot world

```
?- findall(B, on(B,_), L).                                % L is a List of all blocks
L = [a,b,c,d,e]
```

Procedure *findall*, *bagof*, *setof*

Examples:

```
child(joze, ana).      child(miha, ana).
child(lili, ana).      child(lili, andrej).
```

```
?- findall(X, child(X, ana), S).
S = [joze, miha, lili]
```

```
?- setof(X, child(X, ana), S).
S = [joze, lili, miha]
```

```
?- findall(X, child(X, Y), S).
S = [joze, miha, lili, lili]
```

```
?- bagof(X, child(X, Y), S).
S = [joze, miha, lili]
Y = ana;
```

Input / Output

```
consult(File)      % Load File into Prolog's database  
  
see(File)         % File becomes the current input stream  
  
see(user)         % user input (i.e. from terminal)  
  
seen              % close the current input stream  
  
seeing(X)         % binds X to the current input file  
  
tell(File)         % File becomes the current output stream  
  
tell(user)         % user output (i.e. output to terminal)  
  
told              % close the current output stream  
  
telling(X)        % binds X to the current output file
```

Input / Output

`write(Term) % write Term to current output stream`

`writeln(Term) % write Term and append newline`

`nl % write newline to current output stream`

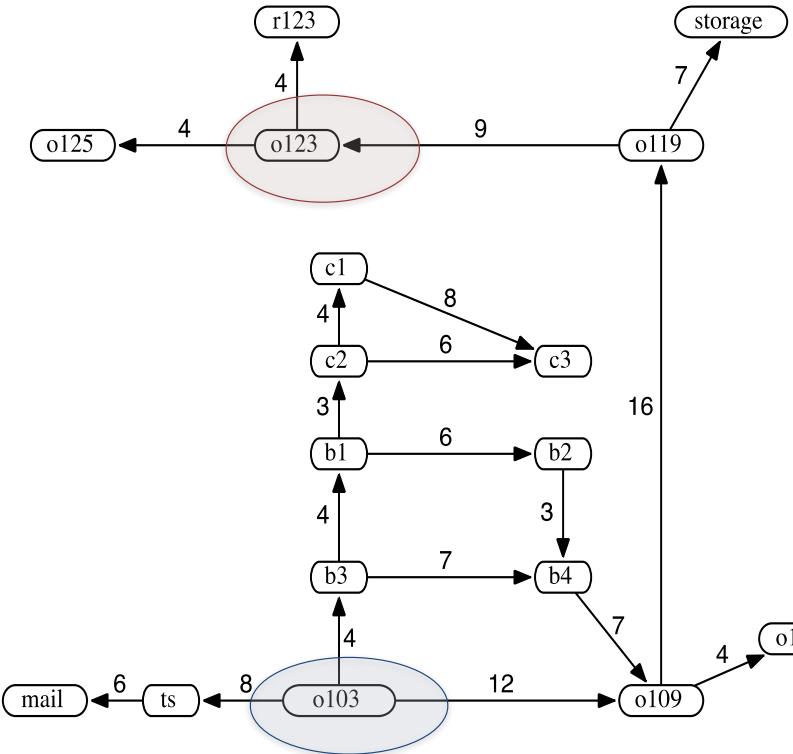
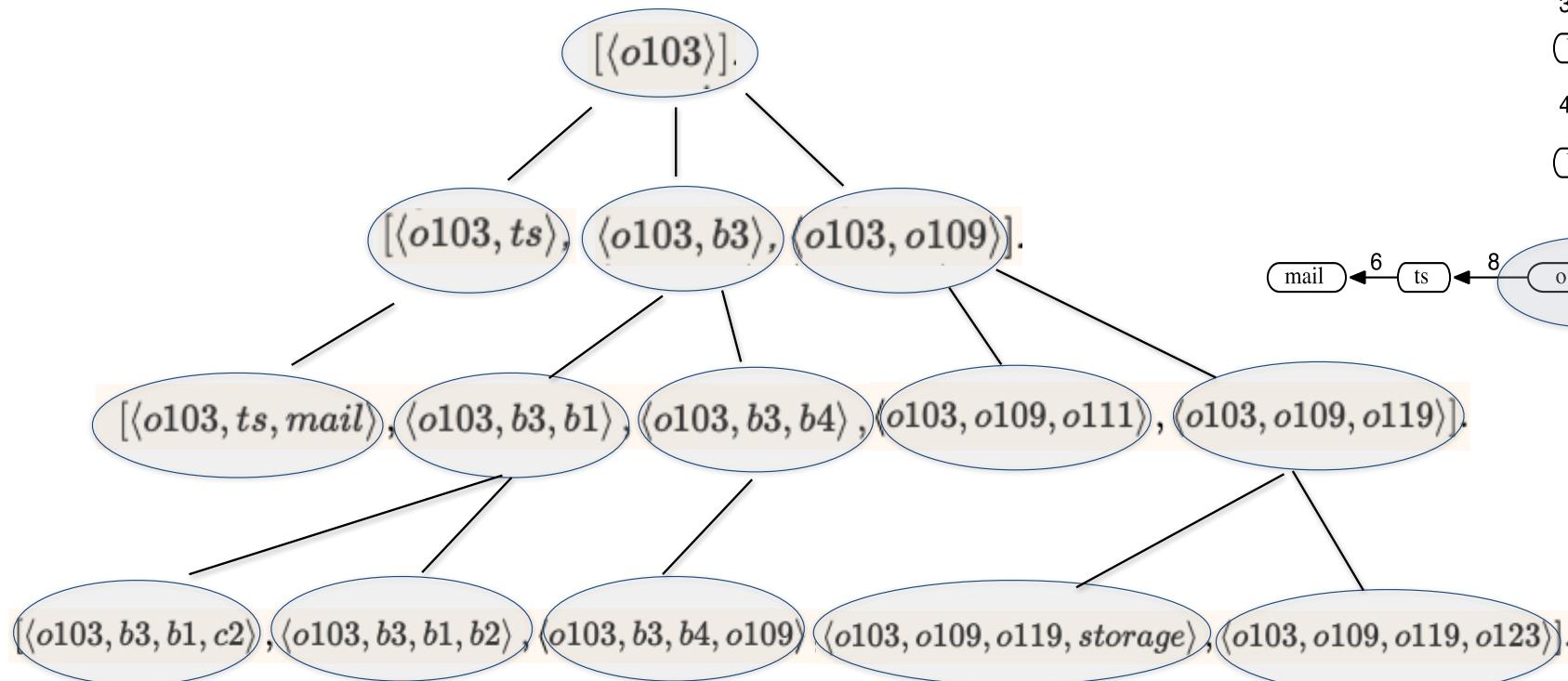
`read(Term) % read Term from current input stream`

SWI Prolog Manual

There is a lot more to learn in the user manual:

https://www.swi-prolog.org/pldoc/doc_for?object=manual

Breadth-First Search



Each path on the frontier has either the same number of arcs or one more arc than the next element of the frontier that will be selected.

Breadth-First Search

```
solve(Start, Solution)  :- breadthfirst([[Start]], Solution).

breadthfirst([[Node|Path] | _], [Node|Path]) :- goal(Node).
breadthfirst([Path|Paths], Solution)  :-
    extend(Path, NewPaths),
    append(Paths, NewPaths, Paths1),
    breadthfirst(Paths1, Solution).

extend([Node|Path], NewPaths)  :-
    findall([Neighbour, Node|Path], new_neighbour([Neighbour, Node|Path]), NewPaths).

new_neighbour([Neighbour, Node|Path]) :- 
    edge(Node, Neighbour),
    \+ member(Neighbour, [Node|Path]).
```

Knowledge Representation

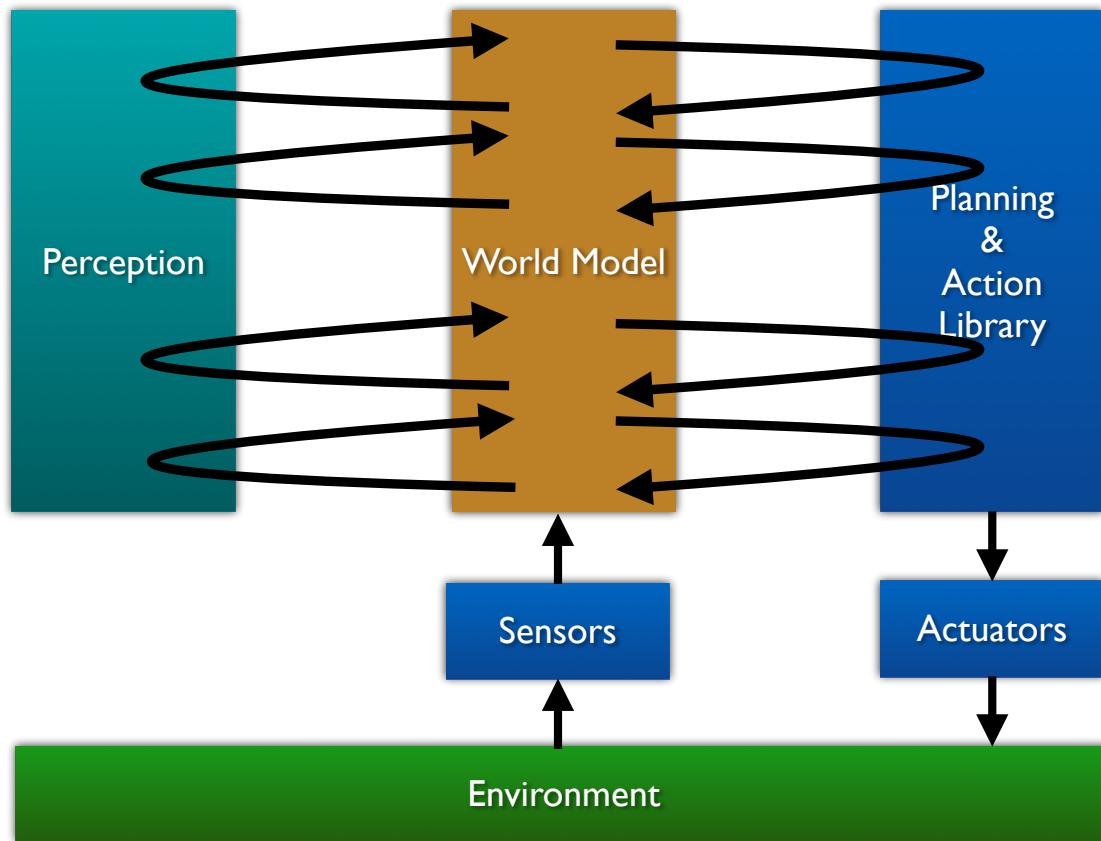
COMP3411/9814: Artificial Intelligence

Lecture Overview

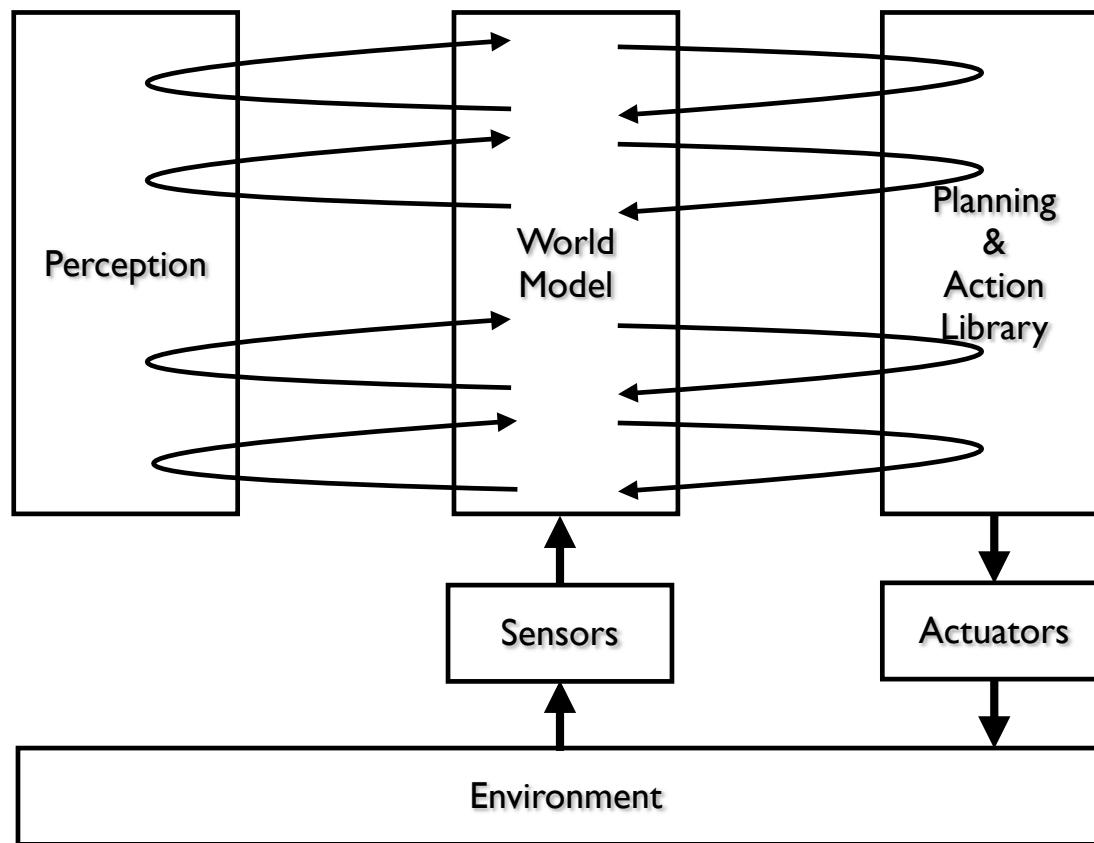
- Cognitive Architectures
- The Knowledge Level
- Knowledge Representation
- Ontologies, Taxonomy, Categories and Objects
- Semantic Networks
- Rule based representation
- Inference Networks
- Deduction, Abduction, and Induction

Cognitive Architectures

Nilsson's Triple Tower



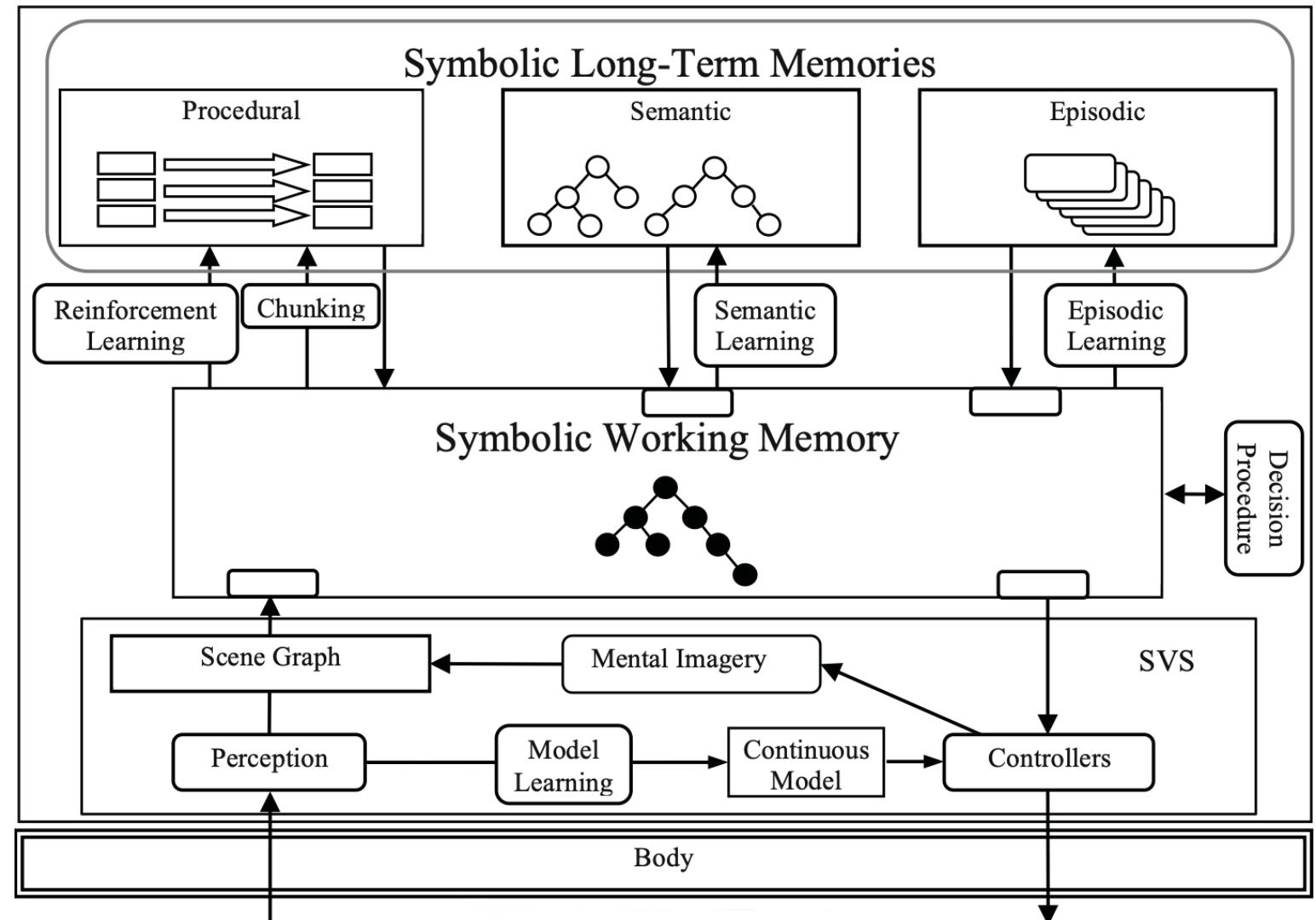
Nilsson's Triple Tower



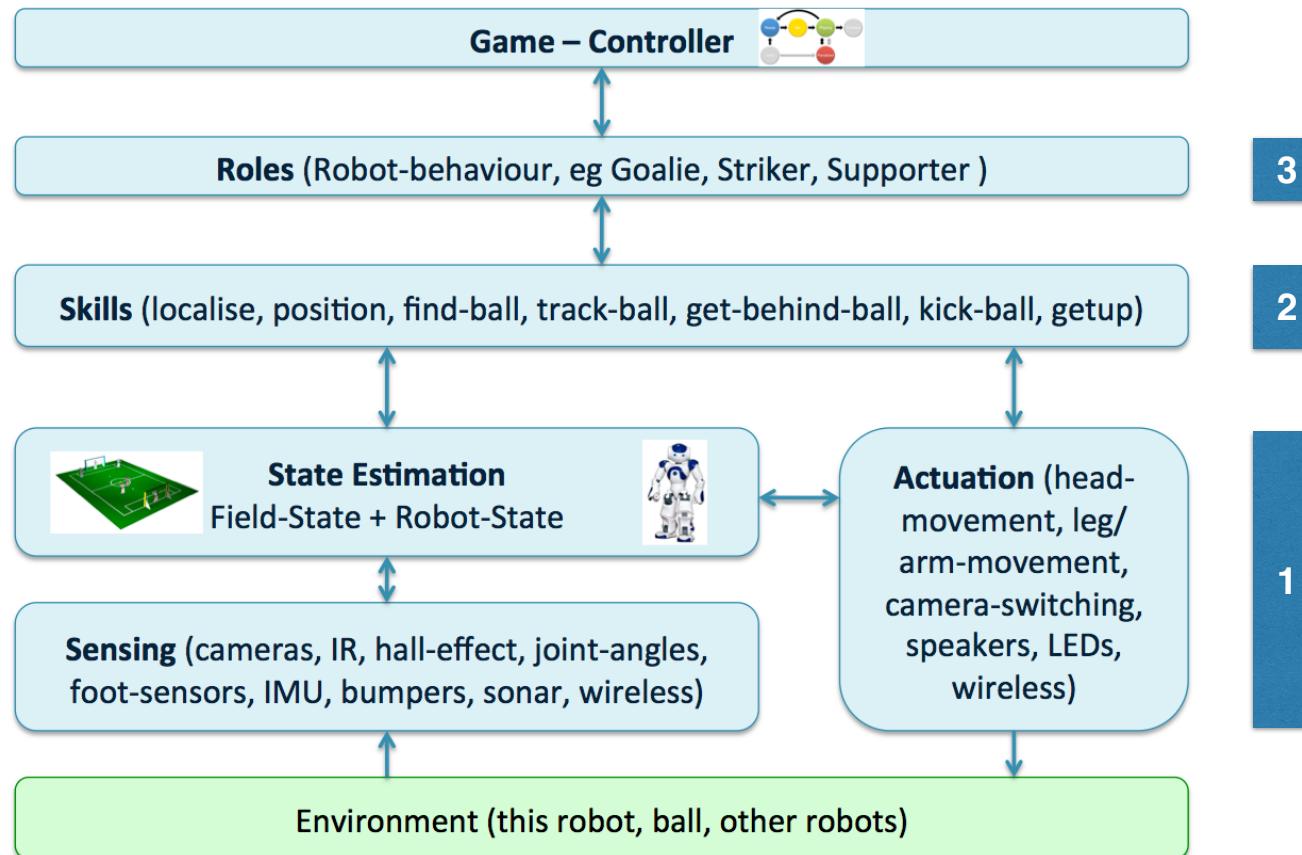
SOAR



Laird, J. E., Kinkade, K. R., Mohan, S., & Xu, J. Z. (2012). Cognitive Robotics Using the Soar Cognitive Architecture (pp. 46–54). In *International Conference on Cognitive Robotics, (Cognitive Robotics Workshop, Twenty-Sixth Conference on Artificial Intelligence (AAAI-12), Toronto.*



A Three-Level Architecture



“Classical” AI

Symbolic Representations and Reasoning

The Physical Symbol System Hypothesis

"A physical symbol system has the [necessary and sufficient means](#) for general intelligent action."^[1]

— [Allen Newell](#) and [Herbert A. Simon](#)

Criticisms:

- Lacks “symbol grounding” (what does a symbol refer to?).
- AI requires non-symbolic processing (e.g. connectionist architecture).
- Brain is not a computer and computation is not an appropriate model for intelligence.
- Brain is mindless - mostly chemical reactions
 - human intelligent behaviour is analogous to behaviour displayed by ant colonies

The Knowledge Level

- **Knowledge Level Hypothesis.** There exists a distinct computer systems level, lying immediately above the symbol level, which is characterised by knowledge as the medium and the principle of rationality as the law of behaviour.
- **Principle of Rationality.** If an agent has knowledge that one of its actions will lead to one of its goals, then the agent will select that action.
- **Knowledge.** Whatever can be ascribed to an agent, such that its behaviour can be computed according to the principle of rationality.

“The Knowledge Level” (Newell, 1982)

Knowledge Representation

- Any agent can be described on different levels:
 - Knowledge level (knowledge ascribed to agent)
 - Logical level (algorithms for manipulating knowledge)
 - Implementation level (how algorithms are implemented)
- Knowledge Representation is concerned with expressing knowledge **explicitly** in a **computer-tractable** way (i.e. for reasoning)
- **Reasoning** takes knowledge and draw inferences
 - answer queries, determine facts that follow from the knowledge, decide what to do, etc.

Knowledge Representation and Reasoning

- A knowledge-based agent has at its core a **knowledge** base
- A knowledge base is an explicit set of **sentences** about some domain expressed in a suitable formal representation language
- Sentences express facts (**true**) or non-facts (**false**)
- Fundamental Questions
 - How do we write down knowledge about a domain/problem?
 - How do we automate reasoning to deduce new facts or ensure consistency of a knowledge base?

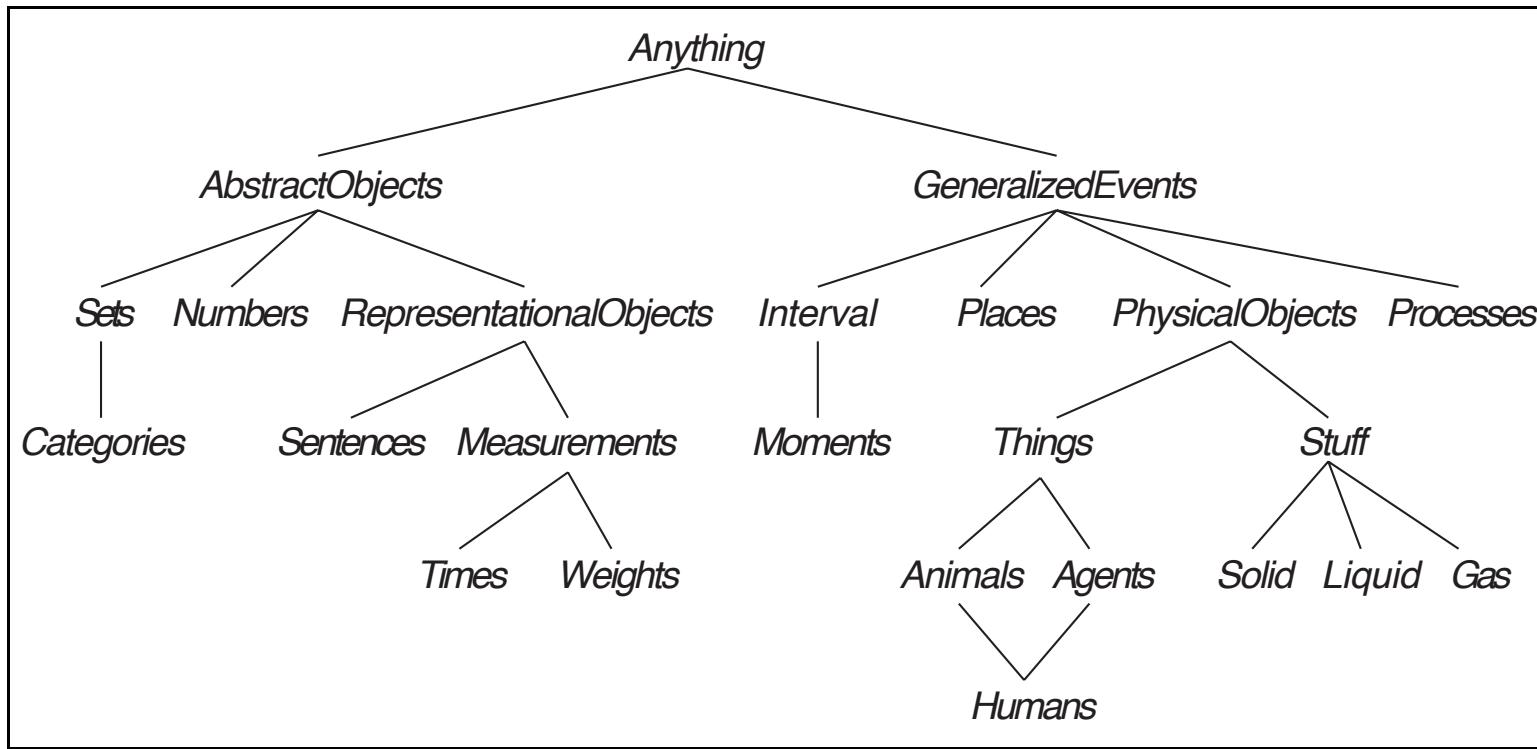
Knowledge representation

- We study the technology for knowledge-based agents:
 - syntax, semantics and proof theory of propositional and first-order logic
 - the implementation of agents that use these logics.
- Also need to address question:
 - What *content* to put into such an agent's knowledge base?
 - how to represent facts about the world.

Ontologies and Ontology Engineering

- An [ontology](#) organises everything into a hierarchy of categories.
- Can't actually write a complete description of everything
 - far too much
 - can leave placeholders where new knowledge can fit in.
 - define what it means to be a physical object
 - details of different types of objects (robots, televisions, books, ...) filled in later
- Similar to Object Oriented programming framework

Ontology Example



- Child concept is a specialisation of parent
- Specialisations are not necessarily disjoint (a human is both an animal and an agent)

Ontology Example

- Equality: Scott Morrison is Prime Minister Morrison
- Role: Scott Morrison is Prime Minister of Australia
- Part of: Scott Morrison is in the government
- A kind of: NSW is a state
- Part of: NSW is in Australia
- Vaccination implies Medical treatment – linguistic meaning/semantics
- Ontology = Set of such facts

Categories and Objects

- Organising objects into categories is vital for knowledge representation.
- Interaction with world takes place at level of individual objects, but ...
 - much reasoning takes place at level of categories
- Categories help make predictions about objects once they are classified
- Two choices for representing categories in first-order logic:
 - predicates and objects.

Categories and Objects

- Infer presence **objects** from perceptual input
- Infer category membership from perceived properties of the objects
- Use category information to make predictions about the objects
 - green and yellow mottled skin & 30cm diameter & ovoid shape & red flesh, black seeds & presence in the fruit aisle → watermelon
 - from this, infer that it would be useful for fruit salad.

Categories and Objects

- Categories organise and simplify knowledge base through inheritance.
 - if all instances of category Food are edible, and
 - if Fruit is a subclass of Food and Apples is a subclass of Fruit, then
 - infer that every apple is edible.
- Individual apples inherit property of edibility
 - in this case from membership in the Food category.

Taxonomic hierarchy

- Subclass relations organise categories into a [taxonomy](#), or [taxonomic hierarchy](#)
- Taxonomies have been used explicitly for centuries in technical fields.
 - Taxonomy of living things organises about 10 million living and extinct species
 - Library science has developed a taxonomy of all fields of knowledge
- Taxonomies are also an important aspect of general [commonsense knowledge](#)

Categories and Objects and FOL

- First-order logic can state facts about categories, relating objects to categories or by quantifying over members.
- An object is a member of a category

$BB_9 \in Basketballs$

- A category is a subclass of another category

$Basketballs \subset Balls$

- All members of a category have some properties

$(\forall x)(x \in Basketballs \implies Spherical(x))$

- Members of a category can be recognised by some properties

$Orange(x) \wedge Round(x) \wedge Diameter(x) = 24cm \wedge x \in Balls \implies x \in Basketballs$

- A category as a whole has some properties

$Dogs \in DomesticatedSpecies$

Prolog:

```
basketball(X) :-  
    orange(X),  
    round(X),  
    diameter(X, 24),  
    ball(X).
```

Reasoning system for categories

- Categories are the building blocks of knowledge representation schemes
- Two closely related families of systems:
 - semantic networks:
 - graphical aids for visualizing a knowledge base
 - efficient algorithms for inferring properties of object based in category membership
 - description logics:
 - formal language for constructing and combining category definitions
 - efficient algorithms for deciding subset and superset relationships between categories

Semantic Networks

- Fact , Objects, Attributes and Relationships
 - Relationships exist among instances of objects and classes of objects.
- Attributes and relationships can be represented as a network, known as an **associative network** or **semantic network**
- We can build a model of the subject area of interest

Semantic networks

- In 1909, Charles S. Peirce proposed a graphical notation of nodes and edges called **existential graphs** that he called “the logic of the future.”
- Long-running debate between advocates of “logic” and “semantic networks”:
 - semantics networks with well-defined semantics are a form of logic.
 - notation provided by semantic networks for certain kinds of sentences is often more convenient,
 - underlying concepts
 - objects, relations, quantification, and so on...
 - are the same as in logic.

Knowledge and Semantic Networks

- Facts can be:
 - **static** - can be written into the knowledge base.
static facts need not be permanent, but change infrequently so changes can be accommodated by updating the knowledge base when necessary.
 - **transient** - apply at a specific instance only or for a single run of system

Knowledge and Semantic Networks

- Important aspect of semantic networks - can represent **default values** for categories
- Knowledge base may contain *defaults* that can be used as facts in the absence of transient facts

Example – A simple set of statements

- My car is a car
- A car is a vehicle
- A car has four wheels
- A car's speed is 0 mph
- My car is red
- My car is in my garage
- My garage is a garage
- A garage is a building
- My garage is made from brick
- My car is in High Street
- High Street is a street
- A street is a road

**Underline = object (instance)
Everything else is a category (class)**

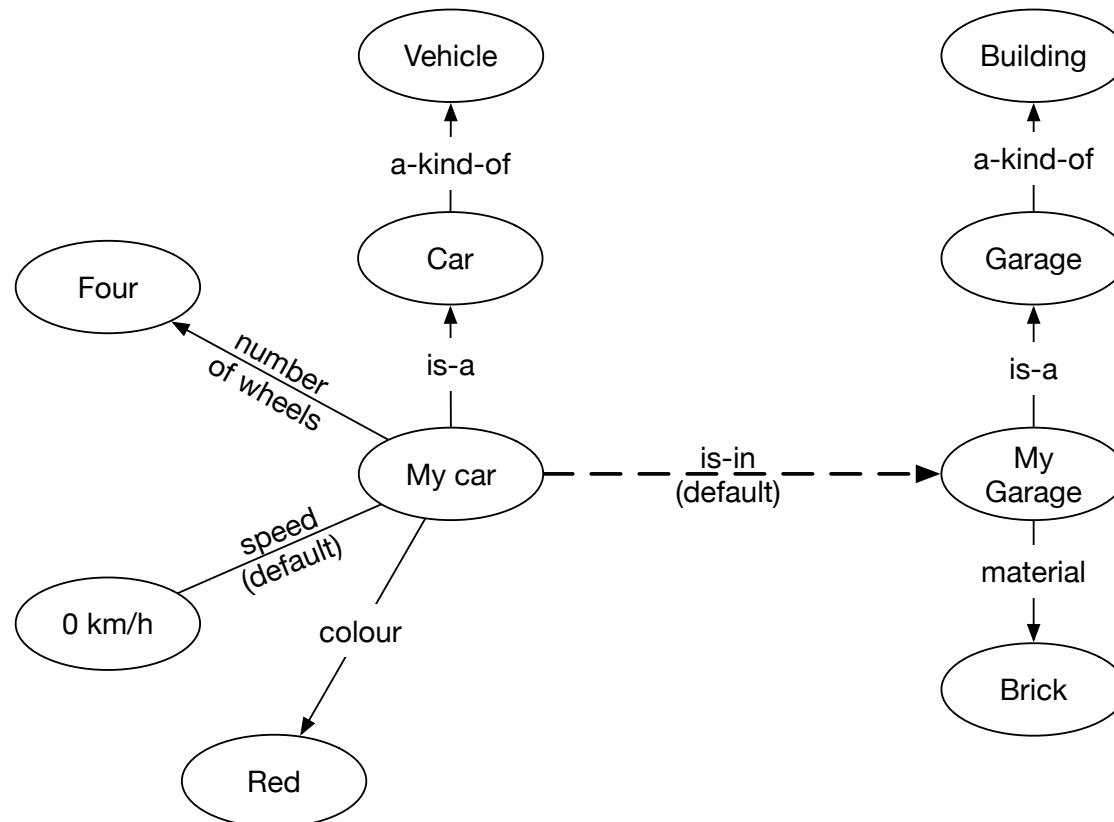
Example – facts, objects and relations

- My car **is a car**
- A car **is a vehicle**
- A car **has four wheels**
- A car's **speed is 0 mph**
- My car **is red**
- My car **is in my garage**
- My garage **is a garage**
- A garage **is a building**
- My garage **is made from brick**
- My car **is in High Street**
- High Street **is a street**
- A street **is a road**

Example – facts, objects and relations

- My car **is a car** (static relationship)
- A car **is a vehicle** (static relationship)
- A car **has four wheels** (static attribute)
- A car's **speed is 0 mph** (default attribute)
- My car **is red** (static attribute)
- My car **is in my garage** (default relationship)
- My garage **is a garage** (static relationship)
- A garage **is a building** (static relationship)
- My garage **is made from brick** (static attribute)
- My car **is in High Street** (transient relationship)
- High Street **is a street** (static relationship)
- A street **is a road** (static relationship)

A semantic network (with a default)



Classes and Instances

- Distinction between object instances and classes of objects:
 - Car and vehicle are both classes of objects
 - Linked by “ako” relation (a-kind-of)
 - Direction of arrow indicates “car is a vehicle” and not “vehicle is a car”
 - *My car* is a unique entity.
 - Relationship between *my car* and *car* is “isa” (is an instance of)

Semantic Networks - Reasoning

- Inheritance is good for default reasoning weak otherwise
- Extend by *procedural attachment*
 - **Frames:** Demons are triggered when attributes if instances are added, deleted or modified
 - **Agents:** Contain **goals** and **plans** and run as concurrently
 - **Objects:** Methods an implement attached procedures

Rule-Based Systems

- A **production rule** and has the form

```
if <condition> then <conclusion>
```

- Production rule for dealing with the payroll of ACME, Inc., might be

```
rule r1_1  
if the employer of Person is acme  
then the salary of Person becomes large.
```

Rule-Based Systems

```
rule r1_1
```

```
if the employer of Person is acme
```

```
then the salary of Person becomes large.
```

- Production rules can often be written to closely resemble natural language

```
/* fact f1_1 */
```

```
the employer of joe_bloggs is acme.
```

- Capitalisation (like Prolog) indicates that “Person” is a variable that can be replaced by a constant, such as “joe_bloggs” or “mary_smith”, through pattern matching.

Rule-Based Systems

```
rule r1_1
```

```
if the employer of Person is acme
```

```
then the salary of Person becomes large.
```

```
rule r1_2
```

```
if the salary of Person is large
```

```
or the job_satisfaction of Person is true
```

```
then the professional_contentment of Person becomes true.
```

- Executing a rule may generate a new derived fact.
- There is a *dependency* between rules `r1_1` and `r1_2` since the conclusion of one can satisfies the condition of the other.

Uncertainty in rules

- Rules can express many types of knowledge
- But how can *uncertainty* be handled?
- Uncertainty may arise from:
 - Uncertain evidence (Not certain that Joe Bloggs works for ACME.)
 - Uncertain link between evidence and conclusion.
(Cannot be certain that ACME employee earns a large salary, just likely.)
 - Vague rule. (What is a “large”?)

Fuzzy Logic

Bayesian inference

Rule-Based Systems

```
rule r1_1

if the employer of Person is acme

then the salary of Person becomes large.
```

```
/* fact f1_1 */

the employer of joe_bloggs is acme.

/* derived fact f1_2 */

the salary of joe_bloggs is large.
```

```
rule r1_2

if the salary of Person is large

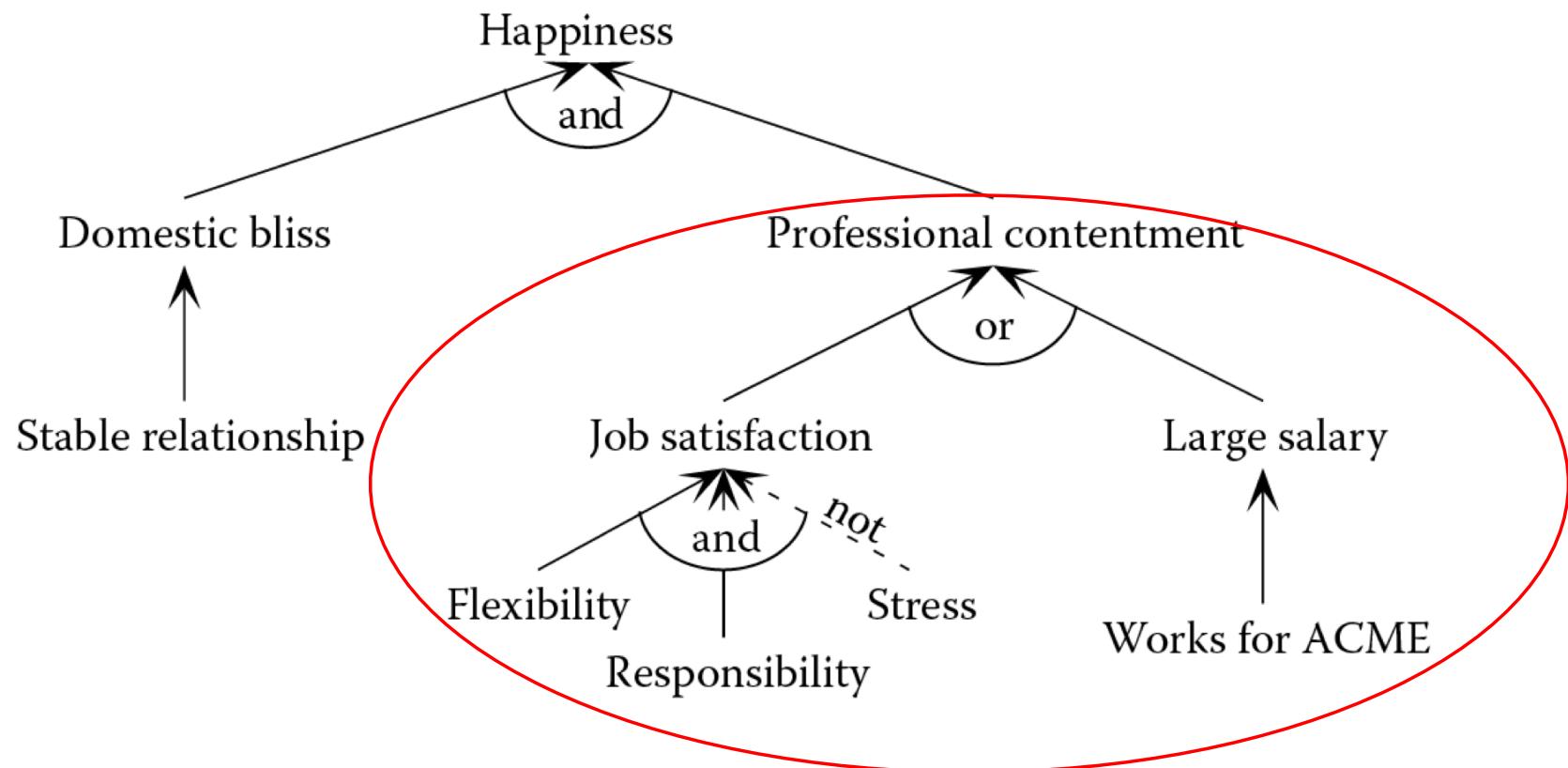
or the job_satisfaction of Person is true

then the professional_contentment of Person becomes true.
```

Inference Networks

- The interdependencies among rules, such as r1_1 and r1_2 define a network
- Inference network shows which facts can be logically combined to form new facts or conclusions
- The facts can be combined using “and”, “or” and “not”.
 - Professional contentment is true if either job satisfaction or large salary is true (or both are true).
 - Job satisfaction is achieved through flexibility, responsibility, and the absence of stress.

An Inference Network



Professional contentment is true if either job satisfaction or large salary is true (or both are true).

An Inference Network

- An inference network can be constructed by
 - taking *facts* and working out what conditions have to be met for those facts to be true.
 - After these conditions are found, they can be added to the diagram and linked by a *logical expression* (such as *and*, *or*, *not*).
 - This usually involves breaking down a complex logical expression into smaller parts.

Deduction, Abduction and Induction

Rules that make up inference network can be used to link cause and effect:

if <cause> then <effect>

E.g.:

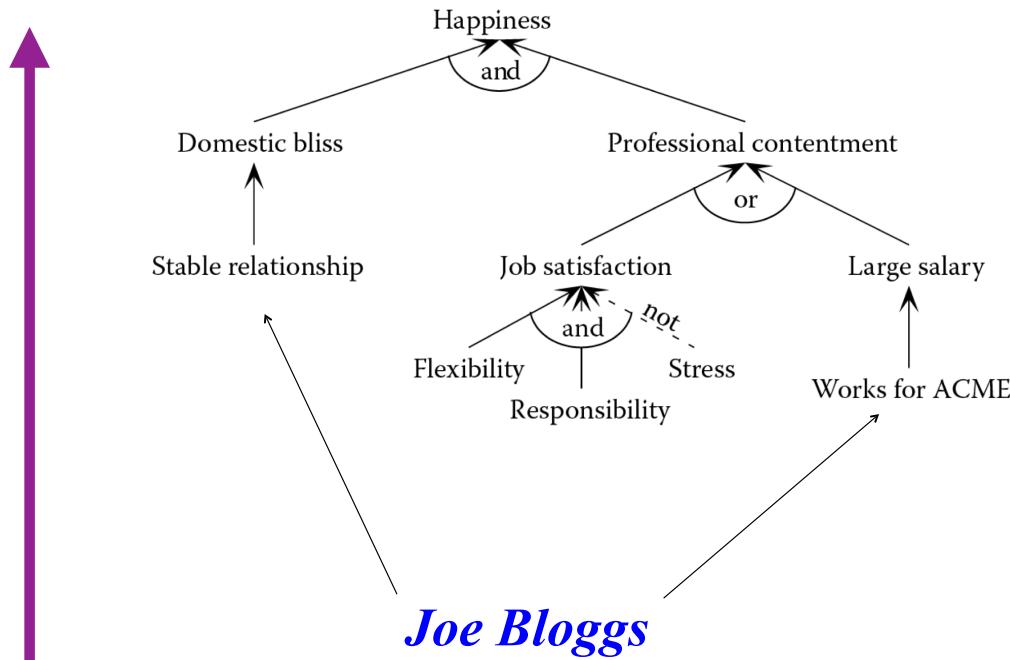
if

Joe Bloggs works for ACME and is in a stable relationship (the causes),
then

he is happy (the effect).

Deduction, Abduction and Induction

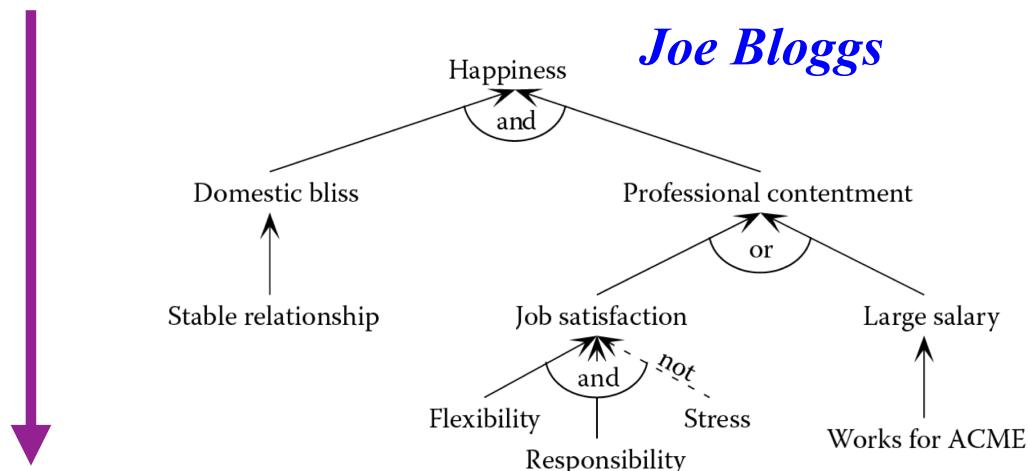
if <cause> then <effect>



if *Joe Bloggs* works for ACME and
is in a stable relationship (causes),
then he is happy (effect).

Deduction, **Abduction** and Induction

- Abduction - Many problems, such as diagnosis, involve reasoning in reverse, i.e, find a **cause**, given **an effect**.
- Given observation **Joe Bloggs is happy**, infer by abduction **Joe Bloggs enjoys domestic bliss and professional contentment**.



Deduction, Abduction and Induction

- If we have many examples of cause and effect, infer the **rule** that links them.
- E.g, if every employee of ACME earns a large salary, infer:

rule r1_1

if the employer of Person is acme
 then the salary of Person becomes large.

- Inferring a rule from a set of examples of cause and effect is **induction**.

Deduction, Abduction and Induction

- deduction: cause + rule \Rightarrow effect
- abduction: effect + rule \Rightarrow cause
- induction: cause + effect \Rightarrow rule

Closed-World Assumption

- Only facts that are in the knowledge base or that can be derived from rules are assumed to be true
- Everything is assumed to be false
- I.e. if we don't know it, it's assumed to be false
- That's why it's more accurate to say:
 - “a proof fails”, instead of “it's false”
 - “a proof succeeds” instead of, “it's true”

How many rabbits are there?



How many rabbits are there?

- Perception isn't all in the eye.
- Knowledge is usually needed to understand the world



References

- Poole & Mackworth, Artificial Intelligence: Foundations of Computational Agents, Chapter 5
- Russell & Norvig, *Artificial Intelligence: a Modern Approach*, Chapter 12.
- Adrian A. Hopgood. *Intelligent Systems for Engineers and Scientists (3rd Edition)*. CRC Press, 2011 – Chapter 1.

COMP3411/9814: Artificial Intelligence

Propositions and Inference

Lecture Outline

- Knowledge Representation and Logic
- Logical Arguments
- Propositional Logic
 - Syntax
 - Semantics
- Validity, Equivalence, Satisfiability, Entailment

Knowledge Bases

- A knowledge base is a set of sentences in a formal language.
- Declarative approach to building an agent:
 - Tell the system what it needs to know,
then it can ask itself what it needs to do
 - Answers should follow from the knowledge based.
- How do you formally specify how to answer questions?

Knowledge Based Agent

The agent must be able to:

- represent states, actions, etc.
- incorporate new percepts
- update internal representations of the world
- deduce hidden properties of the world
- determine appropriate actions

Why Formal Languages (not English, or other natural language)?

- Natural languages are **ambiguous**: “The fisherman went to the bank” (lexical)
- “The boy saw a girl with a telescope” (structural)
- “The table won’t fit through the doorway because it is too [wide/narrow]”
(co-reference)
- Ambiguity makes it difficult to interpret meaning of phrases/sentences
 - But also makes inference harder to define and compute
- Symbolic logic is a syntactically unambiguous language

Syntax vs Semantics

Syntax - legal sentences in knowledge representation language
(e.g. in the language of arithmetic expressions $x < 4$)

Semantics - meaning of sentences.

Refers to a sentence's relationship to the “real world” or to some model of the world.

- Semantic properties of sentences include truth and falsity
(e.g. $x < 4$ is true for $x = 3$ and false when $x = 5$).
- Semantic properties of names and descriptions include referents.
- The meaning of a sentence is not intrinsic to that sentence.
 - An interpretation is required to determine sentence meanings.
 - Interpretations are agreed amongst a linguistic community.

Propositions

- Propositions are entities (facts or non-facts) that can be true or false

Examples:

- “The sky is blue” - the sky is blue (here and now).
- “Socrates is bald” (assumes ‘Socrates’, ‘bald’ are well defined)
“The car is red” (requires ‘the car’ to be identified)
- “Socrates is bald and the car is red” (complex proposition)
- Use single letters to represent propositions, e.g. P : Socrates is bald
- Reasoning is independent of definitions of propositions

Logical Arguments

An **argument** relates a set of premises to a conclusion

- **valid** if the conclusion **necessarily follows** from the premises

All humans have 2 eyes

Jane is a human

Therefore Jane has 2 eyes

All humans have 4 eyes

Jane is a human

Therefore Jane has 4 eyes

- Both are (logically) correct valid arguments
- Which statements are true/false? Why?

Logical Arguments

An **argument** relates a set of premises to a conclusion

- **invalid** if the conclusion can be false when the premises are all true

All humans have 2 eyes

Jane has 2 eyes

Therefore Jane is human

No human has 4 eyes

Jane has 2 eyes

Therefore Jane is not human

- Both are (logically) **incorrect invalid** arguments
- Which statements are true/false? Why?

Propositional Logic

- Letters stand for “basic” propositions
- Combine into more complex sentences using operators **not**, **and**, **or**, **implies**, **iff**
- Propositional **connectives**:

\neg	negation	$\neg P$	“not P”
\wedge	conjunction	$P \wedge Q$	“P and Q”
\vee	disjunction	$P \vee Q$	“P or Q”
\rightarrow	implication	$P \rightarrow Q$	“If P then Q”
\leftrightarrow	bi-implication	$P \leftrightarrow Q$	“P if and only if Q”

From English to Propositional Logic

- “It is not the case that the sky is blue”: $\neg B$
(alternatively “the sky is not blue”)
- “The sky is blue and the grass is green”: $B \wedge G$
- “Either the sky is blue or the grass is green”: $B \vee G$
- “If the sky is blue, then the grass is not green”: $B \rightarrow \neg G$
- “The sky is blue if and only if the grass is green”: $B \leftrightarrow G$
- “If the sky is blue, then if the grass is not green, the plants will not grow”: $B \rightarrow (\neg G \rightarrow \neg P)$

Improving Readability

- $(P \rightarrow (Q \rightarrow (\neg(R))))$ vs $P \rightarrow (Q \rightarrow \neg R)$
- Rules for omitting parentheses
 - Omit parentheses where possible
 - Precedence from highest to lowest is: \neg , \wedge , \vee , \rightarrow , \leftrightarrow
 - All binary operators are left associative
 - so $P \rightarrow Q \rightarrow R$ abbreviates $(P \rightarrow Q) \rightarrow R$
 - Sometimes parentheses can't be removed:
 - Is $(P \vee Q) \vee R$ (always) the same as $P \vee (Q \vee R)$?
 - Is $(P \rightarrow Q) \rightarrow R$ (always) the same as $P \rightarrow (Q \rightarrow R)$? **NO!**
 - <https://web.stanford.edu/class/cs103/tools/truth-table-tool/>

P	Q	R	$((P \rightarrow Q) \rightarrow R)$	$(P \rightarrow (Q \rightarrow R))$
F	F	F	F	T
F	F	T	T	T
F	T	F	F	T
F	T	T	T	T
T	F	F	T	T
T	F	T	T	T
T	T	F	F	F
T	T	T	T	T

Truth Table Semantics

- The semantics of the connectives can be given by [truth tables](#)

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \rightarrow Q$	$P \leftrightarrow Q$
True	True	False	True	True	True	True
True	False	False	False	True	False	False
False	True	True	False	True	True	False
False	False	True	False	False	True	True

- One row for each possible assignment of True/False to variables
- Important: P and Q are any sentences, including complex sentences

Example – Complex Sentence

R	S	$\neg R$	$R \wedge S$	$\neg R \vee S$	$(R \wedge S) \rightarrow (\neg R \vee S)$
True	True	False	True	True	True
True	False	False	False	False	True
False	True	True	False	True	True
False	False	True	False	True	True

Thus $(R \wedge S) \rightarrow (\neg R \vee S)$ is a tautology

<https://web.stanford.edu/class/cs103/tools/truth-table-tool/>

Definitions

- A sentence is **valid** if it is True under all possible assignments of True/False to its variables (e.g. $P \vee \neg P$)
- A **tautology** is a valid sentence
- Two sentences are **equivalent** if they have the same truth table, e.g. $P \wedge Q$ and $Q \wedge P$
 - ▶ So P is equivalent to Q if and only if $P \leftrightarrow Q$ is valid
- A sentence is **satisfiable** if there is **some** assignment of True/False to its variables for which the sentence is True
- A sentence is **unsatisfiable** if it is not satisfiable (e.g. $P \wedge \neg P$)
 - ▶ Sentence is False for all assignments of True/False to its variables
 - ▶ So P is a tautology if and only if $\neg P$ is unsatisfiable

Material Implication

- $P \rightarrow Q$ evaluates to False only when P is True and Q is False
- $P \rightarrow Q$ is equivalent to $\neg P \vee Q$: material implication
- English usage often suggests a causal connection between antecedent (P) and consequent (Q) – this is not reflected in the truth table
- All these are tautologies
 - ▶ $(P \wedge Q) \rightarrow Q$
 - ▶ $P \rightarrow (P \vee Q)$
 - ▶ $(P \wedge \neg P) \rightarrow Q$

Material Implication

- $P \rightarrow Q$ evaluates to False only when P is True and Q is False
- $P \rightarrow Q$ is equivalent to $\neg P \vee Q$: material implication
- English usage often suggests a causal connection between antecedent (P) and consequent (Q) – this is not reflected in the truth table
- All these are tautologies
 - ▶ $(P \wedge Q) \rightarrow Q = \neg(P \wedge Q) \vee Q = \neg P \vee \neg Q \vee Q = T$
 - ▶ $P \rightarrow (P \vee Q) = \neg P \vee P \vee Q = T$
 - ▶ $(P \wedge \neg P) \rightarrow Q = \neg(P \wedge \neg P) \vee Q = \neg P \vee P \vee Q = T$

Logical Equivalences – All Valid

Commutativity:	$p \wedge q \leftrightarrow q \wedge p$	$p \vee q \leftrightarrow q \vee p$
Associativity:	$p \wedge (q \wedge r) \leftrightarrow (p \wedge q) \wedge r$	$p \vee (q \vee r) \leftrightarrow (p \vee q) \vee r$
Distributivity:	$p \wedge (q \vee r) \leftrightarrow (p \wedge q) \vee (p \wedge r)$	$p \vee (q \wedge r) \leftrightarrow (p \vee q) \wedge (p \vee r)$
Implication:	$(p \rightarrow q) \leftrightarrow (\neg p \vee q)$	
Idempotent:	$p \wedge p \leftrightarrow p$	$p \vee p \leftrightarrow p$
Double negation:	$\neg \neg p \leftrightarrow p$	
Contradiction:	$p \wedge \neg p \leftrightarrow \text{FALSE}$	
Excluded middle:		$p \vee \neg p \leftrightarrow \text{TRUE}$
De Morgan:	$\neg(p \wedge q) \leftrightarrow (\neg p \vee \neg q)$	$\neg(p \vee q) \leftrightarrow (\neg p \wedge \neg q)$

Proof of Equivalence

Let $P \Leftrightarrow Q$ mean “ P is equivalent to Q ” ($P \Leftrightarrow Q$ is not a formula)

Then $P \wedge (Q \rightarrow R) \Leftrightarrow \neg(P \rightarrow Q) \vee (P \wedge R)$

$$\begin{aligned} P \wedge (Q \rightarrow R) &\Leftrightarrow P \wedge (\neg Q \vee R) && [\text{Implication}] \\ &\Leftrightarrow (P \wedge \neg Q) \vee (P \wedge R) && [\text{Distributivity}] \\ &\Leftrightarrow (\neg \neg P \wedge \neg Q) \vee (P \wedge R) && [\text{Double negation}] \\ &\Leftrightarrow \neg(\neg P \vee Q) \vee (P \wedge R) && [\text{De Morgan}] \\ &\Leftrightarrow \neg(P \rightarrow Q) \vee (P \wedge R) && [\text{Implication}] \end{aligned}$$

Assumes substitution: if $A \Leftrightarrow B$, replace A by B in any subformula

Assumes equivalence is transitive: if $A \Leftrightarrow B$ and $B \Leftrightarrow C$ then $A \Leftrightarrow C$

Interpretations and Models

- An **interpretation** is an assignment of values to all variables.
- A **model** is an interpretation that satisfies the constraints.
 - A model is a **possible world** in which a sentence (or set of sentences) is true, e.g.
 - $x + y = 4$ in a world where $x = 2$ and $y = 2$
 - May be more than one possible world (e.g. $x = 3$ and $y = 1$)
- Often want to know what is true in all models.
- A proposition is statement that is true or false in each interpretation.

Entailment

- Entailment means that one sentence follows logically from another sentence, or set of sentences (i.e. a knowledge base):

$$KB \models \alpha$$

- Knowledge base KB entails sentence α if and only if α is true in all models (possible worlds) where KB is true.

e.g. the KB containing “the Moon is full” and “the tide is high” entails “Either the Moon is full or the tide is high”.

$$\text{e.g. } x + y = 4 \text{ entails } 4 = x + y$$

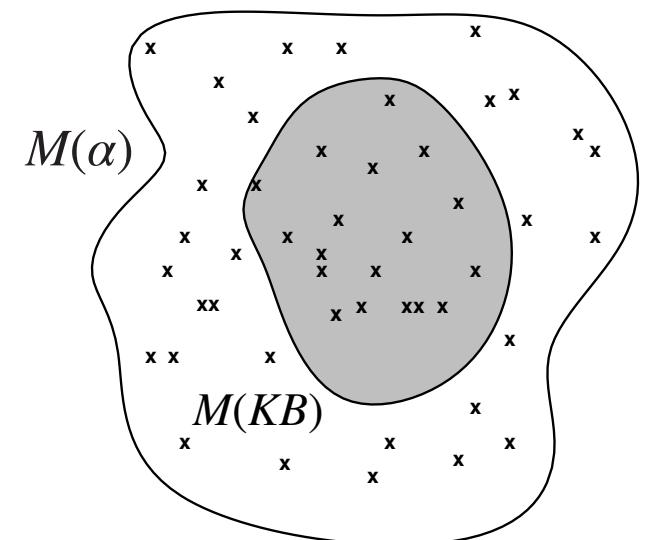
- Entailment is a relationship between sentences based on semantics.

Models

- For propositional logic, a model is **one** row of the truth table
- A model M **is a model of** a sentence α if α is True in M

Let $M(\alpha)$ be the set of all models of α

Then $KB \models \alpha$ if and only if $M(KB) \subseteq M(\alpha)$



Entailment

- S entails P ($S \models P$) if whenever all formulae in S are True, P is True
 - ▶ Semantic definition – concerns truth (not proof)
- Compute whether $S \models P$ by calculating a truth table for S and P
 - ▶ Syntactic notion – concerns computation/proof
 - ▶ Not always this easy to compute (how inefficient is this?)
- A tautology is a special case of entailment where S is the empty set
 - ▶ All rows of the truth table are True

Entailment Example

P	Q	$P \rightarrow Q$	Q
True	True	True	True
True	False	False	False
False	True	True	True
False	False	True	False

- $\{P, P \rightarrow Q\} \models Q$ since when both P and $P \rightarrow Q$ are True (row 1), Q is also True
- $P \rightarrow Q$ is calculated from P and Q using the truth table definition, and Q is used again to check the entailment

Example – $S \models P$

$$S = \{p \rightarrow q, q \rightarrow p, p \vee q\}$$

$$P = p \wedge q$$

Each row is an interpretation of S .
Only the first row is a model of S .

p	q	$p \rightarrow q$	$q \rightarrow p$	$p \vee q$	S	$p \wedge q$
T	T	T	T	T	T	T
T	F	F	T	T	F	F
F	T	T	F	T	F	F
F	F	T	T	F	F	F

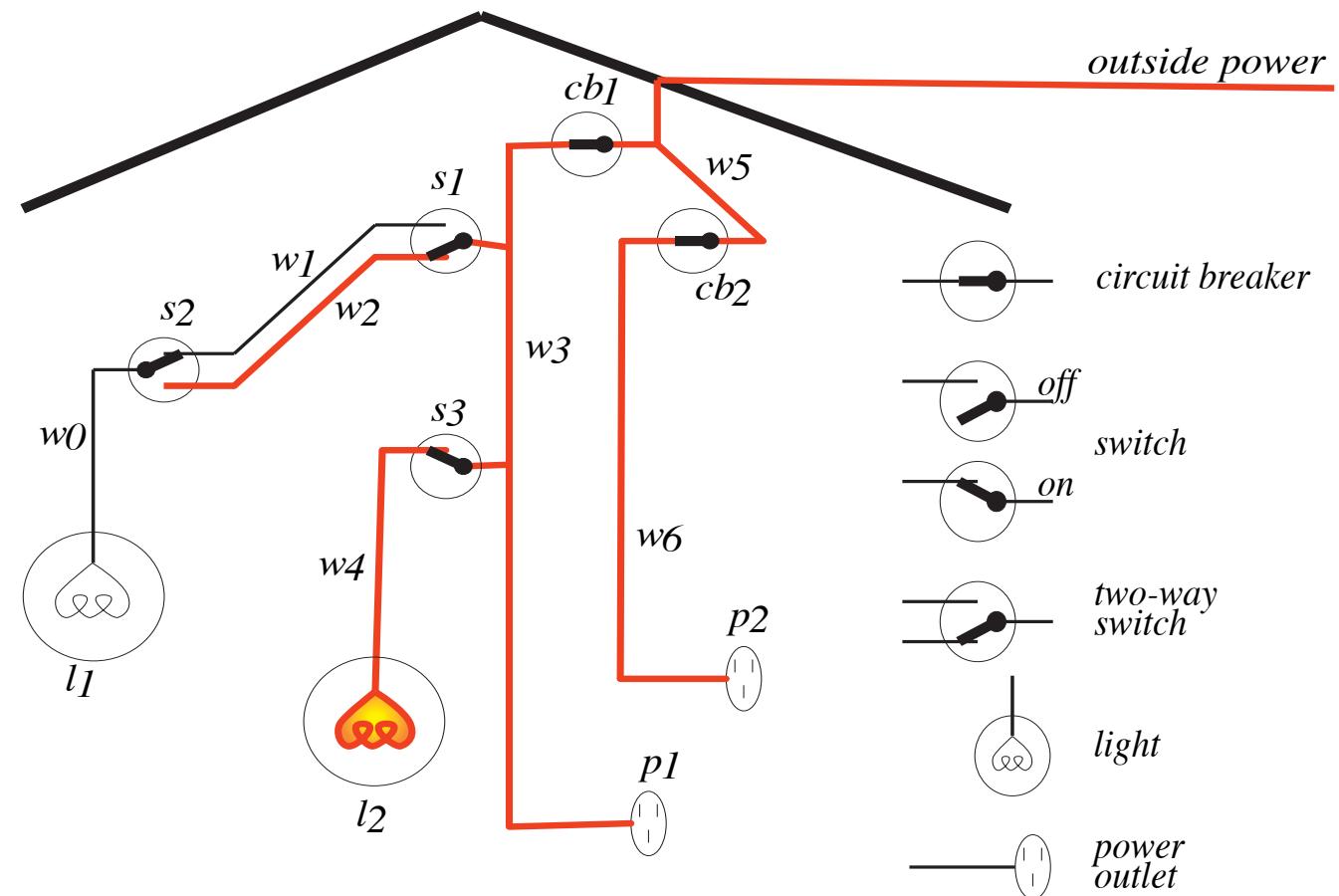
Example – $S \models P$

$$S = \{q \vee r, q \rightarrow \neg p, \neg(r \wedge p) \}$$

$$P = \neg p$$

p	q	r	$q \vee r$	$q \rightarrow \neg p$	$\neg(r \wedge p)$	S	$\neg p$
T	T	T	T	F		F	
T	T	F	T	F		F	
T	F	T	T	T	F	F	
T	F	F	F			F	
F	T	T	T	T	T	T	T
F	T	F	T	T	T	T	T
F	F	T	T	T	T	T	T
F	F	F	F			F	

Example - Modelling Electrical Circuits



Electrical Circuit in Proposition Logic

light_l1.

light_l2.

down_s1.

up_s2.

up_s3.

ok_l1.

ok_l2.

ok_cb1.

ok_cb2.

live_outside.

lit_l1 \leftarrow *live_w0* \wedge *ok_l1*

live_w0 \leftarrow *live_w1* \wedge *up_s2*.

live_w0 \leftarrow *live_w2* \wedge *down_s2*.

live_w1 \leftarrow *live_w3* \wedge *up_s1*.

live_w2 \leftarrow *live_w3* \wedge *down_s1*.

lit_l2 \leftarrow *live_w4* \wedge *ok_l2*.

live_w4 \leftarrow *live_w3* \wedge *up_s3*.

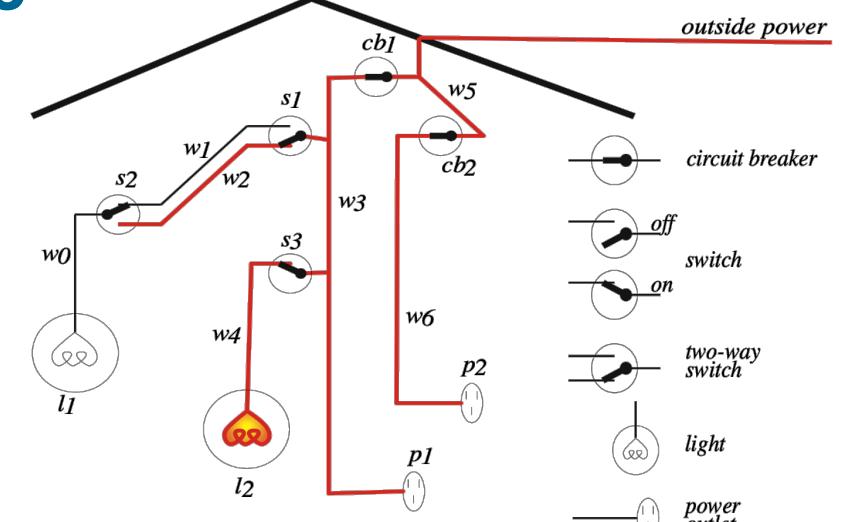
live_p1 \leftarrow *live_w3*.

live_w3 \leftarrow *live_w5* \wedge *ok_cb1*.

live_p2 \leftarrow *live_w6*.

live_w6 \leftarrow *live_w5* \wedge *ok_cb2*.

live_w5 \leftarrow *live_outside*.



Conclusion

- Ambiguity of natural languages avoided with formal languages
- Enables formalisation of (truth preserving) entailment
- Propositional Logic: Simplest logic of truth and falsity
- Knowledge Based Systems: First-Order Logic
- Automated Reasoning: How to compute entailment (inference)
- Many many logics not studied in this course

COMP3411: Artificial Intelligence

Automated Reasoning

C. Sammut & W. Wobcke

This Lecture

- Proof systems
 - ▶ Soundness, completeness, decidability
- Resolution and Refutation
- Horn clauses and SLD resolution
- Prolog

Summary So Far

- Propositional Logic
 - ▶ Syntax: Formal language built from \wedge , \vee , \neg , \rightarrow
 - ▶ Semantics: Definition of truth table for every formula
 - ▶ $S \models P$ if whenever all formulae in S are True, P is True
- Proof System
 - ▶ System of axioms and rules for **deduction**
 - ▶ Enables computation of proofs of P from S
- Basic Questions
 - ▶ Are the proofs that are computed always correct? (soundness)
 - ▶ If $S \models P$, is there always a proof of P from S (completeness)

Mechanising Proof

- A **proof** of a formula P from a set of **premises** S is a sequence of lines in which any line in the proof is
 1. An axiom of logic or premise from S , or
 2. A formula deduced from previous lines of the proof using a **rule of inference** and the last line of the proof is the formula P
- Formally captures the notion of mathematical proof
- S **proves** P ($S \vdash P$) if there is a proof of P from S ; alternatively, P **follows** from S
- Example: Resolution proof

Soundness and Completeness

- A proof system is **sound** if (intuitively) it preserves truth
 - ▶ Whenever $S \vdash P$, if every formula in S is True, P is also True
 - ▶ Whenever $S \vdash P$, $S \models P$
 - ▶ If you start with true assumptions, any conclusions **must** be true
- A proof system is **complete** if it is capable of proving all consequences of any set of premises (including infinite sets)
 - ▶ Whenever P is entailed by S , there is a proof of P from S
 - ▶ Whenever $S \models P$, $S \vdash P$
- A proof system is **decidable** if there is a mechanical procedure (computer program) which when asked whether $S \vdash P$, can **always** answer ‘true’ – or ‘false’ – correctly

Resolution

- A common type of proof system based on **refutation**
- Better suited to computer implementation than systems of axioms and rules
(**can** give correct ‘false’ answers)
- Decidable in the case of Propositional Logic
- Generalises to First-Order Logic (see next set of lectures)
- Needs all formulae to be converted to **clausal form**

Normal Forms

- A **literal** ℓ is a propositional variable or the negation of a propositional variable (P or $\neg P$)
- A **clause** is a disjunction of literals $\ell_1 \vee \ell_2 \vee \cdots \vee \ell_n$
- Conjunctive Normal Form (CNF) — a conjunction of clauses, e.g.

$$(P \vee Q \vee \neg R) \wedge (\neg S \vee \neg R)$$
 – or just one clause, e.g. $P \vee Q$

- Disjunctive Normal Form (DNF) — a disjunction of conjunctions of literals, e.g.

$$(P \wedge Q \wedge \neg R) \vee (\neg S \wedge \neg R)$$
 – or just one conjunction, e.g. $P \wedge Q$

- Every Propositional Logic formula can be converted to CNF and DNF
- Every Propositional Logic formula is equivalent to its CNF and DNF

Conversion to Conjunctive Normal Form

- Eliminate \leftrightarrow rewriting $P \leftrightarrow Q$ as $(P \rightarrow Q) \wedge (Q \rightarrow P)$
- Eliminate \rightarrow rewriting $P \rightarrow Q$ as $\neg P \vee Q$
- Use De Morgan's laws to push \neg inwards (repeatedly)
 - ▶ Rewrite $\neg(P \wedge Q)$ as $\neg P \vee \neg Q$
 - ▶ Rewrite $\neg(P \vee Q)$ as $\neg P \wedge \neg Q$
- Eliminate double negations: rewrite $\neg\neg P$ as P
- Use the distributive laws to get CNF [or DNF] – if necessary
 - ▶ Rewrite $(P \wedge Q) \vee R$ as $(P \vee R) \wedge (Q \vee R)$ [for CNF]
 - ▶ Rewrite $(P \vee Q) \wedge R$ as $(P \wedge R) \vee (Q \wedge R)$ [for DNF]

Example Clausal Form

Clausal Form = set of clauses in the CNF

- $\neg(P \rightarrow (Q \wedge R))$
- $\neg(\neg P \vee (Q \wedge R))$
- $\neg\neg P \wedge \neg(Q \wedge R)$
- $\neg\neg P \wedge (\neg Q \vee \neg R)$
- $P \wedge (\neg Q \vee \neg R)$
- Clausal Form: $\{P, \neg Q \vee \neg R\}$

Resolution Rule of Inference

$$\begin{array}{ccc} A_1 \vee \cdots \vee A_m \vee B & & \neg B \vee C_1 \vee \cdots \vee C_n \\ & \swarrow & \searrow \\ & A_1 \vee \cdots \vee A_m \vee C_1 \vee \cdots \vee C_n & \end{array}$$

where B is a propositional variable and A_i and C_j are literals

- B and $\neg B$ are **complementary literals**
- $A_1 \vee \cdots \vee A_m \vee C_1 \vee \cdots \vee C_n$ is the **resolvent** of the two clauses
- Special case: If no A_i and C_j , resolvent is empty clause, denoted \square or \perp

Resolution Rule

- Consider $A_1 \vee \cdots \vee A_m \vee B$ and $\neg B \vee C_1 \vee \cdots \vee C_n$
 - ▶ Suppose both are True
 - ▶ If B is True, $\neg B$ is False so $C_1 \vee \cdots \vee C_n$ must be True
 - ▶ If B is False, $A_1 \vee \cdots \vee A_m$ must be True
 - ▶ Hence $A_1 \vee \cdots \vee A_m \vee C_1 \vee \cdots \vee C_n$ is True

Hence the resolution rule is **sound**

- Starting with true premises, any conclusion made using resolution must be true

Applying Resolution: Naive Method

- Convert knowledge base into clausal form
- Repeatedly apply resolution rule to the resulting clauses
- P follows from the knowledge base if and only if each clause in the CNF of P can be derived using resolution from the clauses of the knowledge base (or subsumption)
- Example
 - ▶ $\{P \rightarrow Q, Q \rightarrow R\} \vdash P \rightarrow R$
 - ▶ Clauses $\neg P \vee Q$, $\neg Q \vee R$, show $\neg P \vee R$
 - ▶ Follows from one resolution step (Q and $\neg Q$ cancel, leaving $\neg P \vee R$)

Refutation Systems

- To show that P follows from S (i.e. $S \vdash P$) using refutation, start with S and $\neg P$ in clausal form and derive a contradiction using resolution
- A contradiction is the “empty clause” (a clause with no literals)
- The empty clause \square is unsatisfiable (always False)
- So if the empty clause \square is derived using resolution, the original set of clauses is unsatisfiable (never all True together)
- That is, if we can derive \square from the clausal forms of S and $\neg P$, these clauses can never be all True together
- Hence whenever the clauses of S are all True, at least one clause from $\neg P$ must be False, i.e. $\neg P$ must be False and P must be True
- By definition, $S \models P$ (so P can correctly be concluded from S)

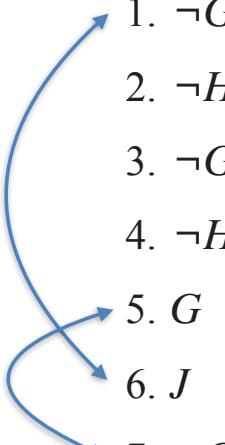
Applying Resolution Refutation

- Negate query to be proven (resolution is a refutation system)
- Convert knowledge base and negated query into CNF
- Repeatedly apply resolution until either the empty clause (contradiction) is derived or no more clauses can be derived
- If the empty clause is derived, answer ‘true’ (query follows from knowledge base), otherwise answer ‘false’ (query does not follow from knowledge base)

Resolution: Example 1

$$(G \vee H) \rightarrow (\neg J \wedge \neg K), G \vdash \neg J$$

Clausal form of is $\{ \neg G \vee \neg J, \neg H \vee \neg J, \neg G \vee \neg K, \neg H \vee \neg K \}$

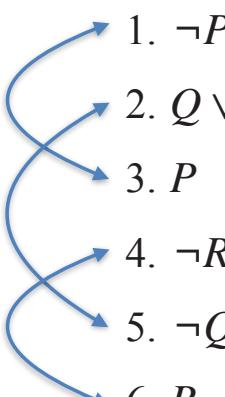
- 
1. $\neg G \vee \neg J$ [Premise]
 2. $\neg H \vee \neg J$ [Premise]
 3. $\neg G \vee \neg K$ [Premise]
 4. $\neg H \vee \neg K$ [Premise]
 5. G [Premise]
 6. J [\neg Query]
 7. $\neg G$ [1, 6 Resolution]
 8. \square [5, 7 Resolution]

Resolution: Example 2

$$P \rightarrow \neg Q, \neg Q \rightarrow R \vdash P \rightarrow R$$

Recall $P \rightarrow R \Leftrightarrow \neg P \vee R$

Clausal form of $\neg(\neg P \vee R)$ is $\{P, \neg R\}$

- 
1. $\neg P \vee \neg Q$ [Premise]
 2. $Q \vee R$ [Premise]
 3. P [\neg Query]
 4. $\neg R$ [\neg Query]
 5. $\neg Q$ [1, 3 Resolution]
 6. R [2, 5 Resolution]
 7. \square [4, 6 Resolution]

Resolution: Example 3

$$\vdash ((P \vee Q) \wedge \neg P) \rightarrow Q$$

Clausal form of $\vdash ((P \vee Q) \wedge \neg P) \rightarrow Q$ is $\{P \vee Q, \neg P, \neg Q\}$

- | | |
|---------------|-------------------|
| 1. $P \vee Q$ | [\neg Query] |
| 2. $\neg P$ | [\neg Query] |
| 3. $\neg Q$ | [\neg Query] |
| 4. Q | [1, 2 Resolution] |
| 5. \square | [3, 4 Resolution] |

Rewriting negated query in CNF:

$$\neg[((P \vee Q) \wedge \neg P) \rightarrow Q]$$

$$\neg[\neg((P \vee Q) \wedge \neg P) \vee Q]$$

$$\neg\neg((P \vee Q) \wedge \neg P) \wedge \neg Q$$

$$(P \vee Q) \wedge \neg P \wedge \neg Q$$

Now write in clausal form:

$$\{P \vee Q, \neg P, \neg Q\}$$

Soundness and Completeness Again

For Propositional Logic

- Resolution refutation is **sound**, i.e. it preserves truth (if a set of premises are all true, any conclusion drawn from those premises **must** also be true)
- Resolution refutation is **complete**, i.e. it is capable of proving all consequences of any knowledge base (not shown here!)
- Resolution refutation is **decidable**, i.e. there is an algorithm implementing resolution which when asked whether $S \vdash P$, can always answer ‘true’ or ‘false’ (correctly)

Heuristics in Applying Resolution

- Clause elimination — can disregard certain types of clauses
 - ▶ Pure clauses: contain literal L where $\neg L$ doesn't appear elsewhere
 - ▶ Tautologies: clauses containing both L and $\neg L$
 - ▶ Subsumed clauses: another clause is a subset of the literals
- Ordering strategies
 - ▶ Resolve unit clauses (only one literal) first
 - ▶ Start with query clauses
 - ▶ Aim to shorten clauses

Horn Clauses

Using a less expressive language makes proof procedure easier.

- Review
 - ▶ **literal** – proposition variable or negation of proposition variable
 - ▶ **clause** – disjunction of literals
- **Definite Clause** – exactly one positive literal
 - ▶ e.g. $B \vee \neg A_1 \vee \dots \vee \neg A_n$, i.e. $B \leftarrow A_1 \wedge \dots \wedge A_n$
- **Negative Clause** – no positive literals
 - ▶ e.g. $\neg Q_1 \vee \neg Q_2$ (negation of a query)
- **Horn Clause** – clause with at most one positive literal

Prolog

- Horn clauses in First-Order Logic
- SLD resolution
- Depth-first search strategy with backtracking
- User control
 - ▶ Ordering of clauses in Prolog database (facts and rules)
 - ▶ Ordering of subgoals in body of a rule
- Prolog is a programming language based on resolution refutation relying on the programmer to exploit search control rules

Prolog Clauses

$P :- Q, R, S.$

$P \leftarrow Q \wedge R \wedge S.$

$P \vee \neg(Q \wedge R \wedge S)$

$P \vee \neg Q \vee \neg R \vee \neg S$

Queries:

?- Q, R, S

$\perp \leftarrow Q \wedge R \wedge S$

$\neg(Q \wedge R \wedge S)$

$\neg Q \vee \neg R \vee \neg S$

Prolog DB = set of clauses

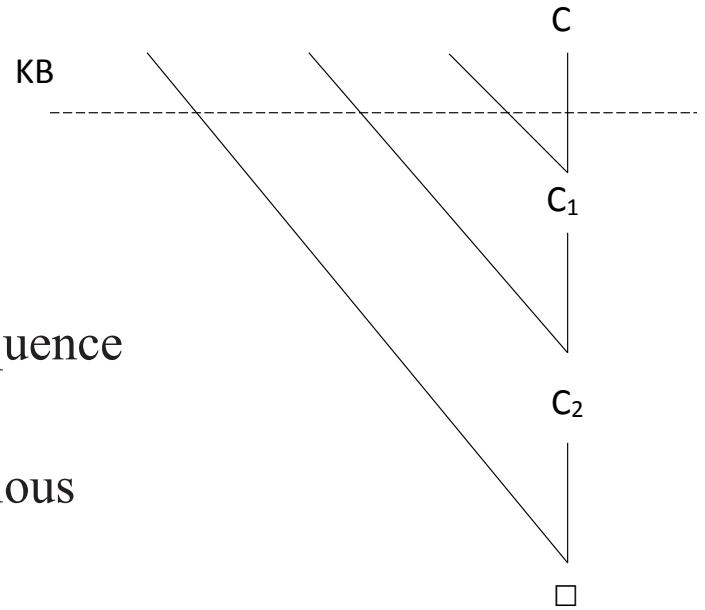
$$P \rightarrow Q \equiv \neg P \vee Q$$

$$P \leftarrow Q \equiv P \vee \neg Q$$

$\perp \equiv$ false (i.e. a contradiction)

SLD Resolution – \vdash_{SLD}

- Selected literals Linear form Definite clauses resolution
- SLD refutation of a clause C from a set of clauses KB is a sequence
 1. First clause of sequence is C
 2. Each intermediate clause C_i is derived by resolving the previous clause C_{i-1} and a clause from KB
 3. The last clause in the sequence is \square
- For a definite KB and negative clause query Q : $KB \cup Q \vdash \square$ if and only if $KB \cup Q \vdash_{SLD} \square$



Prolog Example

```
r.                      % facts
u.
v.

q :- r, u.              % rules
s :- v.
p :- q, r, s.

?- p.                   % query
true
```

Example Execution of Prolog interpreter

```
r.  
u.  
v.  
  
q :- r, u.  
s :- v.  
p :- q, r, s.  
  
?- p.
```

	Initial goal set = {p}	
1.	{q, r, s}	because p :- q, r, s.
2.	{r, u, r, s}	because q :- r, u.
3.	{u, r, s}	because r.
4.	{r, s}	because u.
5.	{s}	because r.
6.	{v}	because s :- v
7.	{}	because v.
8.	=> true	because empty clause

- In each step, we remove the first element in the goal set and replace it with the body of the clause whose head matches that element. E.g. remove *p* and replace by *q, r, s*.
- **Note:** The simple Prolog interpreter isn't smart enough to remove the duplication of *r* in step 2.

Prolog Interpreter

Input: A query Q and a logic program KB

Output: ‘true’ if Q follows from KB , ‘false’ otherwise

Initialise current goal set to $\{Q\}$

while the current goal set is not empty do

 Choose G from the current goal set; (first in goal set)

 Make a copy $G' :- B_1, \dots, B_n$ of a clause from KB

 (**try all in KB**) (**if no such rule, try alternative rules**)

 Replace G by B_1, \dots, B_n in current goal set

if current goal set is empty:

 output ‘true’

else output ‘false’

Inefficient and not how a
real Prolog interpreter works

- Depth-first, left-right **with backtracking**

Conclusion: Propositional Logic

- Propositions built from \wedge , \vee , \neg , \rightarrow
- Sound, complete and decidable proof systems (inference procedures)
 - ▶ Natural deduction
 - ▶ Resolution refutation
 - ▶ Prolog for special case of definite clauses
 - ▶ Tableau method
- Limited expressive power
 - ▶ Cannot express ontologies (no relations)
- First-Order Logic can express knowledge about objects, properties and relationships between objects

COMP3411: Artificial Intelligence

First-Order Logic

C. Sammut & W. Wobcke

Propositional Logic

- Propositions built from \wedge , \vee , \neg , \rightarrow
- Sound, complete and decidable proof systems (inference procedures)
 - ▶ Resolution refutation
 - ▶ Prolog for special case of definite clauses
- Limited expressive power
 - ▶ Cannot express relations and ontologies
- First-Order Logic can express knowledge about objects, properties and relationships between objects

Applications of Logic and Theorem Proving

- Knowledge Representation of AI Systems
 - ▶ Declarative and Working Memory for an intelligent agent
 - ▶ Answering questions about state of the world
- Semantic Web
 - ▶ Extension to World Wide Web
 - ▶ Makes information on the web machine interpretable
- Formal Verification
 - ▶ Used by chip designers to verify VLSI designs
 - ▶ Proof assistants used in software verification

This Lecture

- First-Order Logic
 - ▶ Syntax
 - ▶ Semantics
- Automated Reasoning
 - ▶ Conjunctive Normal Form
 - ▶ First-order resolution and unification
 - ▶ Soundness, completeness, decidability

Syntax of First-Order Logic

- Constant symbols: $a, b, \dots, Mary$ (objects)
- Variables: x, y, \dots
- Function symbols: $f, mother_of, sine, \dots$
- Predicate symbols: $Mother, likes, \dots$
- Quantifiers: \forall (universal); \exists (existential)

Language of First-Order Logic

- Terms: constants, variables, functions applied to terms (refer to objects)
 - ▶ e.g. a , $f(a)$, $\text{mother_of}(\text{Mary})$, ...
- Atomic formulae: predicates applied to tuples of terms
 - ▶ e.g. $\text{likes}(\text{Mary}, \text{mother_of}(\text{Mary}))$, $\text{likes}(x, a)$
- Quantified formulae:
 - ▶ e.g. $\forall x \text{ likes}(x, a)$, $\exists x \text{ likes}(x, \text{mother_of}(y))$
 - ▶ Second occurrences of x are **bound** by quantifier (\forall in first case, \exists in second) and y in the second formula is **free**

Converting English into First-Order Logic

- Everyone likes lying on the beach — $\forall x \text{ likes_lying_on_beach}(x)$
- Someone likes Fido — $\exists x \text{ likes}(x, \text{Fido})$
- No one likes Fido — $\neg(\exists x \text{ likes}(x, \text{Fido}))$ (or $\forall x \neg \text{likes}(x, \text{Fido})$)
- Fido doesn't like everyone — $\neg \forall x \text{ likes}(\text{Fido}, x)$
- All cats are mammals — $\forall x (\text{cat}(x) \rightarrow \text{mammal}(x))$
- Some mammals are carnivorous — $\exists x (\text{mammal}(x) \wedge \text{carnivorous}(x))$
- Note: $\forall x A(x) \Leftrightarrow \neg \exists x \neg A(x)$, $\exists x A(x) \Leftrightarrow \neg \forall x \neg A(x)$

Universal Quantifiers

All men are mortal: $\forall x (\text{man}(x) \rightarrow \text{mortal}(x))$

- $\forall x P$ is (almost) equivalent to the **conjunction** of **instantiations** of P
- $\forall x(\text{man}(x) \rightarrow \text{mortal}(x)) \Leftrightarrow$

$$(\text{man}(\text{Alan}) \rightarrow \text{mortal}(\text{Alan}))$$

$$\wedge \quad (\text{man}(\text{Bill}) \rightarrow \text{mortal}(\text{Bill}))$$

$$\wedge \quad (\text{man}(\text{Colin}) \rightarrow \text{mortal}(\text{Colin}))$$

$$\wedge \quad \dots$$

... only if every **object** (not only man) in the domain has a name

Existential Quantifiers

Some cats are immortal: $\exists x (cat(x) \wedge \neg mortal(x))$

- $\exists x P$ is (almost) equivalent to the **disjunction** of instantiations of P
- $\exists x (cat(x) \wedge \neg mortal(x)) \Leftrightarrow$

$$(cat(Alan) \wedge \neg mortal(Alan))$$

$$\vee (cat(Bill) \wedge \neg mortal(Bill))$$

$$\vee (cat(Colin) \wedge \neg mortal(Colin))$$

$$\vee \dots$$

... only if every **object** (not only cat) in the domain has a name

Nested Quantifiers

The order of quantification is very important

- Everything likes everything — $\forall x \forall y \text{ } \textit{likes}(x, y)$ (or $\forall y \forall x \text{ } \textit{likes}(x, y)$)
- Something likes something — $\exists x \exists y \text{ } \textit{likes}(x, y)$ (or $\exists y \exists x \text{ } \textit{likes}(x, y)$)
- Everything likes something — $\forall x \exists y \text{ } \textit{likes}(x, y)$
- There is something liked by everything — $\exists y \forall x \text{ } \textit{likes}(x, y)$

Defining Semantic Properties

Brothers are siblings

$$\forall x \forall y (\text{brother}(x, y) \rightarrow \text{sibling}(x, y))$$

“Sibling” is symmetric

$$\forall x \forall y (\text{sibling}(x, y) \leftrightarrow \text{sibling}(y, x))$$

One’s mother is one’s female parent

$$\forall x \forall y (\text{mother}(x, y) \leftrightarrow (\text{female}(x) \wedge \text{parent}(x, y)))$$

A first cousin is a child of a parent’s sibling

$$\forall x \forall y (\text{FirstCousin}(x, y) \leftrightarrow \exists p \exists s (\text{parent}(p, x) \wedge \text{sibling}(p, s) \wedge \text{parent}(s, y)))$$

Scope Ambiguity

- Typical pattern for \forall and \exists
 - ▶ $\forall x(type_of(x) \rightarrow predicate(x))$
 - ▶ $\exists x(type_of(x) \wedge predicate(x))$
- Every student took an exam
 - ▶ $\forall x(student(x) \rightarrow \exists y(exam(y) \wedge took(x, y)))$
 - ▶ $\exists y(exam(y) \wedge \forall x(student(x) \rightarrow took(x, y)))$

Scope of Quantifiers

- The **scope** of a quantifier in a formula A is that subformula B of A of which that quantifier is the main logical operator
- Variables belong to the **innermost** quantifier that mentions them
- Examples
 - ▶ $Q(x) \rightarrow \forall y P(x, y)$ — scope of $\forall y$ is $\forall y P(x, y)$
 - ▶ $\forall z P(z) \rightarrow \neg Q(z)$ — scope of $\forall z$ is $\forall z P(z)$ but not $Q(z)$
 - ▶ $\exists x(P(x) \rightarrow \forall x P(x))$
 - ▶ $\forall x(P(x) \rightarrow Q(x)) \rightarrow (\forall x P(x) \rightarrow \forall x Q(x))$

Semantics of First-Order Logic

- An interpretation is required to give semantics to first-order logic.
 - An **interpretation** is a non-empty “domain of discourse” (set of objects).
The truth of any formula depends on the interpretation.
- The interpretation defines, for each
 - constant symbol** an object in the domain
 - function symbol** a function from domain tuples to the domain
 - predicate symbol** a relation over the domain (a set of tuples)
- Then by definition
 - universal quantifier** $\forall x P(x)$ is True iff $P(a)$ is True for all assignments of domain elements a to x
 - existential quantifier** $\exists x P(x)$ is True iff $P(a)$ is True for at least one assignment of domain element a to x

Resolution for First-Order Logic

(Alan Robinson, 1965)

- Based on resolution for Propositional Logic
- Extended syntax: allow variables and quantifiers
- Define “clausal form” for first-order logic formulae
- Eliminate quantifiers from clausal forms
- Adapt resolution procedure to cope with variables (unification)

Conversion to Conjunctive Normal Form

1. Eliminate implications and bi-implications as in propositional case
2. Move negations inward using De Morgan's laws
 - ▶ plus rewriting $\neg\forall x P$ as $\exists x \neg P$ and $\neg\exists x P$ as $\forall x \neg P$
3. Eliminate double negations
4. Rename bound variables if necessary so each only occurs once
 - ▶ e.g. $\forall x P(x) \vee \exists x Q(x)$ becomes $\forall x P(x) \vee \exists y Q(y)$
5. Use equivalences to move quantifiers to the left
 - ▶ e.g. $\forall x P(x) \wedge Q$ becomes $\forall x (P(x) \wedge Q)$ where x is not in Q
 - ▶ e.g. $\forall x P(x) \wedge \exists y Q(y)$ becomes $\forall x \exists y (P(x) \wedge Q(y))$

Conversion to CNF – Continued

6. Skolemise (replace each existentially quantified variable by a new term)
 - ▶ $\exists x P(x)$ becomes $P(a_0)$ using a Skolem constant a_0 since $\exists x$ occurs at the outermost level
 - ▶ Any constant can represent the x , since it must be unique
 - ▶ $\forall x \exists y P(x, y)$ becomes $P(x, f_0(x))$ using a Skolem function f_0 since $\exists y$ occurs within $\forall x$
 - ▶ y may depend on the choice of x , so y is a function of x
7. The formula now has only universal quantifiers and all are at the left of the formula: drop them
8. Use distribution laws to get CNF and then clausal form

PCNF: Example 1

Remember: $\forall x P(x) \Leftrightarrow \neg \exists x \neg P(x)$, $\exists x P(x) \Leftrightarrow \neg \forall x \neg P(x)$

$$\forall x [\forall y P(x, y) \rightarrow \neg \forall y (Q(x, y) \rightarrow R(x, y))]$$

$$1. \forall x [\neg \forall y P(x, y) \vee \neg \forall y (\neg Q(x, y) \vee R(x, y))]$$

Eliminate Implication

$$2, 3. \forall x [\exists y \neg P(x, y) \vee \exists y (Q(x, y) \wedge \neg R(x, y))]$$

Move negation inward

$$4. \forall x [\exists y \neg P(x, y) \vee \exists z (Q(x, z) \wedge \neg R(x, z))]$$

Rename variables

$$5. \forall x \exists y \exists z [\neg P(x, y) \vee (Q(x, z) \wedge \neg R(x, z))]$$

Move quantifiers left

$$6. \forall x [\neg P(x, f(x)) \vee (Q(x, g(x)) \wedge \neg R(x, g(x)))]$$

Replace \exists vars by Skolem fns

$$7. \neg P(x, f(x)) \vee (Q(x, g(x)) \wedge \neg R(x, g(x)))$$

Drop \forall

$$8. (\neg P(x, f(x)) \vee Q(x, g(x))) \wedge (\neg P(x, f(x)) \vee \neg R(x, g(x)))$$

Distribution laws to get CNF

$$8. \{\neg P(x, f(x)) \vee Q(x, g(x)), \neg P(x, f(x)) \vee \neg R(x, g(x))\}$$

Clausal form

CNF: Example 2

$$\neg \exists x \forall y \forall z ((P(y) \vee Q(z)) \rightarrow (P(x) \vee Q(x)))$$

1. $\neg \exists x \forall y \forall z (\neg(P(y) \vee Q(z)) \vee P(x) \vee Q(x))$

Eliminate Implication

2. $\forall x \neg \forall y \forall z (\neg(P(y) \vee Q(z)) \vee P(x) \vee Q(x))$

Move negation inward

2. $\forall x \exists y \neg \forall z (\neg(P(y) \vee Q(z)) \vee P(x) \vee Q(x))$

Move negation inward

2. $\forall x \exists y \exists z \neg(\neg(P(y) \vee Q(z)) \vee P(x) \vee Q(x))$

Move negation inward

2. $\forall x \exists y \exists z ((P(y) \vee Q(z)) \wedge \neg(P(x) \vee Q(x)))$

Move negation inward

6. $\forall x ((P(f(x)) \vee Q(g(x))) \wedge \neg P(x) \wedge \neg Q(x))$

Replace \exists vars by Skolem fns

7. $(P(f(x)) \vee Q(g(x))) \wedge \neg P(x) \wedge \neg Q(x)$

Drop \forall

8. $\{P(f(x)) \vee Q(g(x)), \neg P(x), \neg Q(x)\}$

Clausal form

Unification

- A **unifier** of two atomic formulae is a **substitution** of terms **for variables** that makes them identical
 - ▶ Each variable has at most one associated term
 - ▶ Substitutions are applied simultaneously
- Unifier of $P(x, f(a), z)$ and $P(z, z, u)$: $\{x / f(a), z / f(a), u / f(a)\}$
- Substitution σ_1 is a **more general unifier** than a substitution σ_2 if for some substitution τ ,
$$\sigma_2 = \sigma_1 \tau \text{ (i.e. } \sigma_1 \text{ followed by } \tau\text{)}$$
- **Theorem.** If two atomic formulae are unifiable, they have a *most general unifier* (mgu).

Examples

- $\{P(x, a), P(b, c)\}$ is not unifiable (where a, b, c are constants)
- $\{P(f(x), y), P(a, w)\}$ is not unifiable
- $\{P(x, c), P(b, c)\}$ is unifiable by $\{x/b\}$
- $\{P(f(x), y), P(f(a), w)\}$ is unifiable by
 $\sigma = \{x/a, y/w\}$, $\tau = \{x/a, y/a, w/a\}$, $\nu = \{x/a, y/b, w/b\}$
Note that σ is an *mgu* and $\tau = \sigma\theta$ where $\theta = \dots?$
- $\{P(x), P(f(x))\}$ is not unifiable (circular reference requires occurs check)

Unification Algorithm

```
Unify( $\Psi_1, \Psi_2$ ): // returns substitution or Failure
  if  $\Psi_1$  or  $\Psi_2$  is a variable or constant, then:
    if  $\Psi_1$  or  $\Psi_2$  are identical, then return NIL // unification succeeds but no substitution needed
    else if  $\Psi_1$  is a variable
      then if  $\Psi_1$  occurs in  $\Psi_2$ , then return Failure
      else return { $\Psi_1 / \Psi_2$ }
    else if  $\Psi_2$  is a variable
      if  $\Psi_2$  occurs in  $\Psi_1$  then return Failure
      else return { $\Psi_2 / \Psi_1$ }
    else return Failure
  if the predicate symbols of  $\Psi_1$  and  $\Psi_2$  are not same then return Failure
  if  $\Psi_1$  and  $\Psi_2$  have a different number of arguments then return Failure
  Set Substitution, SUBST to NIL
  for i=1 to the number of elements in  $\Psi_1$ :
    S = Unify( $\Psi_1[i], \Psi_2[i]$ )
    if S = Failure then return Failure
    if S ≠ NIL then
      Apply S to the remainder of both  $\Psi_1$  and  $\Psi_2$ 
      SUBST = APPEND(S, SUBST).
  return SUBST
```

Unification Algorithm (alternative)

- The disagreement set of S : Find the leftmost position at which not all members E of S have the same symbol.
 - The set of subexpressions of each E in S that begin at this position is the disagreement set of S .
- Algorithm
 1. $S_0 = S, \sigma_0 = \{\}, i = 0$
 2. If S_i is not a singleton find its disagreement set D_i , otherwise terminate with σ_i as the most general unifier
 3. If D_i contains a variable v_i and term t_i such that v_i does not occur in t_i then
$$\sigma_{i+1} = \sigma_i\{v_i/t_i\}, \quad S_{i+1} = S_i\{v_i/t_i\}$$
otherwise terminate as S is not unifiable
 4. $i = i + 1$; resume from step 2

Examples

- $S = \{ f(\underline{x}, g(x)), f(\underline{h(y)}, g(h(z))) \}$

$$D_0 = \{x, h(y)\} \text{ so } \sigma_1 = \{x/h(y)\}$$

$$S_1 = \{ f(h(y), g(\underline{h(y)})), f(h(y), g(\underline{h(z)})) \}$$

$$D_1 = \{y, z\} \text{ so } \sigma_2 = \{x/h(z), y/z\}$$

$$S_2 = \{ f(h(z), g(h(z))), f(h(z), g(\underline{h(z)})) \}$$

i.e. σ_2 is an mgu

- $S = \{ f(h(x), g(x)), f(g(x), h(x)) \}$ (does an mgu exist for these?)

First-Order Resolution

$$\begin{array}{ccc} A_1 \vee \cdots \vee A_m \vee B & & \neg B' \vee C_1 \vee \cdots \vee C_n \\ & \swarrow & \searrow \\ & (A_1 \vee \cdots \vee A_m \vee C_1 \vee \cdots \vee C_n) \theta & \end{array}$$

where B, B' are positive literals, A_i, C_j are literals, θ is an mgu of B and B'

- B and $\neg B'$ are **complementary literals**
- $(A_1 \vee \cdots \vee A_m \vee C_1 \vee \cdots \vee C_n) \theta$ is the **resolvent** of the two clauses
- Special case: If no A_i and C_j , resolvent is empty clause, denoted \square , or \perp

Applying Resolution Refutation

- Negate query to be proven (resolution is a refutation system)
- Convert knowledge base and negated query into CNF
- Repeatedly apply resolution to clauses **or copies of clauses** until either the empty clause (contradiction) is derived or no more clauses can be derived (a copy of a clause is the clause with all variables renamed)
- If the empty clause is derived, answer ‘true’ (query follows from knowledge base), otherwise answer ‘false’ (query does not follow from knowledge base) ... and if there are an infinite number of clauses that can be derived, don’t answer at all

Resolution: Example 1

$$\vdash \exists x(P(x) \rightarrow \forall x P(x))$$

$$\text{CNF}(\neg \exists x(P(x) \rightarrow \forall x P(x)))$$

$$\forall x \neg(\neg P(x) \vee \forall x P(x)) \quad [\text{eliminate implication, move negation}]$$

$$\forall x(\neg\neg P(x) \wedge \neg\forall x P(x)) \quad [\text{move negation}]$$

$$\forall x(P(x) \wedge \exists x \neg P(x)) \quad [\text{move negation, eliminate double negation}]$$

$$\forall x(P(x) \wedge \exists y \neg P(y)) \quad [\text{rename variables}]$$

$$\forall x \exists y(P(x) \wedge \neg P(y)) \quad [\text{move quantifiers left}]$$

$$\forall x(P(x) \wedge \neg P(f(x))) \quad [\text{Skolemse}]$$

$$P(x), \neg P(f(x)) \quad [\text{Distribution}]$$

$$1. P(x) \quad [\neg \text{Query}]$$

$$2. \neg P(f(y)) \quad [\text{Copy of } \neg \text{Query, renaming variables}]$$

$$3. \square \quad [1, 2 \text{ Resolution } \{x/f(y)\}]$$

Resolution: Example 2

$$\vdash \exists x \forall y \forall z ((P(y) \vee Q(z)) \rightarrow (P(x) \vee Q(x)))$$

$$\text{CNF} \equiv \{P(f(x)) \vee Q(g(x)), \neg P(x), \neg Q(x)\}$$

1. $P(f(x)) \vee Q(g(x))$ [¬ Query]
2. $\neg P(x)$ [¬ Query]
3. $\neg Q(x)$ [¬ Query]
4. $\neg P(y)$ [Copy of 2]
5. $Q(g(x))$ [1, 4 Resolution $\{y/f(x)\}$]
6. $\neg Q(z)$ [Copy of 3]
7. \square [5, 6 Resolution $\{z/g(x)\}$]

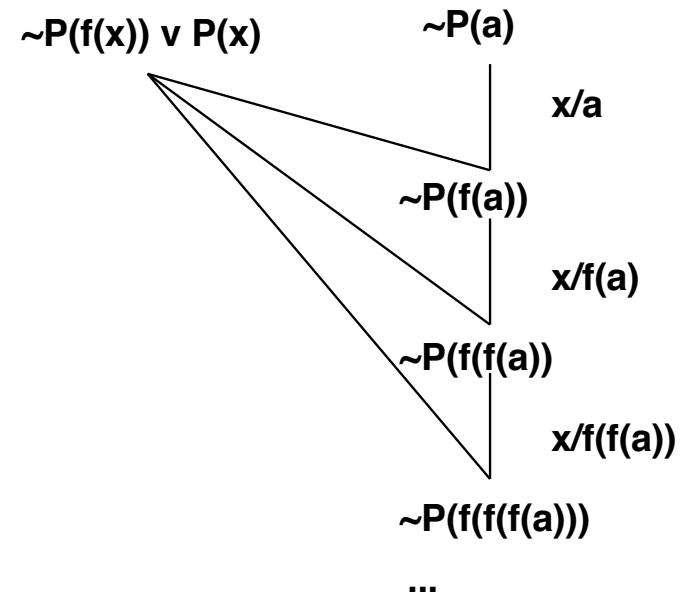
Soundness and Completeness

For First-Order Logic

- Resolution refutation is **sound**, i.e. it preserves truth (if a set of premises are all true, any conclusion drawn from those premises must also be true)
- Resolution refutation is **complete**, i.e. it is capable of proving all consequences of any knowledge base (not shown here!)
- Resolution refutation is not **decidable**, i.e. there is no algorithm implementing resolution which when asked whether $S \vdash P$, can always answer ‘true’ or ‘false’ (correctly)

Undecidability of First-Order Logic

- $KB = \{P(f(x)) \rightarrow P(x)\}$
- $Q = P(a) ?$
- Obviously $KB \models Q$
- However, now try to show this using resolution



Undecidability of First-Order Logic

- Can we determine in general when this problem will arise? **No!**
- There is no general procedure
 - if** (KB unsatisfiable)
 return true
 - else return** false
- Resolution refutation is complete so if KB is unsatisfiable, the search tree will contain the empty clause somewhere
- . . . but if the search tree does not contain the empty clause the search may go on forever
- Even in the propositional case (which is decidable), complexity of resolution is $O(2^n)$ for problems of size n

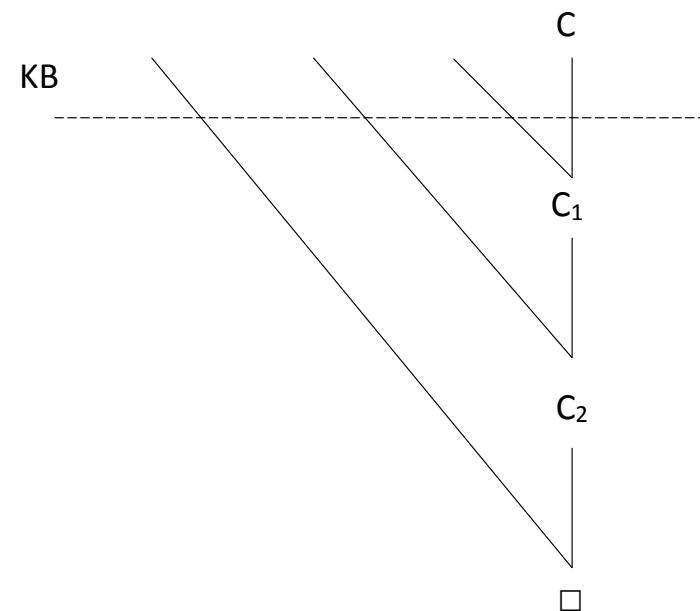
Horn Clauses

Use less expressive language

- Review
 - ▶ **literal** – atomic formula or negation of atomic formula
 - ▶ **clause** – disjunction of literals
- **Definite Clause** – exactly one positive literal
 - ▶ e.g. $B \vee \neg A_1 \vee \dots \vee \neg A_n$, i.e. $B \leftarrow A_1 \wedge \dots \wedge A_n$
- **Negative Clause** – no positive literals
 - ▶ e.g. $\neg Q_1 \vee \neg Q_2$ (negation of a query)
- **Horn Clause** – clause with at most one positive literal

SLD Resolution — \vdash_{SLD}

- Selected literals Linear form Definite clauses resolution
- SLD refutation of a clause C from a set of clauses KB is a sequence
 1. First clause of sequence is C
 2. Each intermediate clause C_i is derived by resolving the previous clause C_{i-1} and a clause from KB
 3. The last clause in the sequence is \square
- Theorem. For a definite KB and negative clause query Q : $KB \cup Q \vdash \square$ if and only if $KB \cup Q \vdash_{SLD} \square$



Prolog

- Horn clauses in First-Order Logic
- SLD resolution
- Depth-first search strategy with backtracking
- User control
 - ▶ Ordering of clauses in Prolog database (facts and rules)
 - ▶ Ordering of subgoals in body of a rule
- Prolog is a programming language based on resolution refutation relying on the programmer to exploit search control rules

Prolog Example

```
p(X) :- q(X), r(X, Z), s(f(Z)).  
q(X) :- r(X, Y), u(X).  
s(f(X)) :- v(X).
```

```
r(a, b).  
u(a).  
v(b).
```

```
?- p(X)  
X = a
```

Prolog Example

```
p(X1) :- q(X1), r(X1, Z), s(f(Z)).  
q(X2) :- r(X2, Y), u(X2).  
s(f(X3)) :- v(X3).
```

```
r(a, b).  
u(a).  
v(b).  
?- p(X0)  
X0 = a
```

Goal stack:	
1. [p(X0)]	
2. [q(X0), r(X0, Z), s(f(Z))]	{X0/X1}
3. [r(a, b), u(a), r(a, b), s(f(b))]	{X0/X1, X1/X2, X2/a, Y/b}
4. [u(a), r(a, b), s(f(b))]	{X0/X1, X1/X2, X2/a, Y/b}
5. [r(a, b), s(f(b))].	{X0/X1, X1/X2, X2/a, Y/b}
6. [s(f(b))]	{X0/X1, X1/X2, X2/a, Y/b, Z/X3, X3/b}
7. [v(b)]	{X0/X1, X1/X2, X2/a, Y/b, Z/X3, X3/b}
8. []	

Prolog Interpreter

Input: A query Q and a logic program KB

Output: ‘true’ if Q follows from KB , ‘false’ otherwise

Initialise current goal set to $\{Q\}$

while the current goal set is not empty do

 Choose G from the current goal set (**first in goal set**)

 Choose a **copy** $G' :- B_1, \dots, B_n$ of a rule from KB for which most general unifier of
 G, G' is θ (**try all in KB**)

 (if no such rule, undo unifications and try alternative rules) Apply θ to the current
 goal set

 Replace $G\theta$ by $B_1\theta, \dots, B_n\theta$ in current goal set

if current goal set is empty

 output true

else output false

Inefficient and not how a
real Prolog interpreter works

- Depth-first, left-right **with backtracking**

Negation as Failure

- Prolog does not implement classical negation
- Prolog `\+ G` is known as **negation as failure**
- $KB \vdash \text{\+ } G$ — KB cannot prove G
- $KB \vdash \neg G$ — can prove $\neg G$
- Negation as failure is **finite** failure

Soundness and Completeness Again

- Prolog including negation as failure is not **sound**, i.e. it does not preserve truth
- Pure Prolog (without negation as failure) is not **complete**, i.e. it is incapable of proving all consequences of any knowledge base (this is because of the search order)
- Even pure Prolog is not **decidable**, i.e. the Prolog implementation of resolution when asked whether $KB \vdash Q$, can not always answer ‘true’ or ‘false’ (correctly)

Conclusion

- First-order logic can express objects, properties of objects and relationships between objects, and allows quantification over variables
- First-order logic is highly expressive
- Resolution refutation is sound and complete, but not decidable
- Prolog is more programming language than theorem prover
- Gödel's **incompleteness theorem**
 - ▶ Any first-order logic system with Peano's axioms for arithmetic cannot be both consistent and prove all true statements (where statements are encoded using numbers)

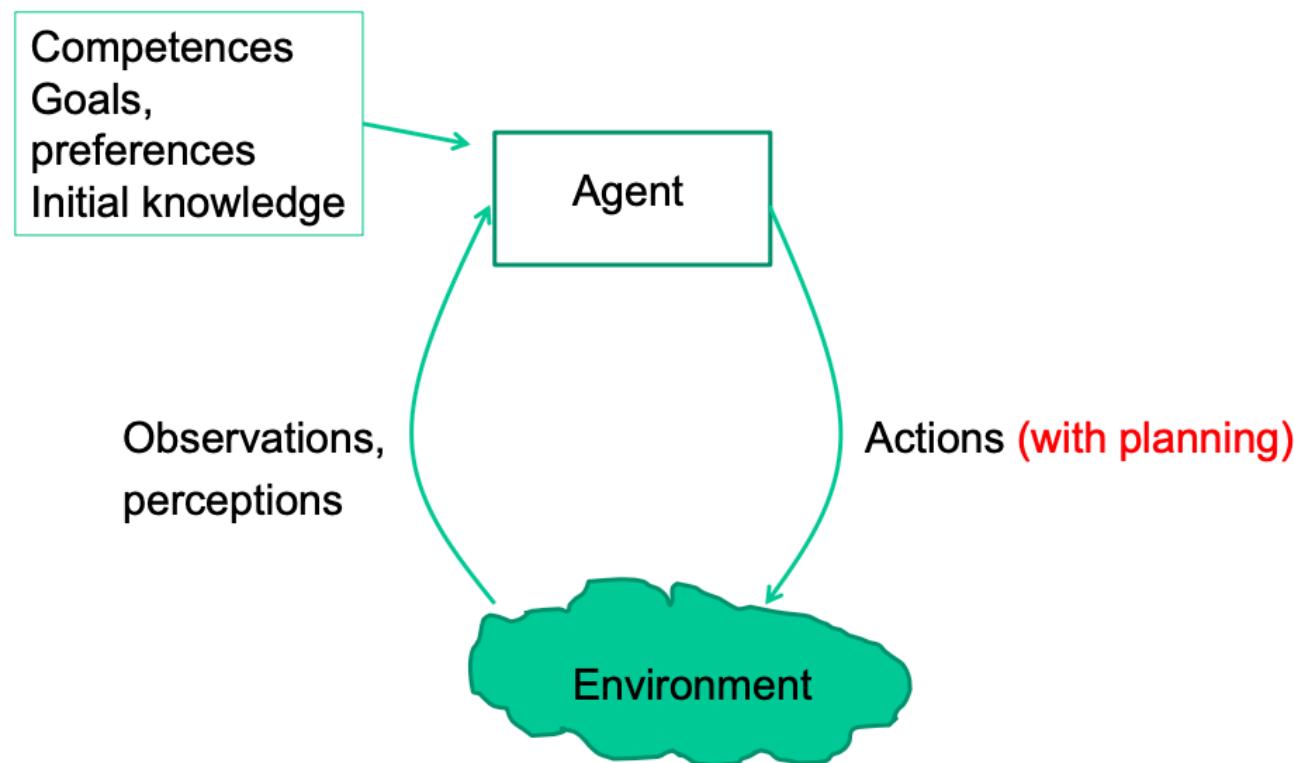
Planning

COMP3411/9814: Artificial Intelligence

Lecture Overview

- Reasoning About Action
- STRIPS Planner
- Forward planning
- Regression Planning
- Partial Order Planning
- GraphPlan
- Planning as Constraint Satisfaction

Agent acting in its environment



Planning

- Planning is deciding what to do based on an agent's ability, its goals. and the state of the world.
- Planning is finding a sequence of actions to solve a goal.
- Assumptions:
 - World is deterministic.
 - No exogenous events outside of control of robot change state of world.
 - The agent knows what state it is in.
 - Time progresses discretely from one state to the next.
 - Goals are predicates of states that need to be achieved or maintained.

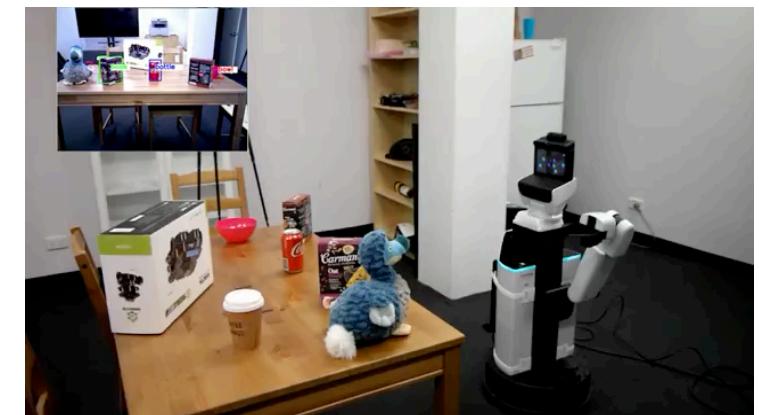
Planning Agent

- The planning agent or goal-based agent is more flexible than a reactive agent because the knowledge that supports its decisions is represented explicitly and **can be modified**.
- The agent's behaviour can easily be changed.
- Doesn't work when assumptions are violated



Planning Agent

- Environment changes due to the performance of actions
- Planning scenario
 - Agent can control its environment
 - Only atomic actions, not processes with duration
 - Only single agent in the environment (no interference)
 - Only changes due to agent executing actions (no evolution)
- More complex examples
 - RoboCup soccer
 - Delivery robot
 - Self-driving car



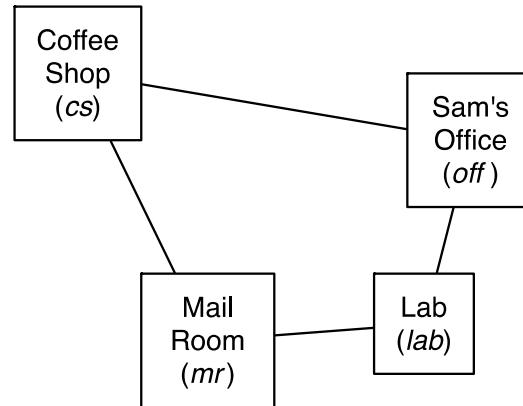
Representation

- How to represent a classical planning problem?
 - States, Actions, and Goals
 - Can represent relation between states and actions
 - explicit state space representation
 - action-centric
 - feature-based

Actions

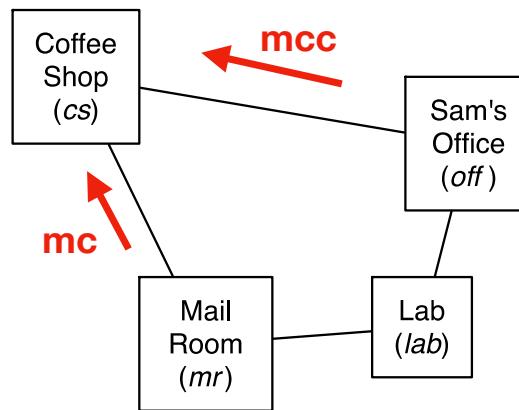
- A deterministic **action** is a partial function from states to states.
- The **preconditions** of an action specify when the action can be carried out.
- The **effect** of an action specifies the resulting state.

Delivery Robot Example



The robot, called Rob, can buy coffee at the coffee shop, pick up mail in the mail room, move, and deliver coffee and/or mail.

Delivery Robot Example



Features:

RLoc – Rob's location
RHC – Rob has coffee
SWC – Sam wants coffee
MW – Mail is waiting
RHM – Rob has mail

Features to describe states

Actions:

mc – move clockwise
mcc – move counterclockwise
puc – pickup coffee
dc – deliver coffee
pum – pickup mail
dm – deliver mail

Robot actions

State Description

The state is described in terms of the following features:

- RLoc - the robot's location,
 - coffee shop (*cs*), Sam's office (*off*), the mail room (*mr*) or laboratory (*lab*)
- SWC - Sam wants coffee.
 - The atom *swc* means Sam wants coffee and $\neg swc$ means Sam does not want coffee.

Robot Actions

- Rob has six actions
 - Rob can move clockwise (*mc*)
 - Rob can move counterclockwise (*mcc*) or (*mac*), for now we use (*mcc*).
 - Rob can pick up coffee (*puc*) if Rob is at the coffee shop.
 - Rob can deliver coffee (*dc*) if Rob has coffee and is in the same location as Sam.
 - Rob can pick up mail (*pum*) if Rob is in the mail room.
 - Rob can deliver mail (*dm*) if Rob has mail and is in the same location as Sam.
- Assume that it is only possible for Rob to do one action at a time.

Explicit State-Space Representation

- The states are specifying the following:
 - the robot's location,
 - whether the robot has coffee,
 - whether Sam wants coffee,
 - whether mail is waiting,
 - whether the robot is carrying the mail.

$$\langle lab, \neg rhc, swc, \neg mw, rhm \rangle$$

Explicit State-Space Representation

<i>State</i>	<i>Action</i>	<i>Resulting State</i>
$\langle lab, \neg rhc, swc, \neg mw, rhm \rangle$	mc	$\langle mr, \neg rhc, swc, \neg mw, rhm \rangle$
$\langle lab, \neg rhc, swc, \neg mw, rhm \rangle$	mcc	$\langle off, \neg rhc, swc, \neg mw, rhm \rangle$
$\langle off, \neg rhc, swc, \neg mw, rhm \rangle$	dm	$\langle off, \neg rhc, swc, \neg mw, \neg rhm \rangle$
$\langle off, \neg rhc, swc, \neg mw, rhm \rangle$	mcc	$\langle cs, \neg rhc, swc, \neg mw, rhm \rangle$
$\langle off, \neg rhc, swc, \neg mw, rhm \rangle$	mc	$\langle lab, \neg rhc, swc, \neg mw, rhm \rangle$
...

The complete representation includes the transitions for the other 62 states.

Explicit State-Space Representation

This is not a good representation:

- There are usually too many states to represent, to acquire, and to reason with.
- Small changes to the model mean a large change to the representation.
 - Adding another feature means changing the whole representation.
- It does not represent the structure of states;
 - there is much structure and regularity in the effects of actions that is not reflected in the state transitions.

STRIPS Language for Problem Definition

- STRIPS = Stanford Research Institute Problem Solver
- Most planners use a “STRIPS-like representation”
 - i.e. STRIPS with some extensions
- STRIPS makes some simplifications:
 - no variables in goals
 - positive relations given only
 - unmentioned relations are assumed false (c.w.a. – closed world assumption)
 - effects are conjunctions of relations

STRIPS Representation

- Each action has a:
 - **precondition** that specifies when the action can be carried out.
 - **effect** a set of assignments of values to primitive features that are made true by this action.
 - Often split into an ADD list (things that become true after action)
 - and DELETE list (things that become false after action)

Assumption: every primitive feature not mentioned in the effects is unaffected by the action.

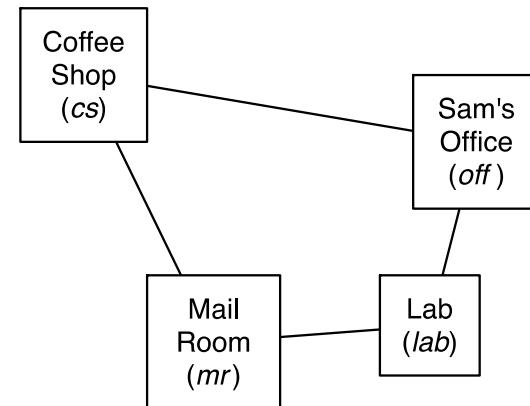
Example STRIPS Representation

Pick-up coffee (puc):

- **precondition:** [cs, \neg rhc]
- **effect:** [rhc]

Deliver coffee (dc):

- **precondition:** [off, rhc]
- **effect:** [\neg rhc, \neg swc]



Feature-Based Representation of Actions

- For each action:
 - **precondition** is a proposition that specifies when the action can be carried out.
- For each feature:
 - **causal rules** that specify when the feature gets a new value and
 - **frame rules** that specify when the feature keeps its value.

Example Feature-Based Representation

- Precondition of pick-up coffee (puc):

$$RLoc=cs \wedge \neg rhc$$

- Rules for “location is cs”:

$$RLoc'=cs \leftarrow Rloc = off \wedge Act=mcc$$

$$RLoc'=cs \leftarrow Rloc = mr \wedge Act=mc$$

$$RLoc'=cs \leftarrow Rloc = cs \wedge Act \neq mcc \wedge Act \neq mc$$

- Rules for “robot has coffee”

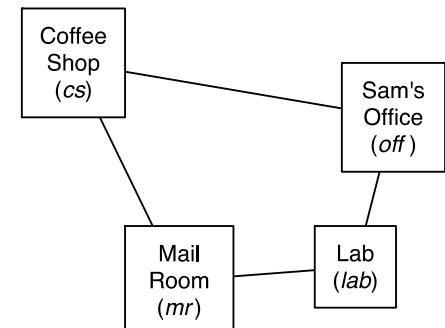
$$rhc' \leftarrow Act=puc \wedge \neg rhc$$

$$rhc' \leftarrow rhc \wedge Act \neq dc$$

- Causal rules say how features change
- Frame rules say how features stay the same

causal rules

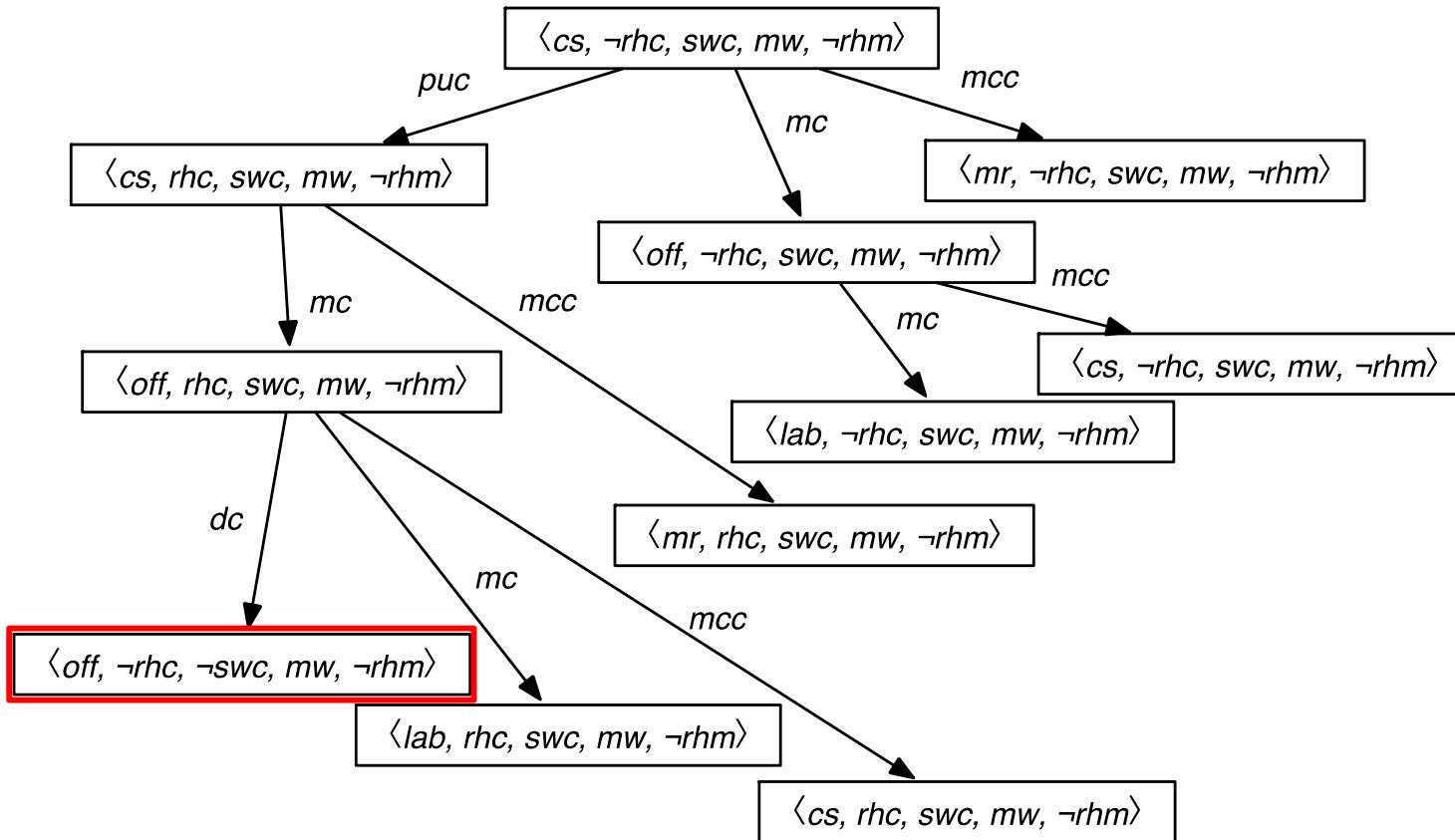
frame rule



Forward Planning

- Nodes are states in the world
- Arcs correspond to actions that transform one state into another
- Start node is the initial state
- If *goal condition* is satisfied, search terminates successfully
- A path corresponds to a plan to achieve goal

Forward Search



Actions:

mc – move clockwise
 mcc – move counterclockwise
 puc – pickup coffee
 dc – deliver coffee
 pum – pickup mail
 dm – deliver mail

Locations:

cs – coffee shop
 off – office
 lab – laboratory
 mr – mail room

Features:

$RLoc$ – Rob's location
 RHC – Rob has coffee
 SWC – Sam wants coffee
 MW – Mail is waiting
 RHM – Rob has mail

Initial:

SWC – Sam wants coffee

Goal:

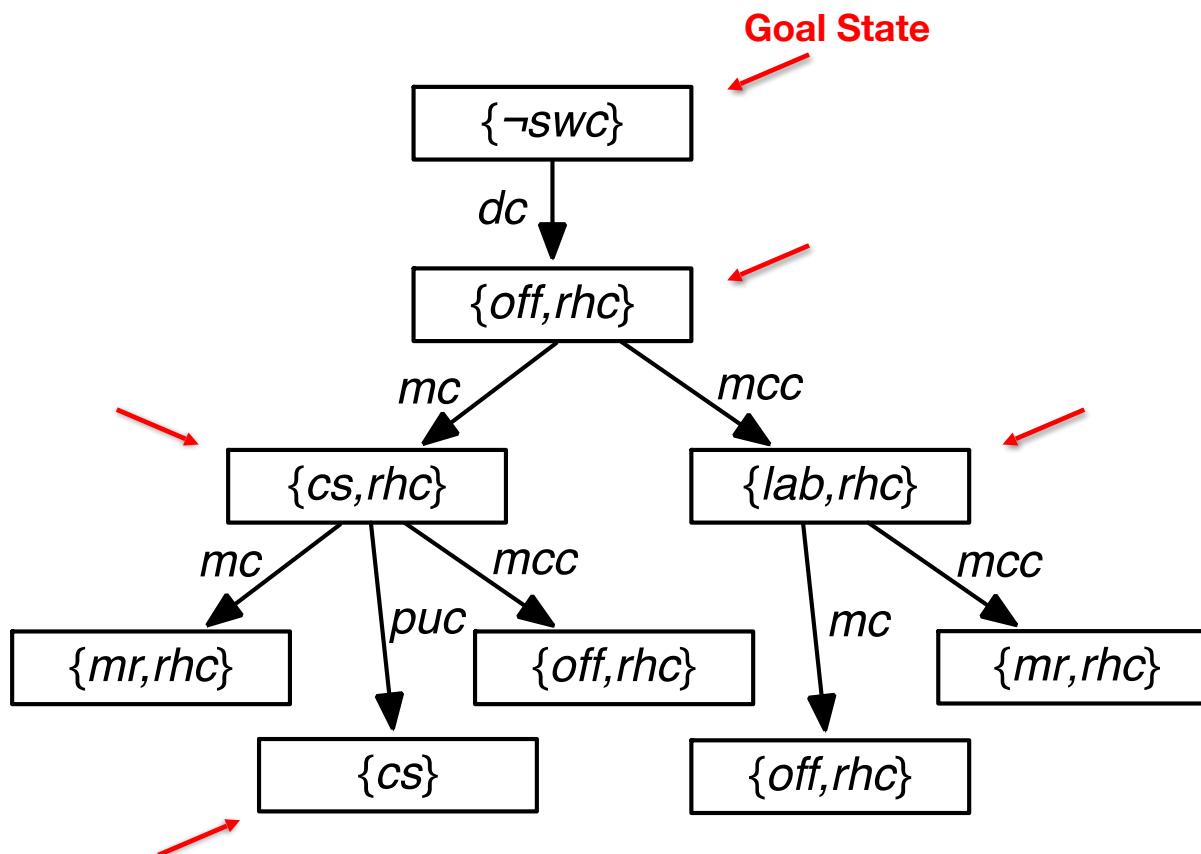
$\neg SWC$ – Sam wants coffee

Recall our Prolog Planner

Regression Planning (Backward Search)

- Nodes are subgoals.
- Arcs correspond to actions. An arc from node g to g' , labelled with action act , means
 - act is the last action that is carried out before subgoal g is achieved, and
 - node g' is a subgoal that must be true immediately before act so that g is true immediately after act .
- The start node is the planning goal to be achieved.
- The goal condition for the search, $\text{goal}(g)$, is true if g is true of the initial state.

Regression Planning



Actions:

mc – move clockwise
mcc – move counterclockwise
puc – pickup coffee
dc – deliver coffee
pum – pickup mail
dm – deliver mail

Locations:

cs – coffee shop
off – office
lab – laboratory
mr – mail room

Features:

RLoc – Rob's location
RHC – Rob has coffee
SWC – Sam wants coffee
MW – Mail is waiting
RHM – Rob has mail

Backward Regression

$$g' = (g - Add(a)) \cup Precond(a)$$

- g' is the regression from goal g over action a
- I.e. going backwards from g , we look for an action, a , that has preconditions and effects that satisfy g'

Backward Chaining

```
% State of the robot's world = state(RobotLocation, BasketLocation, RubbishLocation)
% action(Action, State, NewState): Action in State produces NewState
% We assume robot never drops rubbish on floor and never pushes rubbish around

plan_backwards(State, State, []).
                                         % To achieve State from State itself, do nothing
plan_backwards(State1, GoalState, [Action | RestofPlan]) :-
    action(Action, PreviousState, GoalState),
                                         % Actions will be in reverse order
    plan_backwards(State1, PreviousState, RestofPlan).
                                         % Find an action that achieves GoalState
                                         % Make PreviousState the new goal

id_plan_backwards(Start, Goal, Plan) :-
    append(RevPlan, _, _),
    plan_backwards(Start, Goal, RevPlan),
    reverse(RevPlan, Plan).

?- id_plan_backwards(state(door, corner, floor(middle)), state(_, _, in_basket), Plan).
X = [go(door, corner), push(corner, middle), pickup, drop]
```

Relational State Representation

First-order representations are more flexible

- e.g. states in blocks world can be represented by set of relations
- 1, 2, 3, 4 represent positions on a table:

`on(c, a).`

`on(a, 1).`

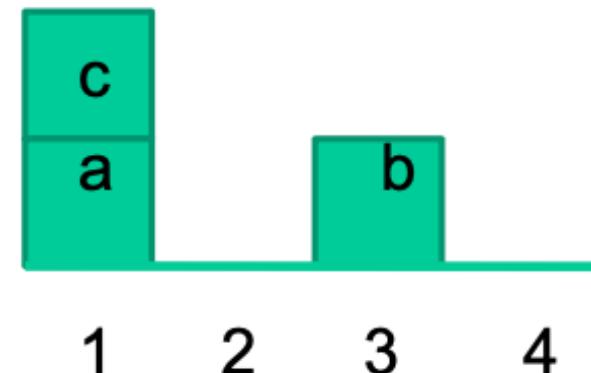
`on(b, 3).`

`clear(2).`

`clear(4).`

`clear(b).`

`clear(c).`



Defining Goals and Possible Actions

- Example of goals:

$\text{on}(a, b), \text{on}(b, c)$

- Example of action:

$\text{move}(a, 1, b)$

(Move block a from 1 to b)

- Action preconditions:

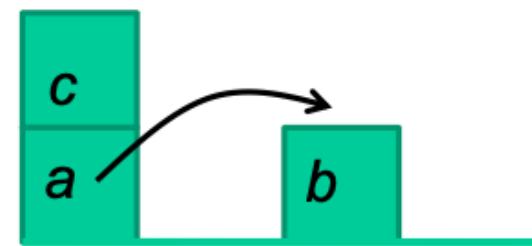
$\text{clear}(a), \text{on}(a, 1), \text{clear}(b)$

“add” (true after action)

- Action effects:

$\text{on}(a, b), \text{clear}(1), \neg \text{on}(a, 1), \neg \text{clear}(b)$

“delete” (no longer true after action)



STRIPS Action Schema

Action schema represents a set of actions using variables
(variable names start with capital letter, like Prolog)

Action:

`move(Block, From, To)`

Precondition:

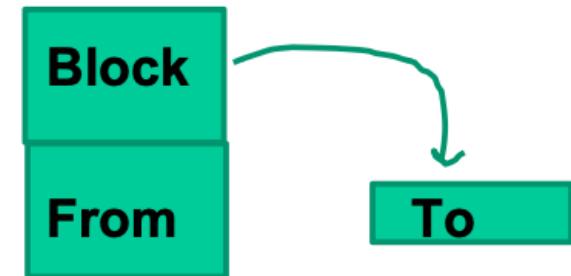
`clear(Block), clear(To), on(Block, From)`

Adds:

`on(Block, To), clear(From)`

Deletes:

`on(Block, From), clear(To)`



Better with Additional Constraints

Action:

`move(Block, From, To)`

Precondition for Action:

`clear(Block), clear(To), on(Block, From)`

Additional constraints:

`block(Block),` % Object Block to be moved must be a block

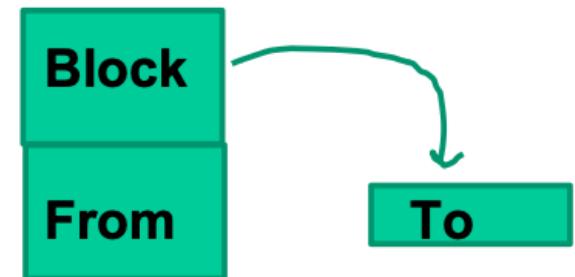
`object(To),` % "To" is an object, i.e. a block or a place

`To ≠ Block,` % Block cannot be moved to itself

`object(From),` % "From" is a block or a place

`From ≠ To,` % Move to new position

`Block ≠ From`



PDDL

Planning Domain Description Language

- Extension of STRIPS representation
- Invented for planning competitions to provide an implementation independent language for describing action schema and domain knowledge
- There are several variants to cover different planning domains
 - e.g. continuous domains, continuous actions, probabilities, etc.

Example

Init: $\text{airport(mel)} \wedge \text{airport(syd)} \wedge \text{plane(p1)} \wedge \text{plane(p2)} \wedge \text{cargo(c1)} \wedge \text{cargo(c2)} \wedge \text{at(c1, syd)} \wedge \text{at(c2, mel)} \wedge \text{at(p1, syd)} \wedge \text{at(p2, mel)}$

Goal: $\text{at(c1, mel)} \wedge \text{at(c2, syd)}$

Action load(C, P, A)

PRECOND: $\text{cargo(C)} \wedge \text{plane(P)} \wedge \text{airport(A)} \wedge \text{at(C, A)} \wedge \text{at(P, A)}$

EFFECT: $\neg \text{at(C, A)} \wedge \text{in(C, P)}$

Action unload(C, P, A)

PRECOND: $\text{cargo(C)} \wedge \text{plane(P)} \wedge \text{airport(a)} \wedge \text{In(C, P)} \wedge \text{at(P, A)}$

EFFECT: $\text{at(C, A)} \wedge \neg \text{in(C, P)}$

Action fly(P, From, To)

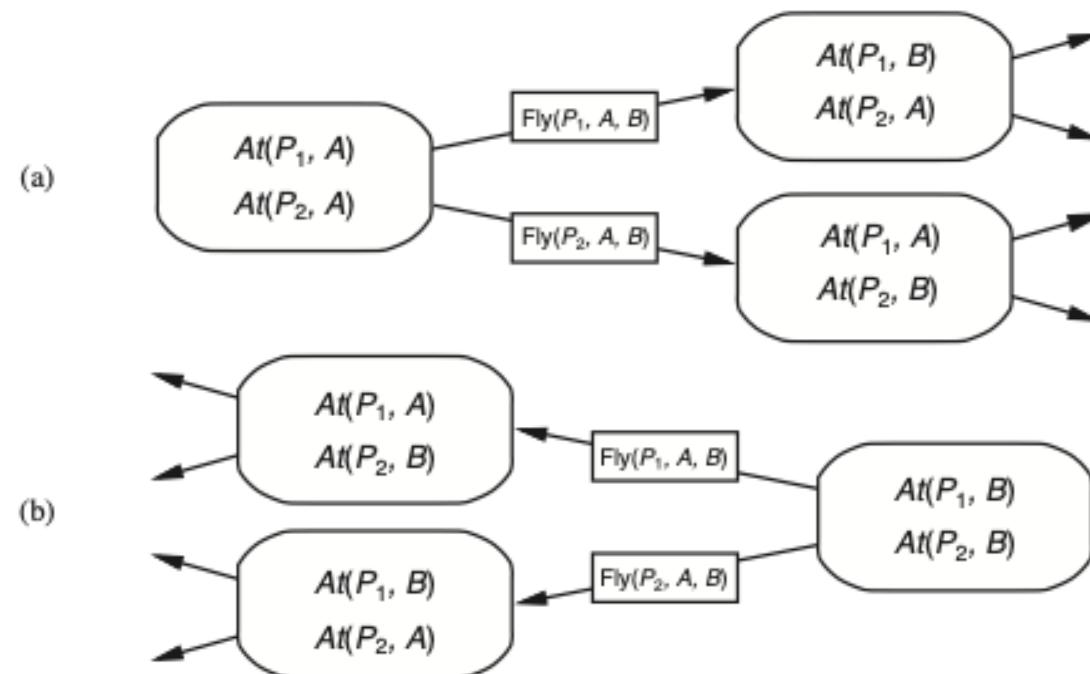
PRECOND: $\text{plane(P)} \wedge \text{airport(From)} \wedge \text{airport(To)} \wedge \text{at(P, From)}$

EFFECT: $\neg \text{at(P, From)} \wedge \text{at(P, To)}$

```
load(c1, p1, syd)
fly(p1, syd, mel)
unload(c1, p1, mel)
load(c2, p2, mel)
fly(p2, mel, syd)
unload(c2, p2, syd)
```

Simple Planning Algorithms

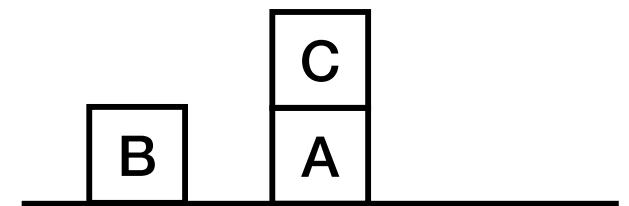
Forward search and goal regression



Sussman's Anomaly

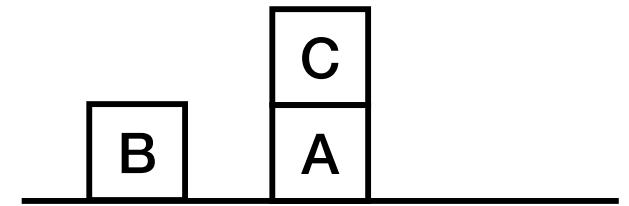
- Goal: $\text{on}(a, b) \wedge \text{on}(b, c)$
- Try achieving $\text{on}(a, b)$ first

[$\text{move}(c, a, \text{floor})$, $\text{move}(a, \text{floor}, b)$,
move(a,b,floor), $\text{move}(b, \text{floor}, c)$]



- Trying $\text{on}(b, c)$ first

[$\text{move}(b, \text{floor}, c)$, **move(b,c,floor)**,
 $\text{move}(c, a, \text{floor})$, $\text{move}(a, \text{floor}, b)$]



- Should be:

[$\text{move}(c, a, \text{floor})$, $\text{move}(b, \text{floor}, c)$, $\text{move}(a, \text{floor}, b)$]

WARPLAN

Warren, D. H. D. (1974). *Warplan: A system for generating plans*.
Memo No. 76, Department of Computational Logic, University of Edinburgh.

- Protect goals once achieved
- If an action undoes a goal
 - try moving new action backwards through plan before the action that achieved first goal
 - check that goals before and after are preserved

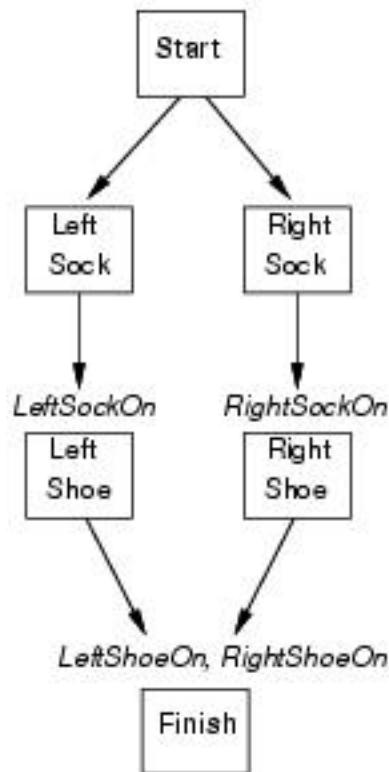
[move(c,a,floor), move(a,floor,b), **move(a,b,floor)**, ...]

[move(c,a,floor), ..., move(a,floor,b)]

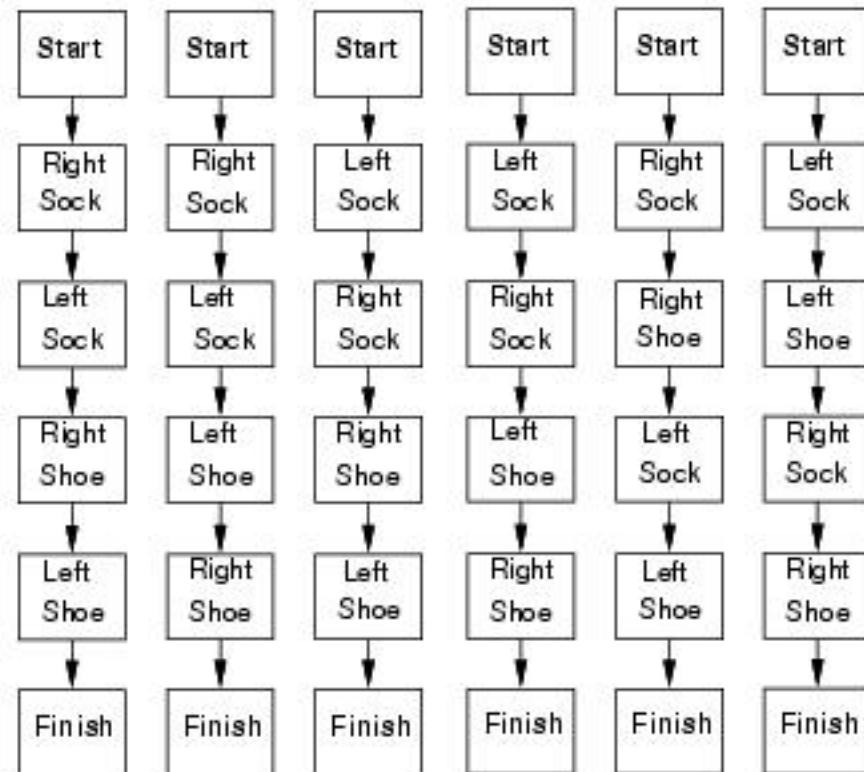
Try inserting plan for on(b,c) here

Partially Ordered Plans

Partial Order Plan:



Total Order Plans:



Partial-Order Planning

Init: $\text{Tire(Flat)} \wedge \text{Tire(Spare)} \wedge \text{at(Flat, Axle)} \wedge \text{at(Spare, Boot)}$

Goal: at (Spare, Axle)

Action Remove(obj, loc)

PRECOND: at(obj, loc)

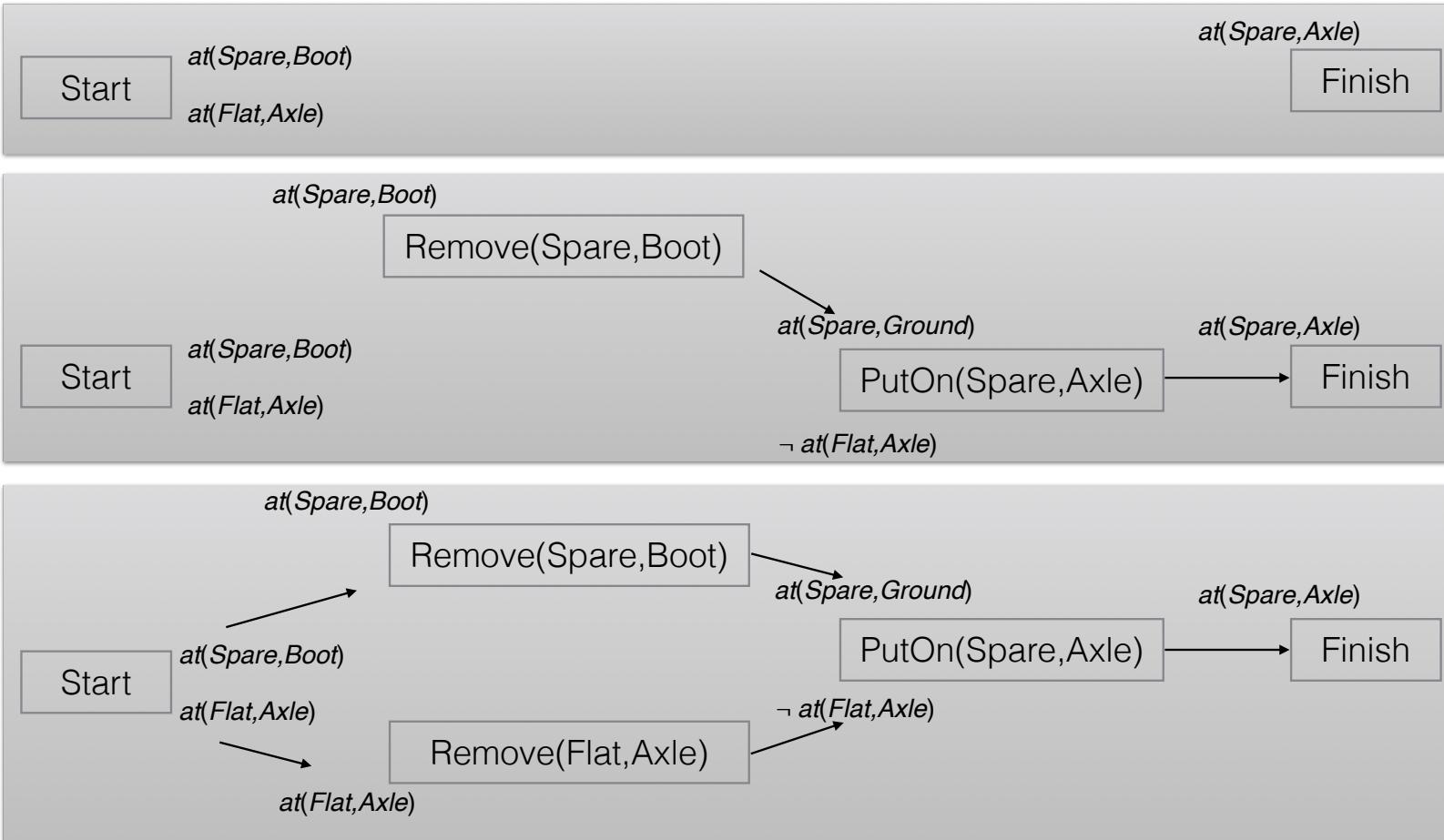
EFFECT: $\neg \text{at(obj, loc)} \wedge \text{at(obj, Ground)}$

Action PutOn(t, Axle)

PRECOND: $\text{Tire(t)} \wedge \text{at(t, Ground)} \wedge \neg \text{at(Flat, Axle)}$

EFFECT: $\neg \text{at(t, Ground)} \wedge \text{at(t, Axle)}$

Partial-Order Planning



Forward Planning

- Forward planners are now among the best.
- Use heuristics to estimate costs
- Possible to use heuristic search, like A*, to reduce search

Planning Graphs

- Use constraint solving to achieve better heuristic estimates
- Only for propositional problems
- Like consistency checking in CSP
 - preprocess constraints to create a planning graph
 - planning graph constrains possible states and actions
- Planning graph is NOT a plan
 - It constrains the range of possible plans

Planning Graph

- A sequence of levels that correspond to time steps in plan
 - Level 0 is initial state
 - Each level consists of:
 - Set of all literals that could be true at that time step
 - depending on actions executed in preceding time step
 - Set of all actions that could have their preconditions satisfied at that time step
 - depending on which literals are true

Mutual Exclusion

- Actions
 - Inconsistent effects: One action negates an effect of the other
 - Competing needs: Precondition of one action is mutually exclusive with a precondition of the other
- Literals
 - One literal is the negation of the other
 - Inconsistent support: Each possible pair of actions that could achieve the two literals is mutually exclusive

Example

Init: Have (Cake)

Goal: Have(Cake) \wedge Eaten(Cake)

Action: Eat (Cake)

PRECOND: Have(Cake)

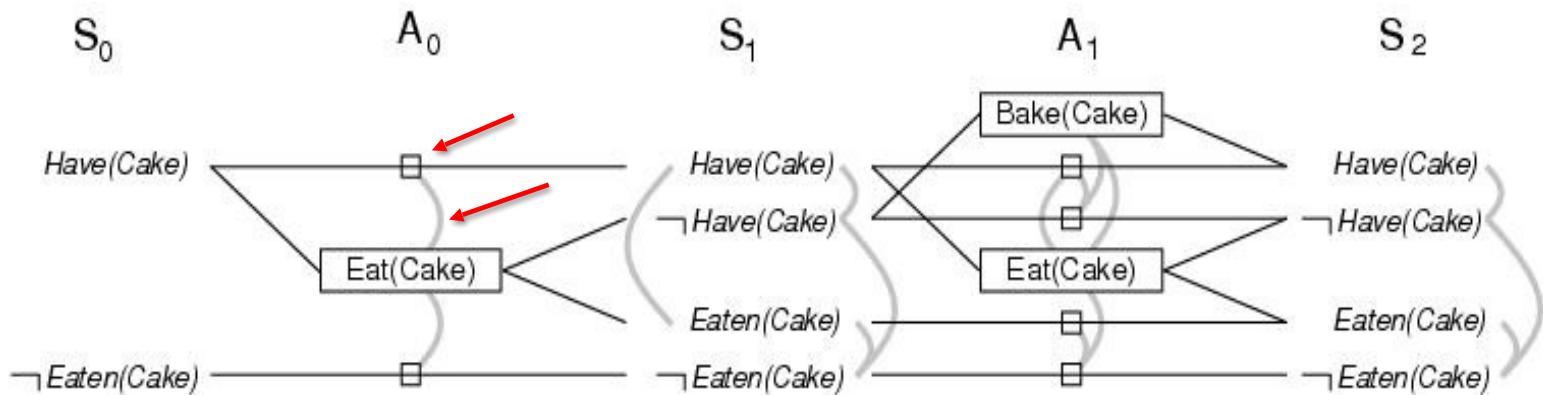
EFFECT: \neg Have(Cake) \wedge Eaten(Cake)

Action: Bake (Cake)

PRECOND: \neg Have(Cake)

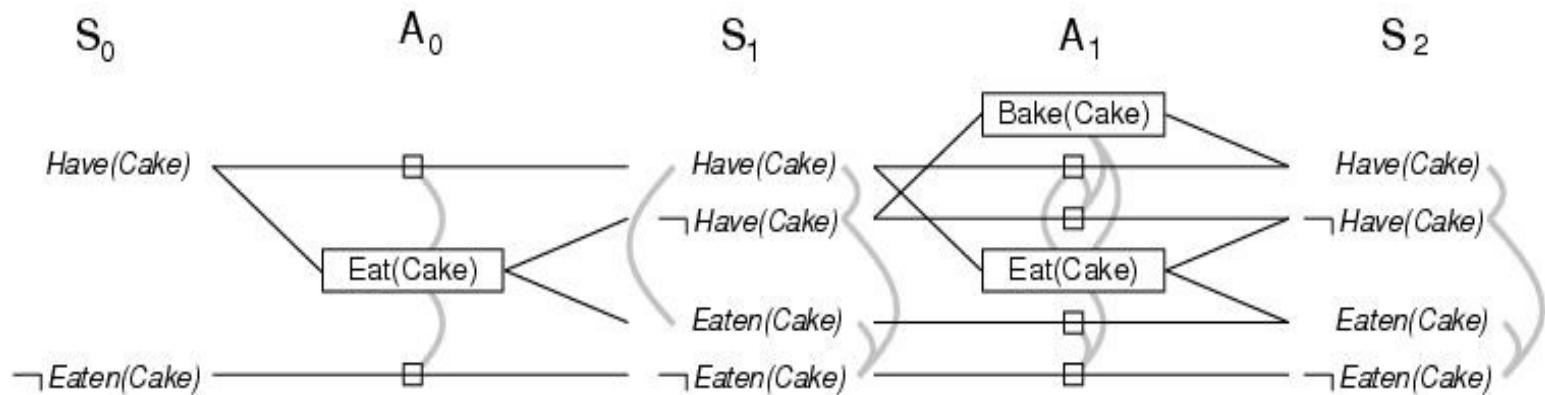
EFFECT: Have(Cake)

Cake Example



- Start at level S_0 and determine action level A_0 and next level S_1 .
 - A_0 – all actions whose preconditions are satisfied in the previous level.
 - Connect precondition and effect of actions $S_0 \rightarrow S_1$
 - Inaction is represented by persistent actions
- Level A_0 contains the actions that could occur
 - Conflicts between actions are represented by mutex links

Cake Example



- Level S_1 contains all literals that could result from picking any subset of actions in A_0
 - Conflicts between literals that cannot occur together are represented by mutex links.
 - S_1 defines multiple states and the mutex links are the constraints that define this set of states.
- Continue until goal is satisfied in level S_i , or no change in consecutive levels: levelled off

Planning Graphs and Heuristic Estimation

- Planning Graphs provide information about the problem
 - A literal that does not appear in final level of graph cannot be achieved by any plan.
 - Useful for backward search ($\text{cost} = \text{inf}$).
 - Level of appearance can be cost estimate of achieving goal literals = level cost.
 - Cost of a conjunction of goals:
 - max-level, sum-level and set-level heuristics.

Example: Spare tire problem

Init: at(Flat, Axle) \wedge at(Spare, Trunk)

Goal: at(Spare, Axle)

Action: Remove(Spare,Trunk)

PRECOND: at(Spare,Trunk)

EFFECT: \neg at(Spare,Trunk) \wedge at(Spare,Ground))

Action: Remove(Flat,Axle)

PRECOND: at(Flat,Axle)

EFFECT: \neg at(Flat,Axle) \wedge at(Flat,Ground))

Action: PutOn(Spare,Axle)

PRECOND: at(Spare,Ground) \wedge \neg at(Flat,Axle)

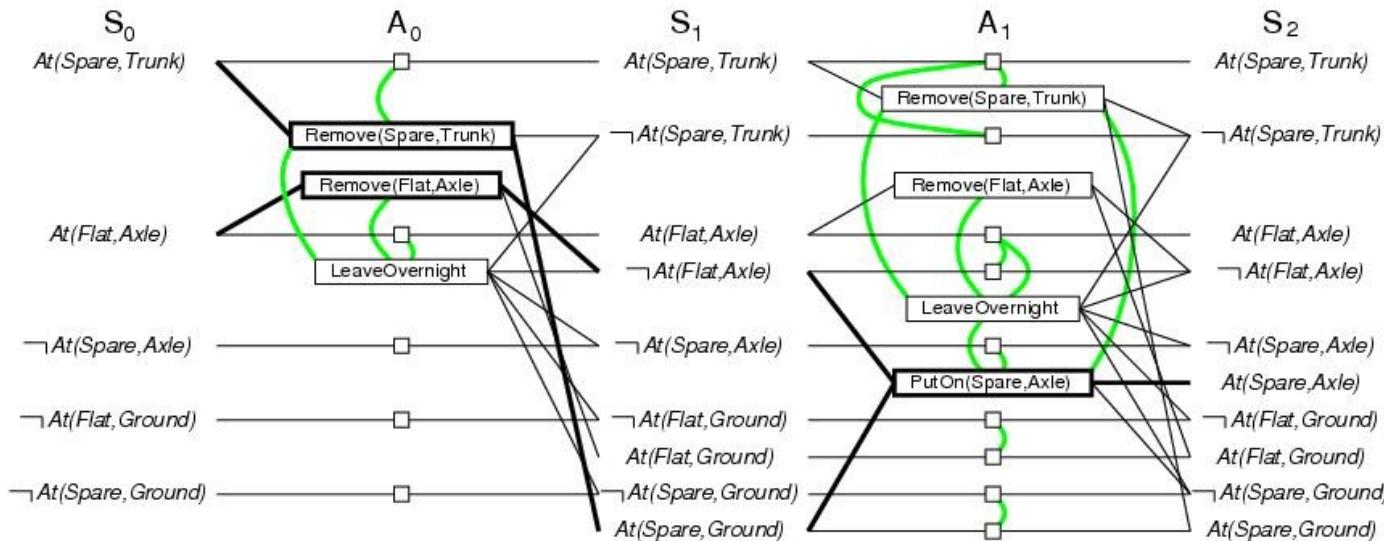
EFFECT: at(Spare,Axle) \wedge \neg at(Spare,Ground))

Action: LeaveOvernight

PRECOND:

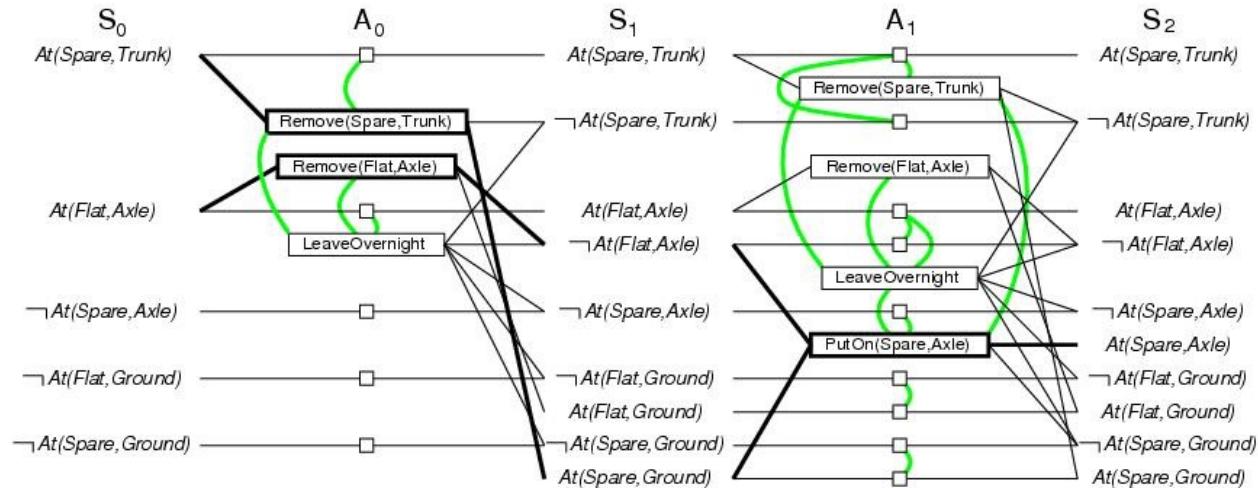
EFFECT: at(Flat,Axle) \wedge at(Spare,trunk) \wedge \neg at(Spare,Ground) \wedge \neg at(Spare,Axle) \wedge \neg at(Flat,Ground)

GRAPHPLAN Example



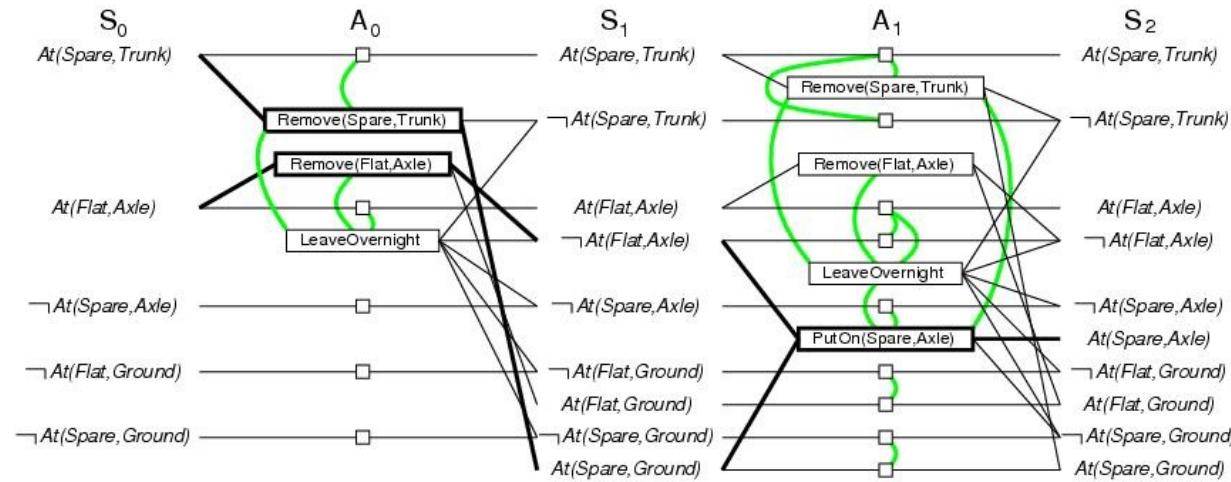
- Initially graph consist of literals from initial state and literals from closed world assumption (S_0).
- Add actions whose preconditions are satisfied by *expanding A_0* (next slide)
- Also add persistence actions and mutex relations
- Add the effects at level S_1
- Repeat until goal is in level S_i

GRAPHPLAN Example



- Expanding graph looks for mutex relations
 - Inconsistent effects
 - E.g. $Remove(Spare, Trunk) \wedge LeaveOverNight$ inconsistent because $at(Spare, Ground) \wedge \neg at(Spare, Ground)$
 - Interference
 - E.g. $Remove(Flat, Axle) \wedge LeaveOverNight$ but can't have $at(Flat, Axle)$ as PRECOND and $\neg at(Flat, Axle)$ as EFFECT
 - Competing needs
 - E.g. $PutOn(Spare, Axle) \wedge Remove(Flat, Axle)$ due to $at(Flat, Axle) \wedge \neg at(Flat, Axle)$
 - Inconsistent support
 - E.g. in S_2 , $at(Spare, Axle) \wedge at(Flat, Axle)$

GRAPHPLAN Example



- If goal literals exist in S_2 and are not mutex with any other \rightarrow solution might exist
- To extract solution use Boolean CSP to solve the problem or backwards search:
 - Initial state: last level of graph + goal literals of planning problem
 - Actions: select any set of non-conflicting actions that cover the goals in the state
 - Try to reach level S_0 such that all goals are satisfied
 - Cost = 1 for each action.

Extracting the Plan

- Heuristic forward search using A* can also find path from start to goal
 - Cost is based on level in graph

Summary

- Reasoning About Action
- STRIPS Planner
- Forward planning
- Regression Planning
- Partial Order Planning
- GraphPlan
- Planning as Constraint Satisfaction

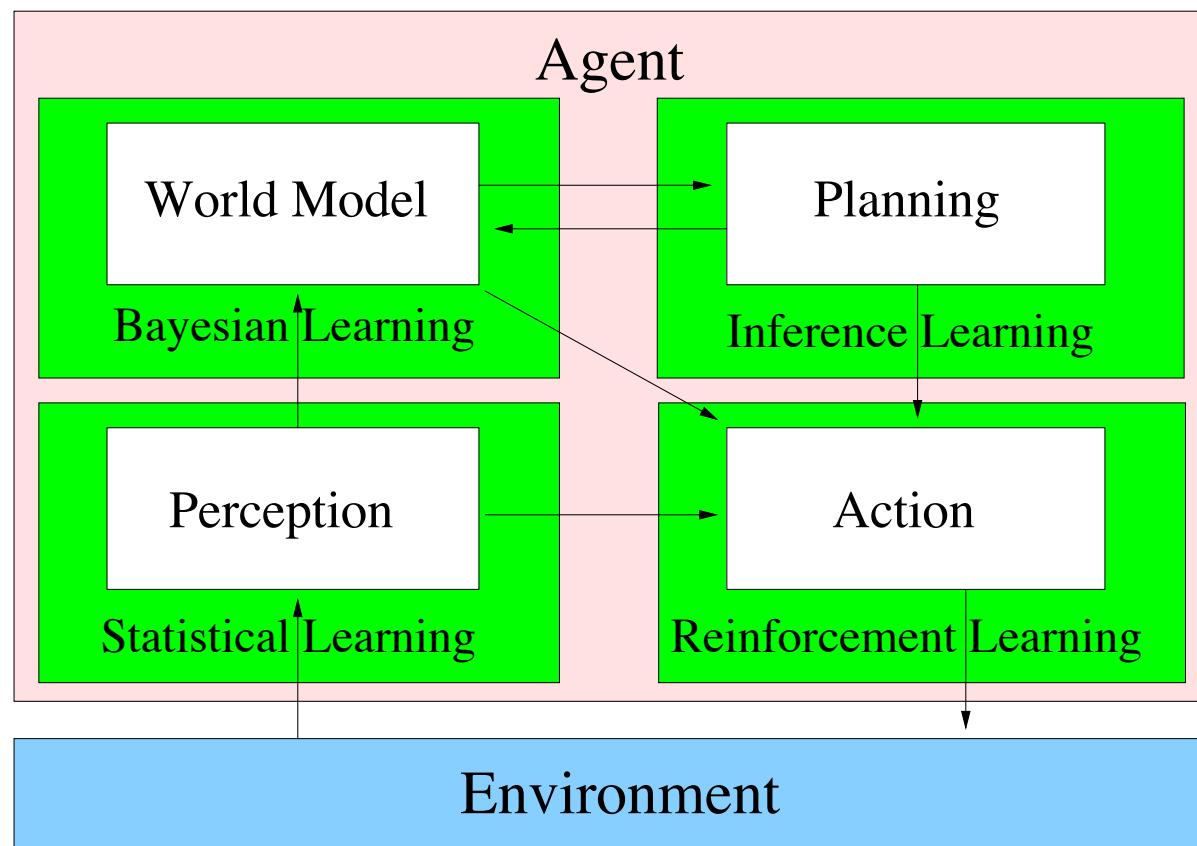
Planning Under Uncertainty Reinforcement Learning

COMP3411/9814: Artificial Intelligence

Lecture Overview

- Reinforcement Learning vs Supervised Learning
- Boxes
- Exploration vs Exploitation
- Q-Learning

Learning Agent



Types of Learning

- Supervised Learning
 - Agent is given examples of input/output pairs
 - Learns a function from inputs to outputs that agrees with the training examples and generalises to new examples
- Unsupervised Learning
 - Agent is only given inputs
 - Tries to find structure in these inputs
- Reinforcement Learning
 - Training examples presented one at a time
 - Must guess best output based on a reward, tries to maximise (expected) rewards over time

Environment Types

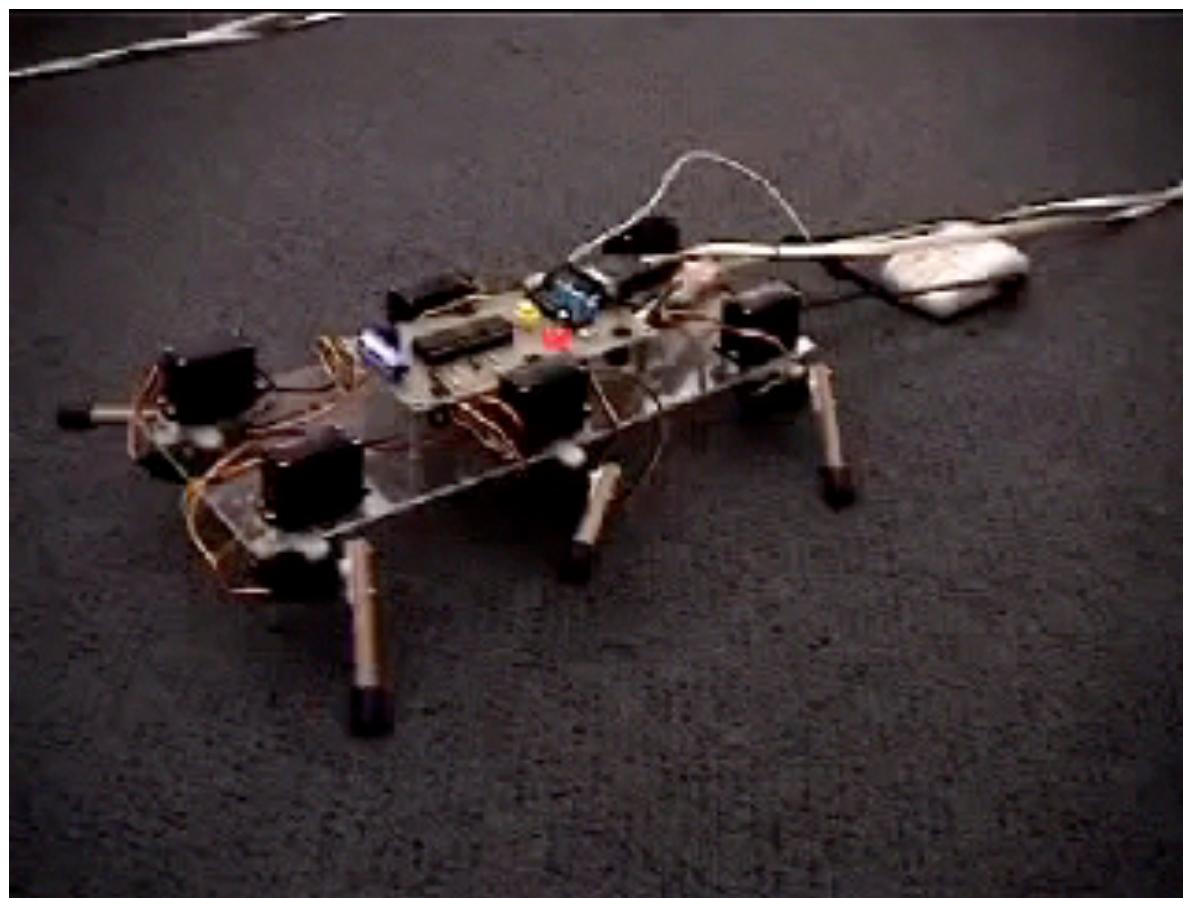
Environments can be:

- passive and deterministic
- passive and stochastic
- active and deterministic (chess)
- active and stochastic (backgammon, robotics)

Reinforcement Learning and Planning

- We start with reinforcement learning because it is also related to planning.
- RL tries to find the best way to act in uncertain and non-deterministic environments.

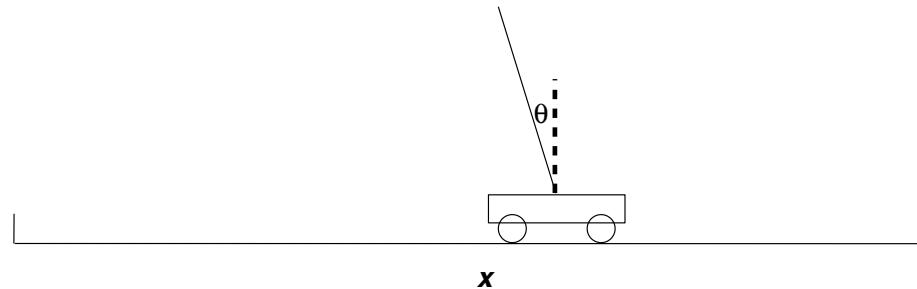
Stumpy - A Simple Learning Robot



Reinforcement Learning

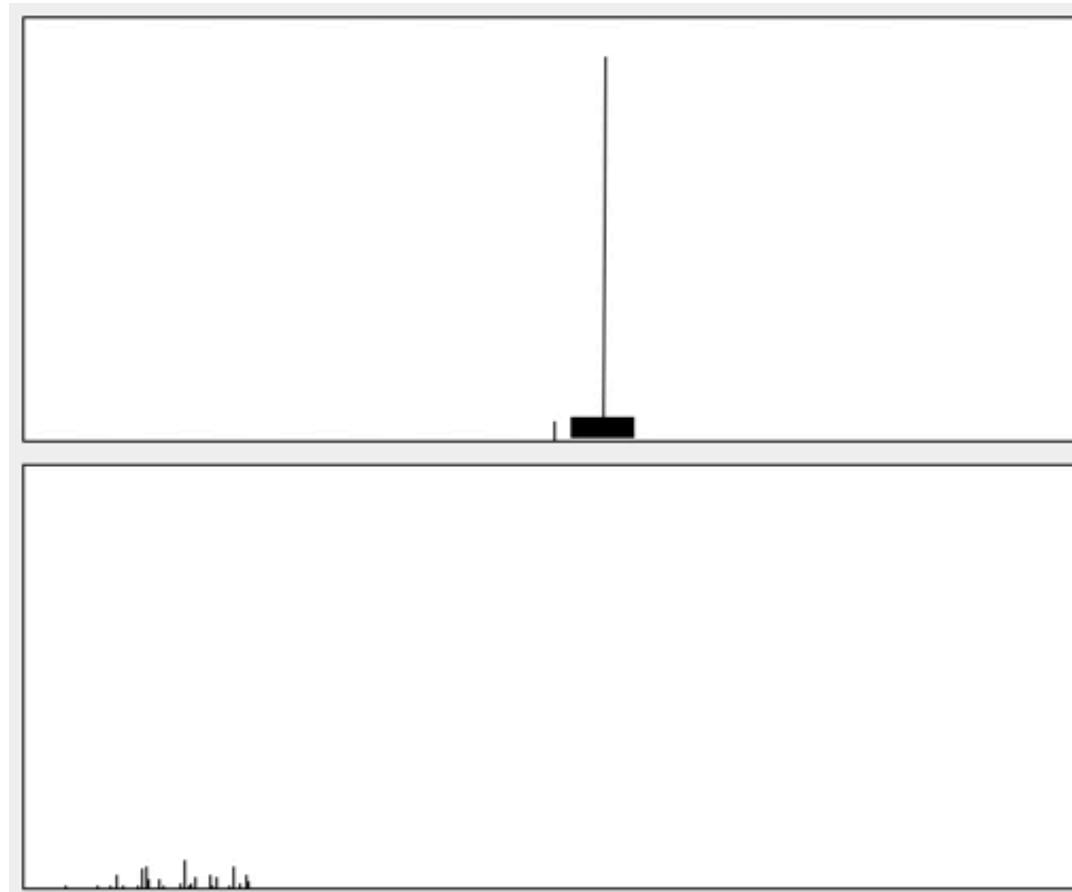
- “Stumpy” receives a *reward* after each action
 - Did it move forward or not?
- After each move, updates its *policy*
- Continues trying to maximise its reward

Pole Balancing



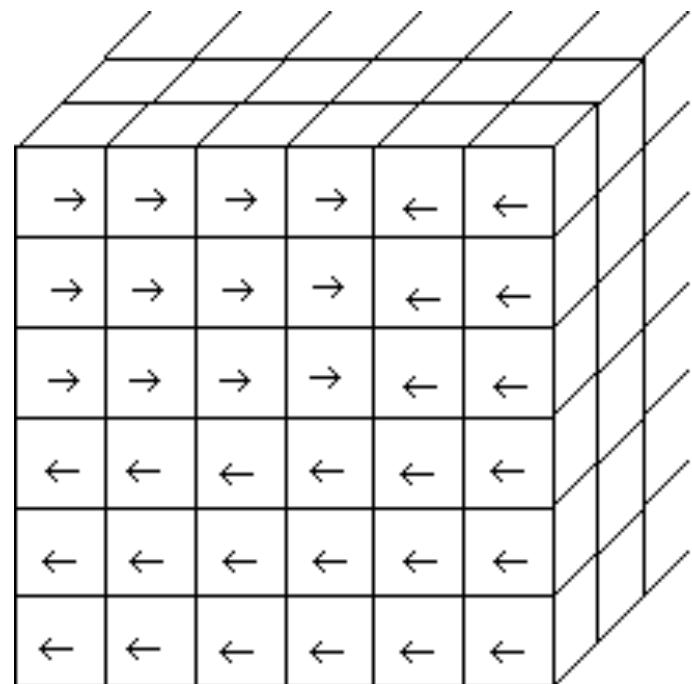
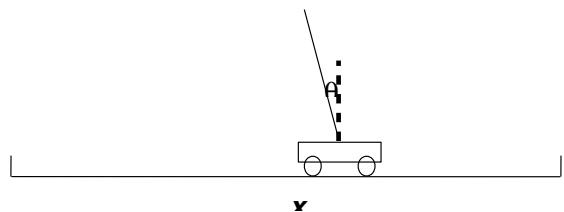
- Pole balancing can be learned the same way except that reward is only received at the end
 - after falling or hitting the end of the track

Pole Balancing



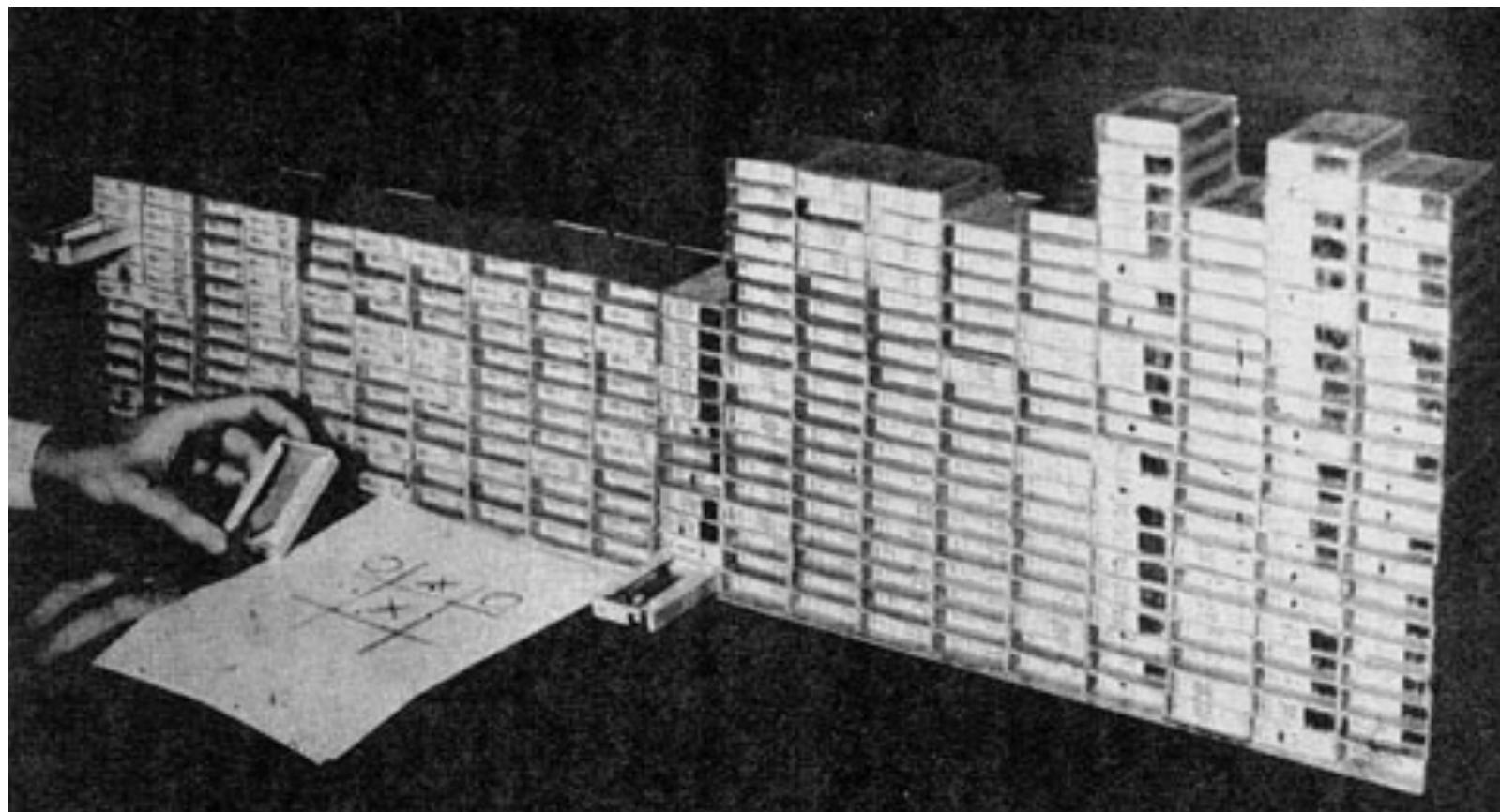
Boxes

- State variables: $\langle x, \dot{x}, \theta, \dot{\theta} \rangle$
- State space is discretised
- Each “box” represents a subset of state space
- When system lands in a box, execute action specified
 - left push
 - right push



MENACE

(Machine Educable Noughts and Crosses Engine – D. Michie, 1961)



Simulation

$$x_{t+1} = x_t + \tau \dot{x}_t$$

$$\dot{x}_{t+1} = \dot{x}_t + \tau \ddot{x}_t$$

$$\theta_{t+1} = \theta_t + \tau \dot{\theta}_t$$

$$\dot{\theta}_{t+1} = \dot{\theta}_t + \tau \ddot{\theta}_t$$

$$\ddot{x}_t = \frac{F_t + m_p l \left[\dot{\theta}_t^2 \sin \theta_t - \ddot{\theta}_t \cos \theta_t \right]}{m_c + m_p}$$

$$\ddot{\theta}_t = \frac{g \sin \theta_t + \cos \theta_t \left[\frac{-F_t - m_p l \dot{\theta}_t^2 \sin \theta_t}{m_c + m_p} \right]}{l \left[\frac{4}{3} - \frac{m_p \cos^2 \theta_t}{m_c + m_p} \right]}$$

$$m_c = 1.0 \text{ kg}$$

mass of cart

$$m_p = 1.0 \text{ kg}$$

mass of pole

$$l = 0.5 \text{ m}$$

*distance of centre of mass
of pole from the pivot*

$$g = 9.8 \text{ ms}^{-2}$$

acceleration due to gravity

$$F_t = \pm 10 \text{ N}$$

force applied to cart

$$t = 0.02 \text{ s}$$

time interval of simulation

The BOXES Algorithm

- Each box contains statistics on performance of controller, which are updated after each failure
 - How many times each action has been performed (*usage*)
 - The sum of lengths of time the system has survived after taking a particular action (*LifeTime*)
- Each sum is weighted by a number less than one which places a discount on earlier experience.

Exploration / Exploitation Tradeoff

- Most of the time choose what we think is the “best” action.
- But to learn, must occasionally choose something different from preferred action

Update Rule

if an action has not been tested

choose that action

else if $\frac{LeftLife}{LeftUsage^k} > \frac{RightLife}{RightUsage^k}$

choose left

k is a bias to force exploration
e.g. $k = 1.4$

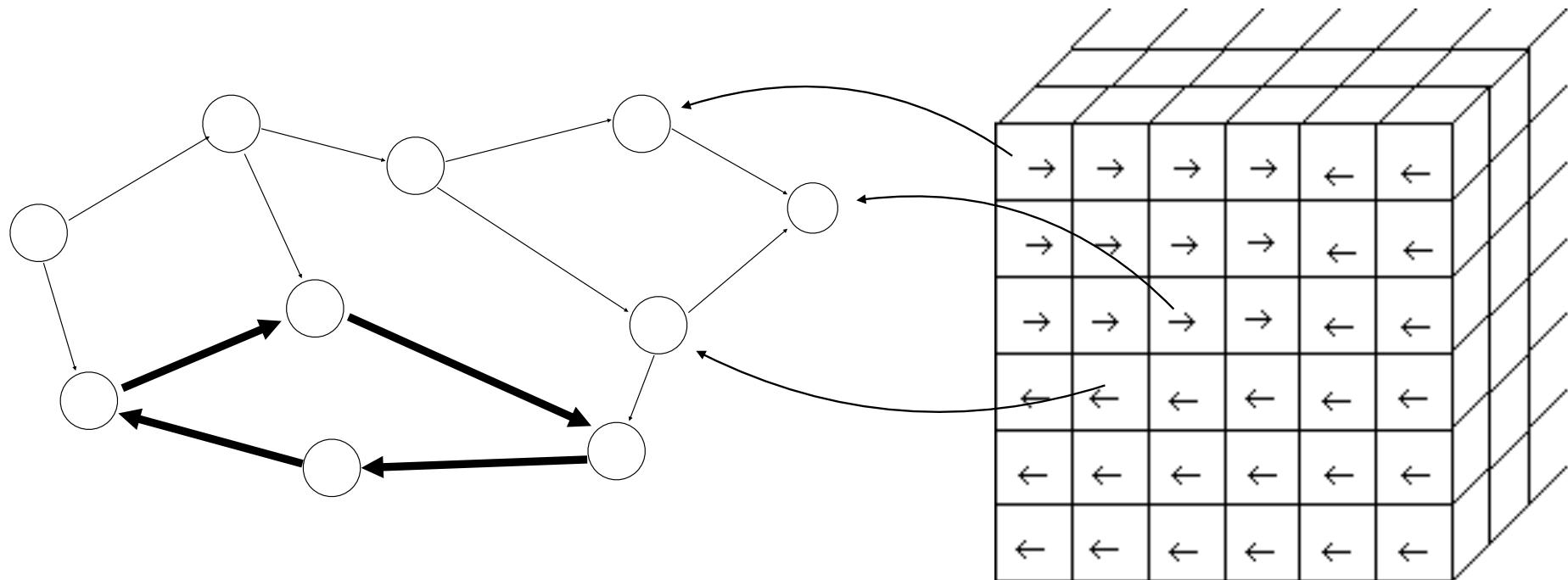
else

choose right

Performance

- BOXES is fast
 - Only 75 trials, on average, to reach 10,000 time steps
- But only works for *episodic* problems
 - i.e. has a specific termination
- Doesn't work for continuous problems like Stumpy

State Transition Graph



States and Actions

- Each node is a *state*
- *Actions* cause transitions from one state to another
- A *policy* is the set of transition rules
 - i.e. which action to apply in a given state
- Agent receives a *reward* after each action
- Actions may be non-deterministic
 - Same action may not always produce same state

Reinforcement Learning Framework

- An agent interacts with its environment.
- There is a set of *states*, S , and a set of *actions*, A .
- At each time step t , agent is in state s_t .
- It must choose an action a_t , which changes state to
- $s_{t+1} = \delta(s_t, a_t)$ and receives reward $r(s_t, a_t)$.
 - The world is non-deterministic, i.e. an action may not always take the system to the same state
 - δ , and therefore r , can be multi-valued, with a random element
- Aim is to find an *optimal policy* $\pi : S \rightarrow A$ that maximises the cumulative reward.

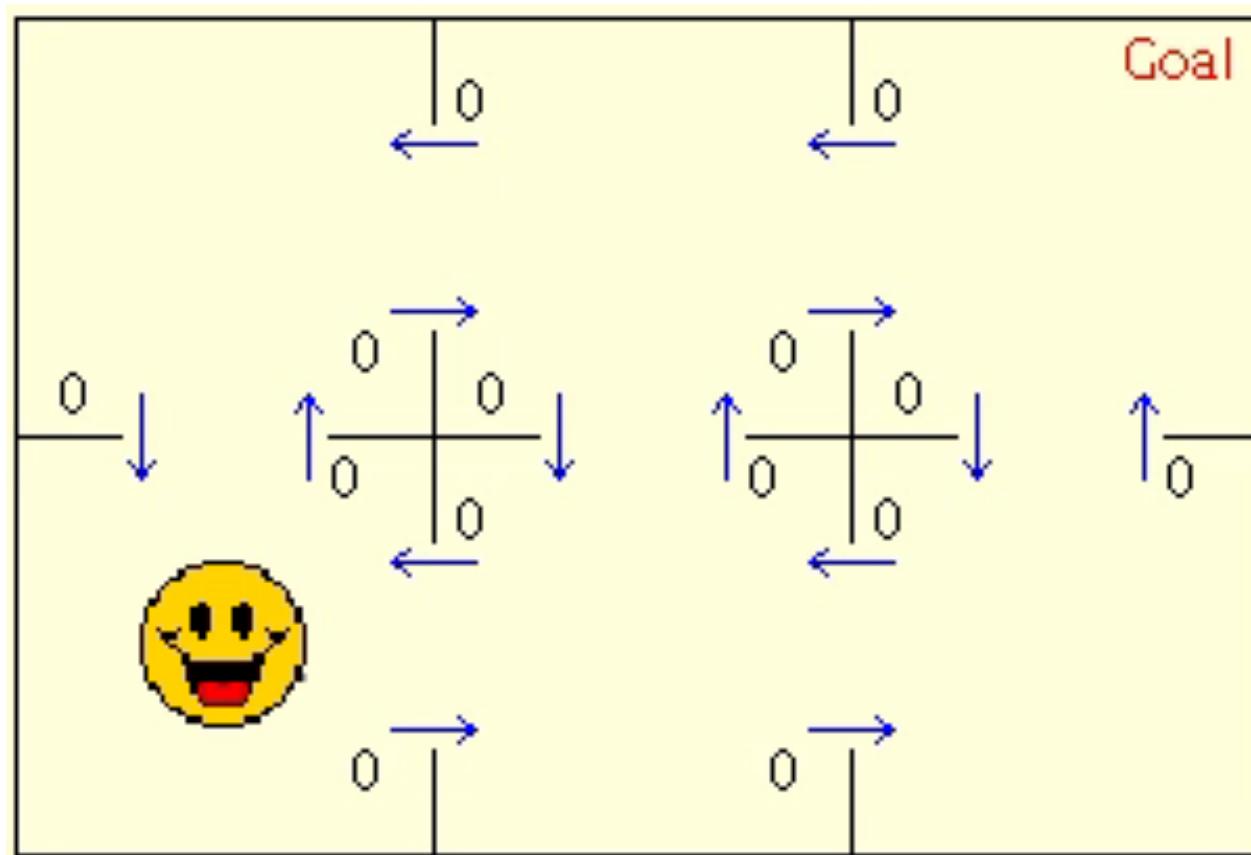
Markov Decision Process (MDP)

- Assume that current state has all the information needed to decide which action to take
- Actions are assumed to have a fixed duration

Learning an MDP

- The agent initially only knows the set of possible states and the set of possible actions.
- The dynamics, $P(s'|a,s)$, and the reward function, $R(s,a)$, are not given to the agent.
- $P(s'|a,s)$ the probability of the agent transitioning into state s' given that the agent is in state s and does action a
- After each action, the agent observes the state it is in and receives a reward.
- Assume that current state has all the information needed to decide which action to take

Grid World Example



Expected Reward

- Try to maximise expected future reward:

$$\begin{aligned} V^\pi(s_t) &= r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots \\ &= \sum_{i=0}^{\infty} \gamma^i r_{t+i} \end{aligned}$$

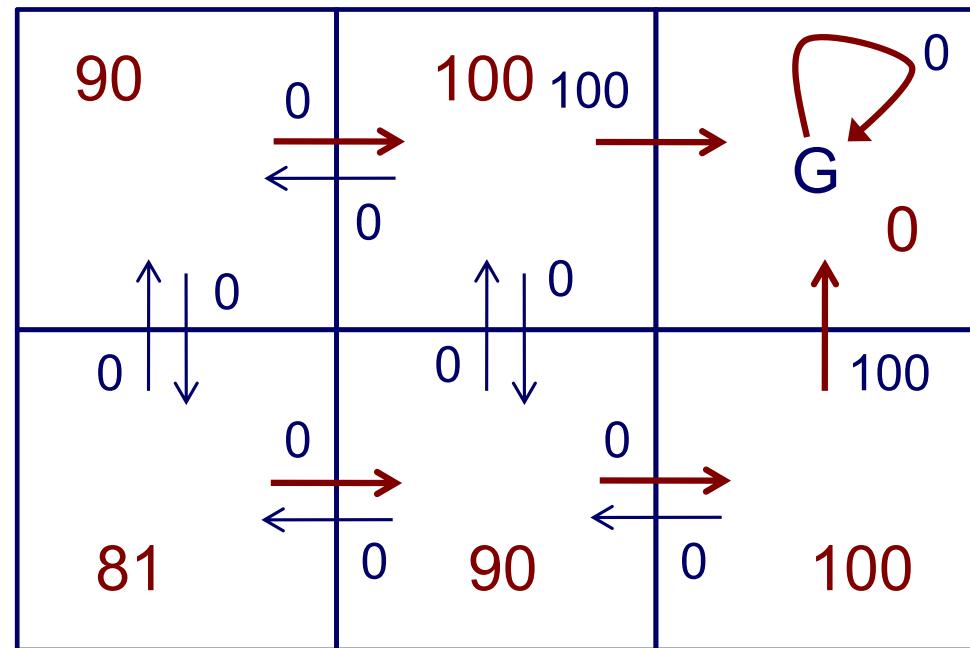
- $V^\pi(s_t)$ is the value of state s_t under policy π
- γ is a discount factor [0..1]

Value Function

- $V^\pi(s)$ is the expected value of following policy π in state s
- $V^*(s)$ be the maximum discounted reward obtainable from s .
 - i.e. the value of following the optimal policy
- We make the simplification that actions are deterministic, but we don't know which action to take.
 - Other RL algorithms relax this assumption

Value Function

- The red arrows show, π^* , is the optimal policy, with $\gamma = 0.9$
- $V^*(s)$ values shown in red



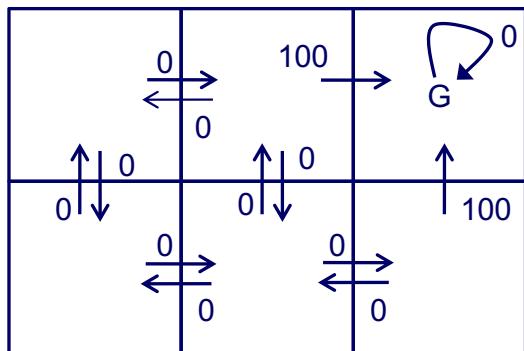
Q Value

- How to choose an action in a state?

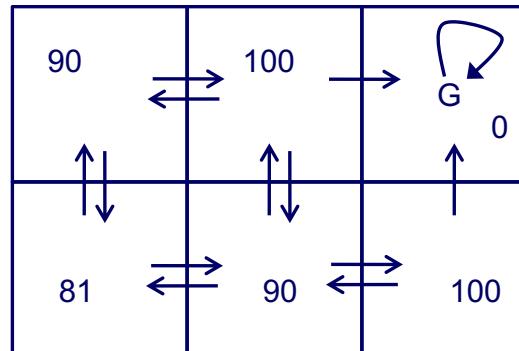
$$Q(s, a) = r(s, a) + \gamma V^*(s')$$

- The Q value for an action, a , in a state, s , is the immediate reward for the action plus the discounted value of following the optimal policy after that action
- V^* is value obtained by following the optimal policy
- $s' = \delta(s, a)$ is the succeeding state, assuming the optimal policy

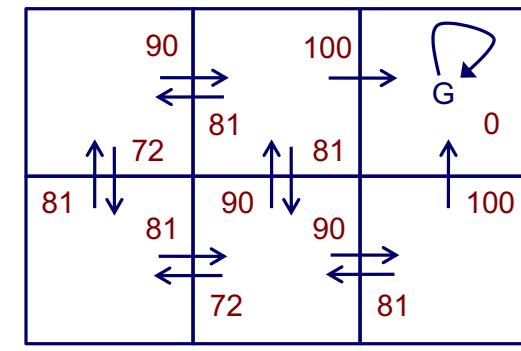
Q values



$r(s, a)$ (immediate reward) values



$V^*(s)$ values



$Q(s, a)$ values

$$\gamma = 0.9$$

Q Learning

initialise $Q(s,a) = 0$ for all s and a

observe current state s

repeat

 select an action a and execute it

 observe immediate reward r and next state s'

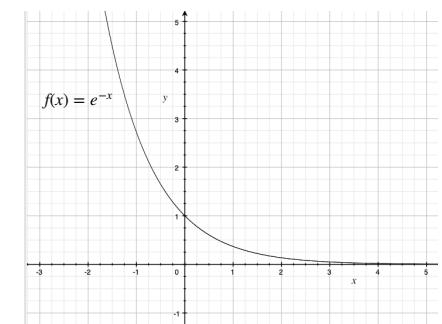
$$Q(s,a) \leftarrow r + \max_{a'} Q(s',a')$$

$$s \leftarrow s'$$

Exploration vs Exploitation

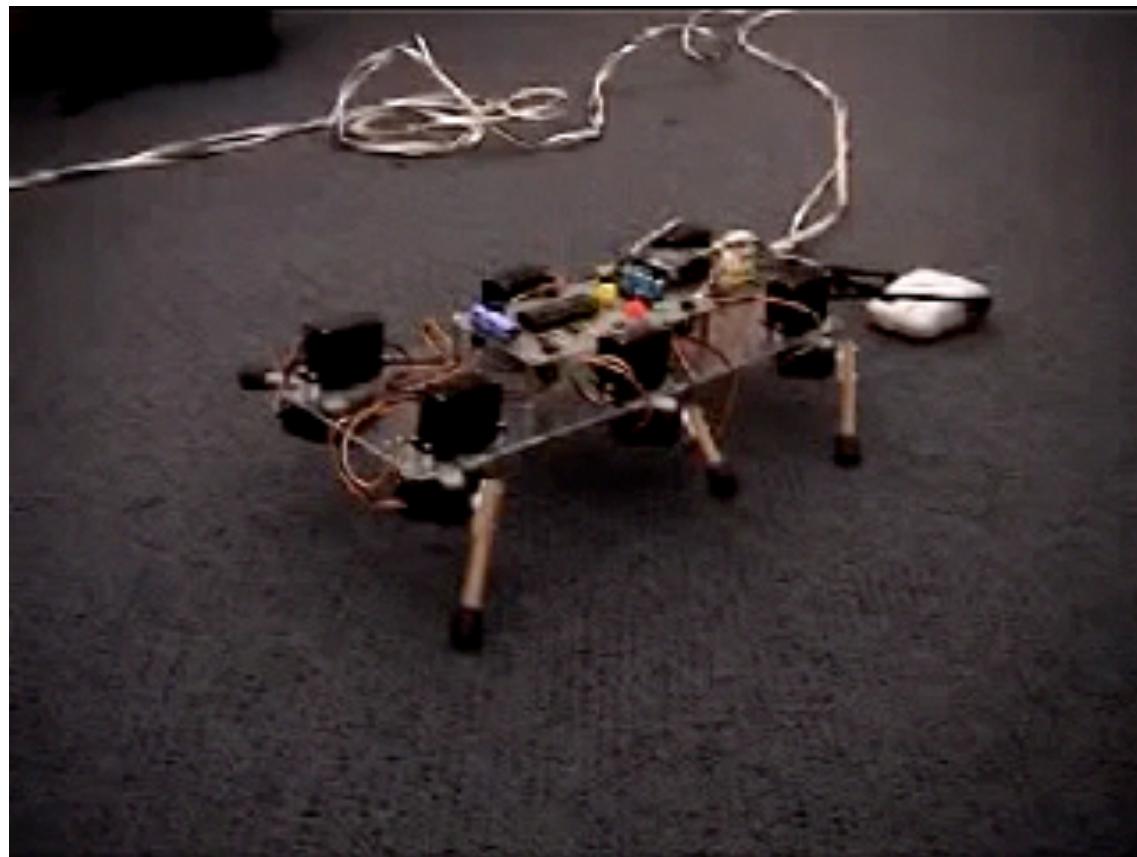
- How do you choose an action?
 - Random
 - Pick the current “best” action
 - Combination:
 - most of the time pick the best action
 - occasionally throw in random action
 - Boltzmann exploration:

$$\pi(s_t, a) \simeq e^{\frac{-Q_t(s_t, a)}{\tau}}$$



- Decrease τ over time
- High τ : exploration
 - Low τ : exploitation

Stumpy after 30 minutes



Reinforcement Learning Variants

- There are *many* variations on reinforcement learning to improve search.
- RL is one of the components of alphaZero, which is currently the best Go and Chess player
- Used to learn helicopter aerobatics

Background

- Reinforcement learning is based in earlier work in optimisation: dynamic programming
- Text book: Sutton & Barto

Learning and Decision Trees

COMP3411/9814: Artificial Intelligence

Lecture Overview

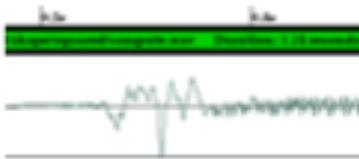
- Learning agents
- Inductive learning
- Decision tree learning

Turing's Child Machine

“Instead of trying to produce a programme to simulate the adult mind, why not rather try to produce one which simulates the child's? If this were then subjected to an appropriate course of education one would obtain the adult brain. Presumably the child-brain is something like a notebook as one buys from the stationers. Rather little mechanism, and lots of blank sheets... Our hope is that there is so little mechanism in the child-brain that something like it can be easily programmed. The amount of work in the education we can assume, as a first approximation, to be much the same as for the human child.”

Machine Learning Applications

Mining Databases



Speech Recognition

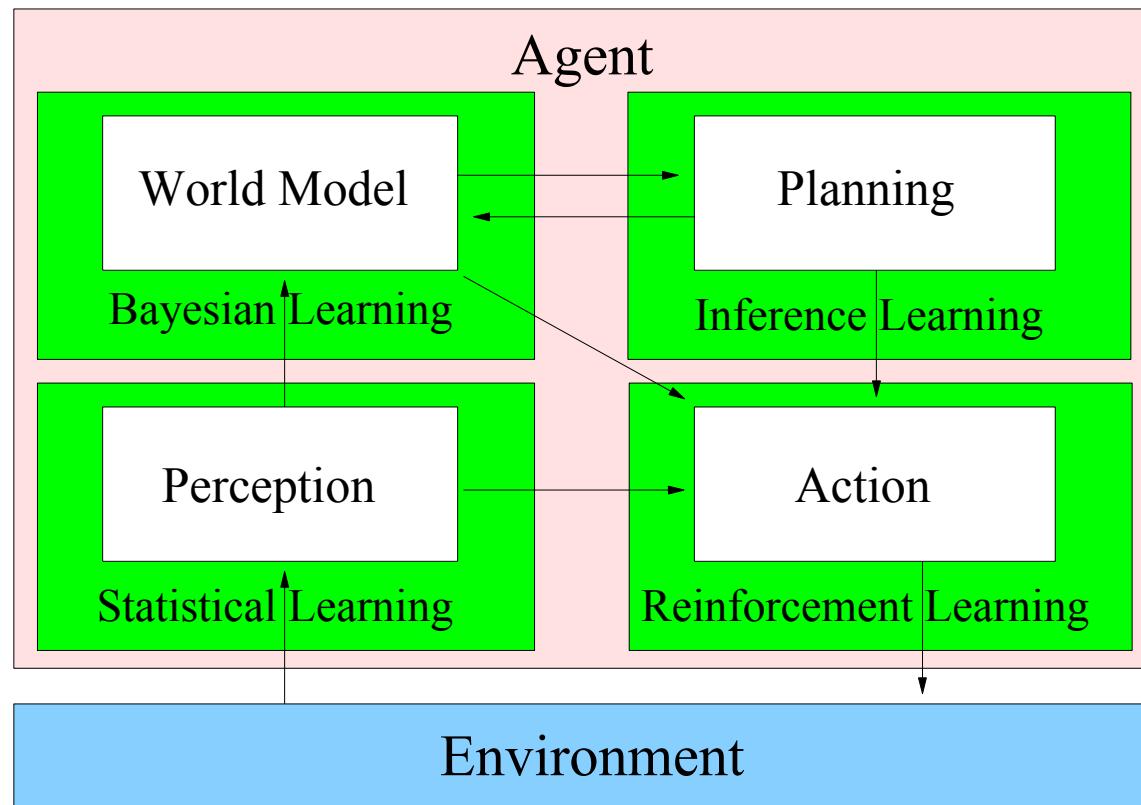


Control learning

Text analysis

Peter H. van Oppen, Chairman of the Board & Chief Executive Officer
Mr. van Oppen has served as chairman of the board and chief executive officer of ADIC since its acquisition by Interpoint in 1994 and a director of ADIC since 1986. Until its acquisition by Crane Co. in October 1996, Mr. van Oppen served as chairman and chief executive officer of Interpoint. Prior to 1986, Mr. van Oppen worked as a consulting manager at Price Waterhouse LLP and at Bain & Company in Boston and London. He has additional experience in medical electronics and venture capital. Mr. van Oppen also serves as a director of Creative Microooks, Inc. and Spacelabs Medical, Inc.. He holds a B.A. from Whitman College and an M.B.A. from Harvard Business School, where he was a Baker Scholar.

Learning Agent



Learning Agents

- Improve performance on future tasks after making observations about the world
- Design of a learning element is affected by
 - Which components of the performance element are to be learned
 - What feedback is available to learn these components
 - What representation is used for the components

Types of Learning

- Supervised Learning
 - Agent given examples of inputs and outputs
 - Learns a function from inputs to outputs that agrees with training examples and generalises to new examples
- Unsupervised Learning
 - Agent only given inputs
 - Tries to find structure in inputs
- Reinforcement Learning
 - Agent given a reward
 - Learns to maximise (expected) rewards over time

Supervised Learning

- Given a **training set** and a **test set**
 - each has set of examples
 - example has **attributes** and a **class, or target value**
 $\langle x_1, x_2, \dots, x_n, y \rangle$ y may be discrete or continuous
- Agent given attributes and class for each example in training set
 - ➡ Try predict class of each example in test set
- Many supervised learning methods, e.g.:
 - Decision Tree
 - Neural Network
 - Support Vector Machine, etc.

Supervised Learning

- Training set:
 - Examples may be presented all at once (batch) or in sequence (online)
 - Examples may be presented at random or in time order (stream)
 - Learner **cannot** use the test set **at all** in defining the model
- Model is evaluated on predicting output for each example in **test set**

Supervised Learning Methodology

1. “Feature engineering” – select relevant features
2. Choose representation of input features and outputs
3. Pre-process to extract features from raw data
4. Choose learning method(s) to evaluate
5. Choose training regime (including parameters)
6. Evaluation
 1. Choose baseline for comparison
 2. Choose type of internal validation, e.g. cross-validation
 3. Sanity check results with human expertise, other benchmark

Inductive Learning

Simplest form: learn a function from examples

f is the **target function**

An **example** is a pair $(x, f(x))$

Problem: find a **hypothesis h**

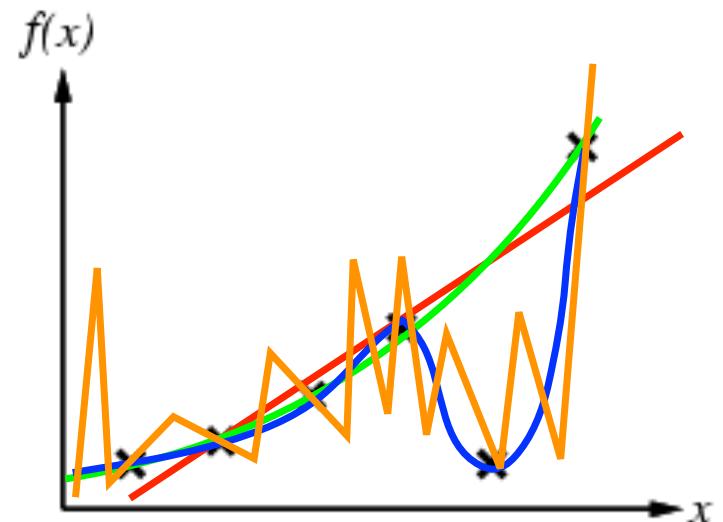
such that $h \approx f$

given a **training set** of examples

- Ignores prior knowledge
 - Assumes examples are given

Inductive Learning Method

- Construct h to agree with f on training set
- (h is **consistent** if it agrees with f in all examples)
- E.g., curve fitting
 - Which curve best fits data?

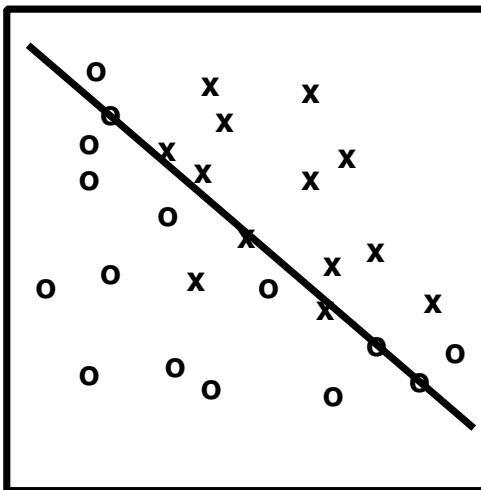


Ockham's razor

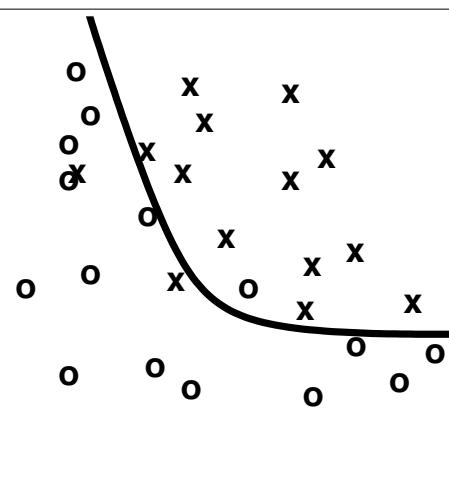
“The most likely hypothesis is the simplest one consistent with the data.”

Ockham's razor

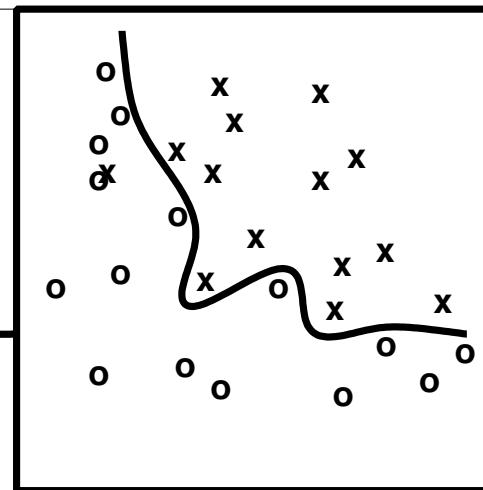
“The most likely hypothesis is the simplest one consistent with the data.”



inadequate



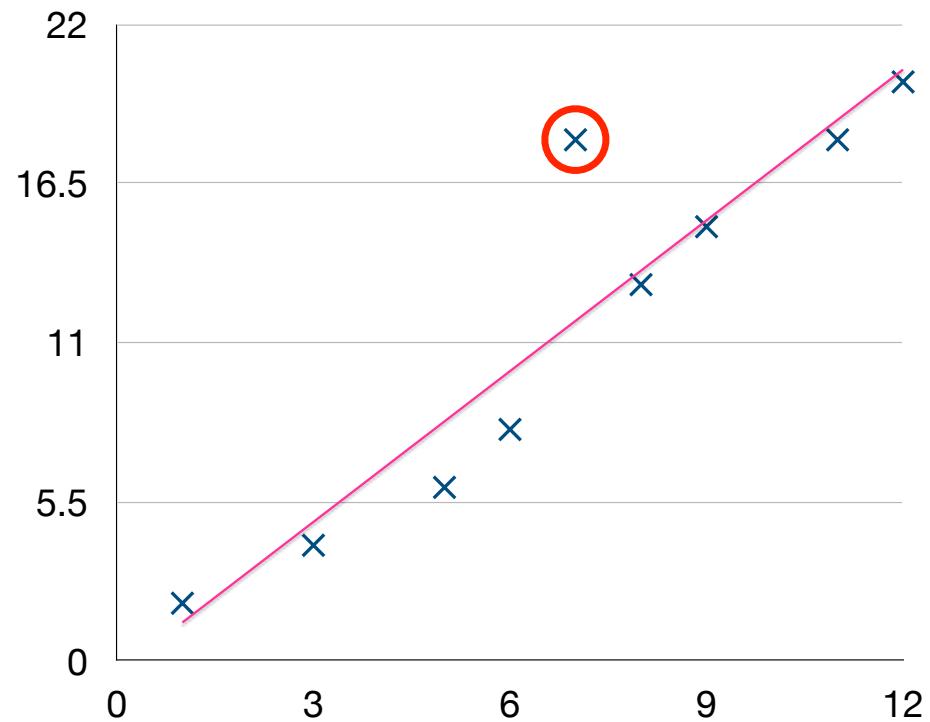
good compromise



over-fitting

- Can have **noise** in data
- Need to tradeoff simplicity and accuracy

Outliers



Supervised Learning (again)

Given:

- a set of **training examples**, each with a set of **features** and **target value (or class)**:

$\langle x_1, x_2, \dots, x_n, y \rangle$

- a new example, where only the values for the input features are given

predict the target value of the new example.

Decision Tree Learning

Learning decision trees

Problem: decide whether to wait for a table at a restaurant, based on the following attributes:

1. **Alternate**: is there an alternative restaurant nearby?
2. **Bar**: is there a comfortable bar area to wait in?
3. **Fri/Sat**: is today Friday or Saturday?
4. **Hungry**: are we hungry?
5. **Patrons**: number of people in the restaurant (None, Some, Full)
6. **Price**: price range (\$, \$\$, \$\$\$)
7. **Raining**: is it raining outside?
8. **Reservation**: have we made a reservation?
9. **Type**: kind of restaurant (French, Italian, Thai, Burger)
10. **WaitEstimate**: estimated waiting time (0-10, 10-30, 30-60, >60)

Restaurant Training Data

- Examples described by **attribute values** (Boolean, discrete, continuous)
 - E.g., situations where I will/won't wait for a table:

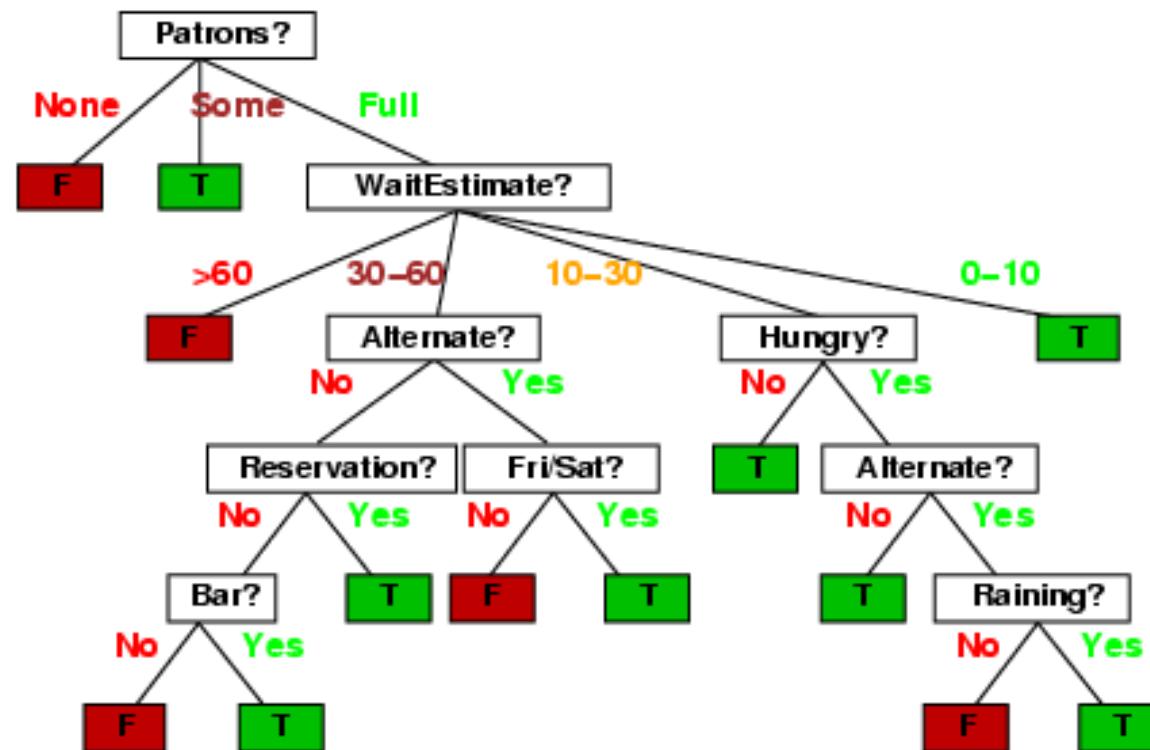
class
value



	Alt	Bar	F/S	Hun	Pat	Price	Rain	Res	Type	Est	Wait?
X ₁	T	F	F	T	Some	\$\$\$	F	T	French	0–10	T
X ₂	T	F	F	T	Full	\$	F	F	Thai	30–60	F
X ₃	F	T	F	F	Some	\$	F	F	Burger	0–10	T
X ₄	T	F	T	T	Full	\$	F	F	Thai	10–30	T
X ₅	T	F	T	F	Full	\$\$\$	F	T	French	>60	F
X ₆	F	T	F	T	Some	\$\$	T	T	Italian	0–10	T
X ₇	F	T	F	F	None	\$	T	F	Burger	0–10	F
X ₈	F	F	F	T	Some	\$\$	T	T	Thai	0–10	T
X ₉	F	T	T	F	Full	\$	T	F	Burger	>60	F
X ₁₀	T	T	T	T	Full	\$\$\$	F	T	Italian	10–30	F
X ₁₁	F	F	F	F	None	\$	F	F	Thai	0–10	F
X ₁₂	T	T	T	T	Full	\$	F	F	Burger	30–60	T

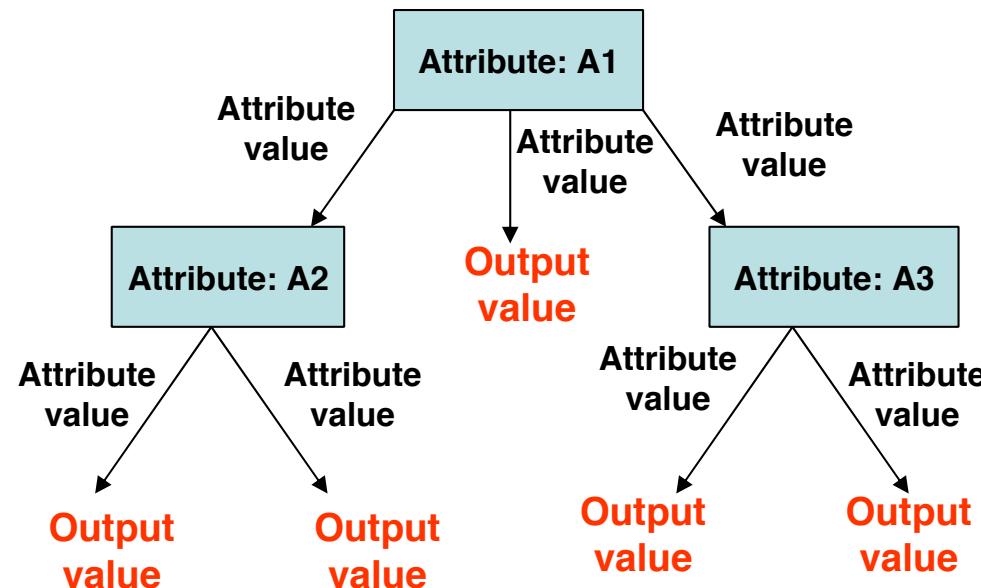
Decision Trees

- One possible representation for hypotheses



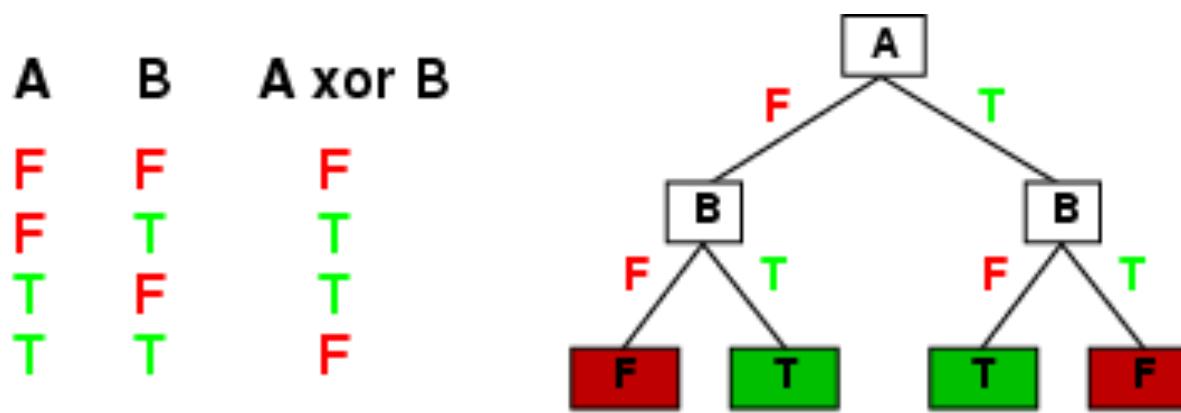
Decision Trees

- Output can be multi-valued, not just binary



Expressiveness

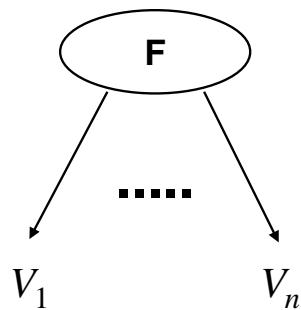
- Decision trees can express any function of the input attributes.
 - E.g., for Boolean functions, truth table row → path to leaf:



- There is a consistent decision tree for any training set with one path to leaf for each example (unless f nondeterministic in x) but it probably won't generalise to new examples
- Prefer to find more **compact** decision trees

ID3 (Quinlan)

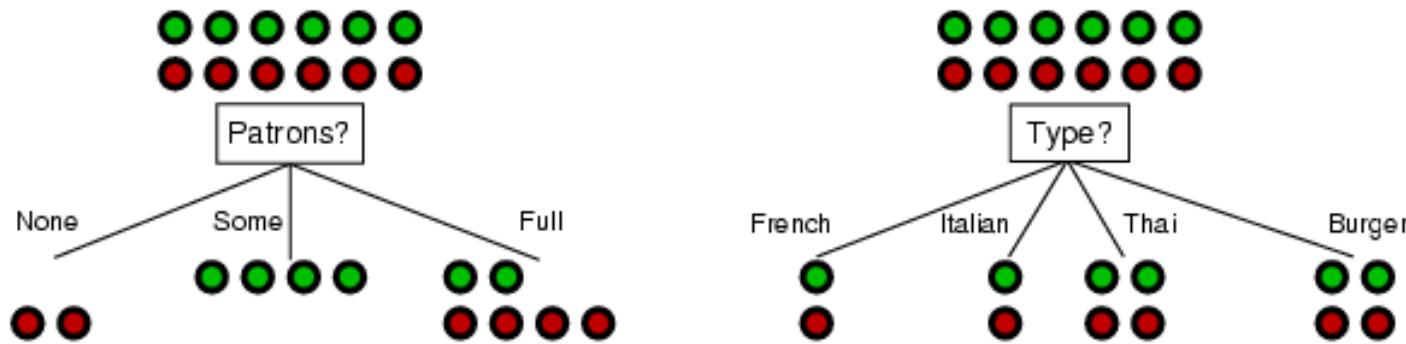
- The algorithm operates over a set of training instances, C .
- If all instances in C are in class P , create a node P and stop, otherwise select a feature F and create a decision node.
- Partition the training instances in C into subsets according to the values of V .
- Apply the algorithm recursively to each of the subsets C_i



Generalisation

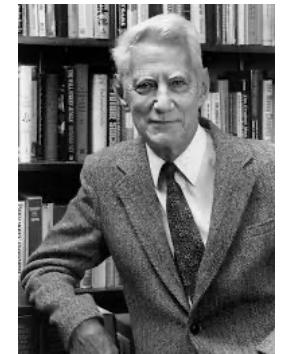
- Training data must not be **inconsistent**
 - see later how to handle inconsistent data
- Can split attributes in any order and still produce a tree that correctly classifies all examples in training set
- However, want a tree that is likely to **generalise** to correctly classify (unseen) examples in **test set**.
- Prefer **simpler** hypothesis, i.e. a smaller tree.
- How can we choose attributes to produce a small tree?

Choosing an attribute



- Patrons is a more “informative” attribute than Type, because it splits the examples more nearly into sets that are “all positive” or “all negative”.
- “Informativeness” can be quantified using the mathematical concept of “[entropy](#)”.
- Trees can be built by minimising entropy at each step

Information Gain



Claude Shannon
(1916 – 2001)

Information gain is based on information theory concept called *Entropy*

In information theory, the **Shannon entropy** or **information entropy** is a measure of the **uncertainty** associated with a random variable.

- It quantifies the information contained in a message, usually in bits or bits/symbol.
- It is the minimum message length necessary to communicate information.

Choosing Attributes

- The order in which attributes are chosen determines how complicated the tree is.
- ID3 uses information theory to determine the most informative attribute.
- The information content of a message is the inverse of the probability of receiving the message

$$\text{information}(M) = - \text{probability}(M)$$

- Taking log (base 2) makes information correspond to the number of bits required to encode a message:

$$\text{information}(M) = - \log_2 \text{probability}(M)$$

Entropy

- Entropy is a measure of how much information we **gain** when the class value is revealed to us.
- In decision tree learning, when we partition a dataset by a particular attribute the resulting entropy is lower.
 - An entropy of 0 means all the examples have the same class value
 - An entropy of 1 means that they are randomly distributed
- If the prior probabilities of the n class values are p_1, \dots, p_n then the entropy is

$$H(\langle p_1, \dots, p_n \rangle) = \sum_{n=1} -p_i \log_2 p_i$$

Entropy and Huffmann Coding

- Entropy is the number of bits per symbol achieved by a (block) Huffman Coding scheme.
- Suppose we want to encode, into a bit string a long message composed of the two letters A and B , which occur with equal frequency. This can be done efficiently by assigning $A = 0, B = 1$. In other words, **one bit** (binary digit) is needed to encode each letter.
- Example 1: $H(\langle 0.5, 0.5 \rangle) = 1$ bit

Entropy and Huffmann Coding

- Suppose we need to encode a message consisting of the letters A, B and C, and that B and C occur equally often but A occurs twice as often as the other two letters.
- In this case, the most efficient code would be

$$A=0, B=10, C=11.$$

- The average number of bits needed to encode each letter is 1.5 .
- Example 2: $H(\langle 0.5, 0.25, 0.25 \rangle) = 1.5 \text{ bit}$

Entropy and Huffmann Coding

Example 3:

Consider a code to distinguish elements of {a, b, c, d} with

$$P(a) = \frac{1}{2}, P(b) = \frac{1}{4}, P(c) = \frac{1}{8}, P(d) = \frac{1}{8}$$

Consider the code:

a 0	b 10	c 110	d 111
-----	------	-------	-------

This code uses 1 to 3 bits. On average, it uses

$$P(a) \times 1 + P(b) \times 2 + P(c) \times 3 + P(d) \times 3 = \frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \frac{3}{8} = 1\frac{3}{4} \text{ bits}$$

Entropy and Huffmann Coding

Information theory

- A **bit** is a binary digit.
- 1 bit can distinguish 2 items
- k bits can distinguish 2^k items
- n items can be distinguished using $\log_2 n$ bits

Entropy and Huffman Coding

If the letters occur in some other proportion, we would need to “block” them together in order to encode them efficiently. But, **the average number of bits required** by the most efficient coding scheme is given by

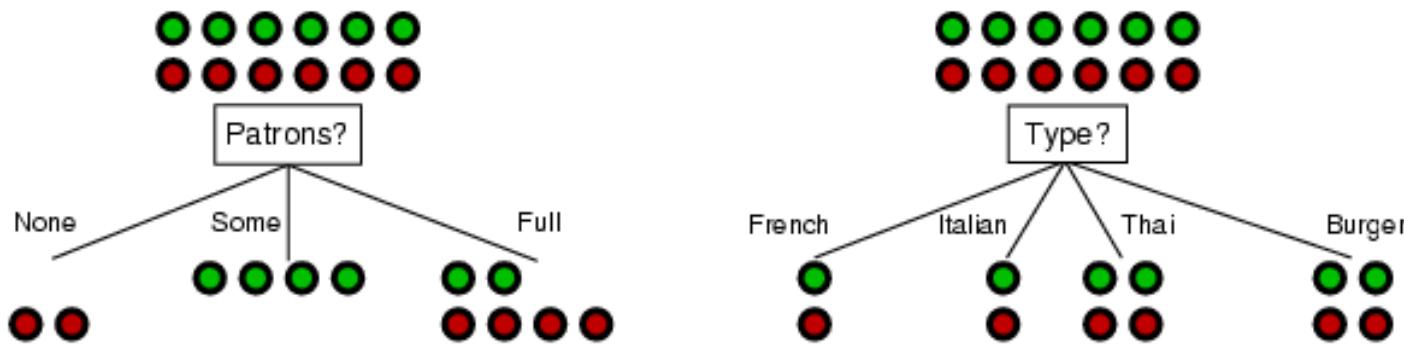
$$H(\langle p_1, \dots, p_n \rangle) = \sum_{n=1} -p_i \log_2 p_i$$

Entropy

- Suppose we have p positive and n negative examples in a node.
- $H\left(\left\langle \frac{p}{p+n}, \frac{n}{p+n} \right\rangle\right)$ bits needed to encode a new example.
 - e.g. for 12 restaurant examples, $p = n = 6$ so we need 1 bit.
- An attribute splits the examples E into subsets E_i , each of which (we hope) needs less information to complete the classification.
- Let E_i have p_i positive and n_i negative examples
 $H\left(\left\langle \frac{p_i}{p_i+n_i}, \frac{n_i}{p_i+n_i} \right\rangle\right)$ bits needed to encode an example
expected number of bits per example over all branches is $\sum_i \frac{p_i + n_i}{p + n} H\left(\left\langle \frac{p_i}{p_i+n_i}, \frac{n_i}{p_i+n_i} \right\rangle\right)$

Probability of going down branch i
- For **Patrons**, this is 0.459 bits, for **Type** this is (still) 1 bit → split on **Patrons**

Choosing and Attribute



$$\text{For Patrons, Entropy} = \frac{1}{6}(0) + \frac{1}{3}(0) + \frac{1}{2} \left[-\frac{1}{3} \log\left(\frac{1}{3}\right) - \frac{2}{3} \log\left(\frac{2}{3}\right) \right]$$

$$= 0 + 0 + \frac{1}{2} \left[\frac{1}{3}(1.585) + \frac{2}{3}(0.585) \right] = 0.459$$

$$\text{For Type, Entropy} = \frac{1}{6}(1) + \frac{1}{6}(1) + \frac{1}{3}(1) + \frac{1}{3}(1) = 1$$

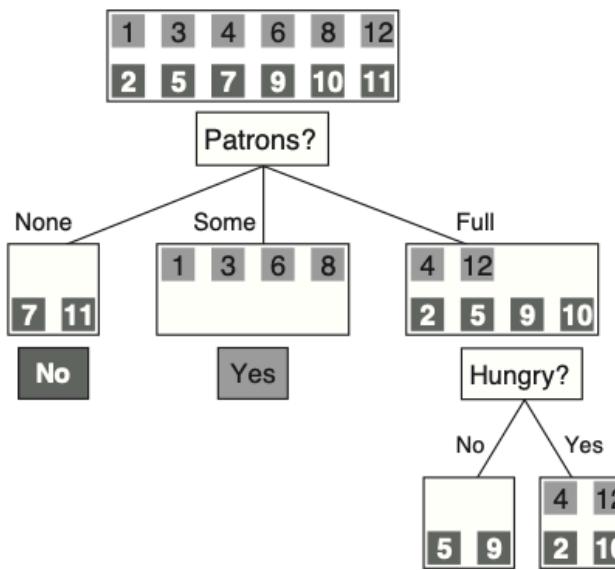
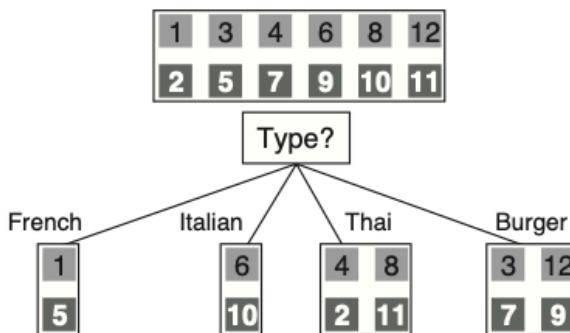
Entropy

- If the prior probabilities of n attribute values are p_1, \dots, p_n , then the entropy of the distribution is

$$H(\langle p_1, \dots, p_n \rangle) = \sum_{n=1} -p_i \log_2 p_i$$

- Entropy is a measure of “randomness” (lack of uniformity)
 - Related to prior distribution of some random variable
 - Higher entropy means more randomness
 - “Information” (about distribution) reduces entropy
- Split based on information gain
 - Loss of entropy based on “communicating” value of attribute
 - Related to Shannon’s information theory
 - Measure information gain in bits

Choosing Next Attribute

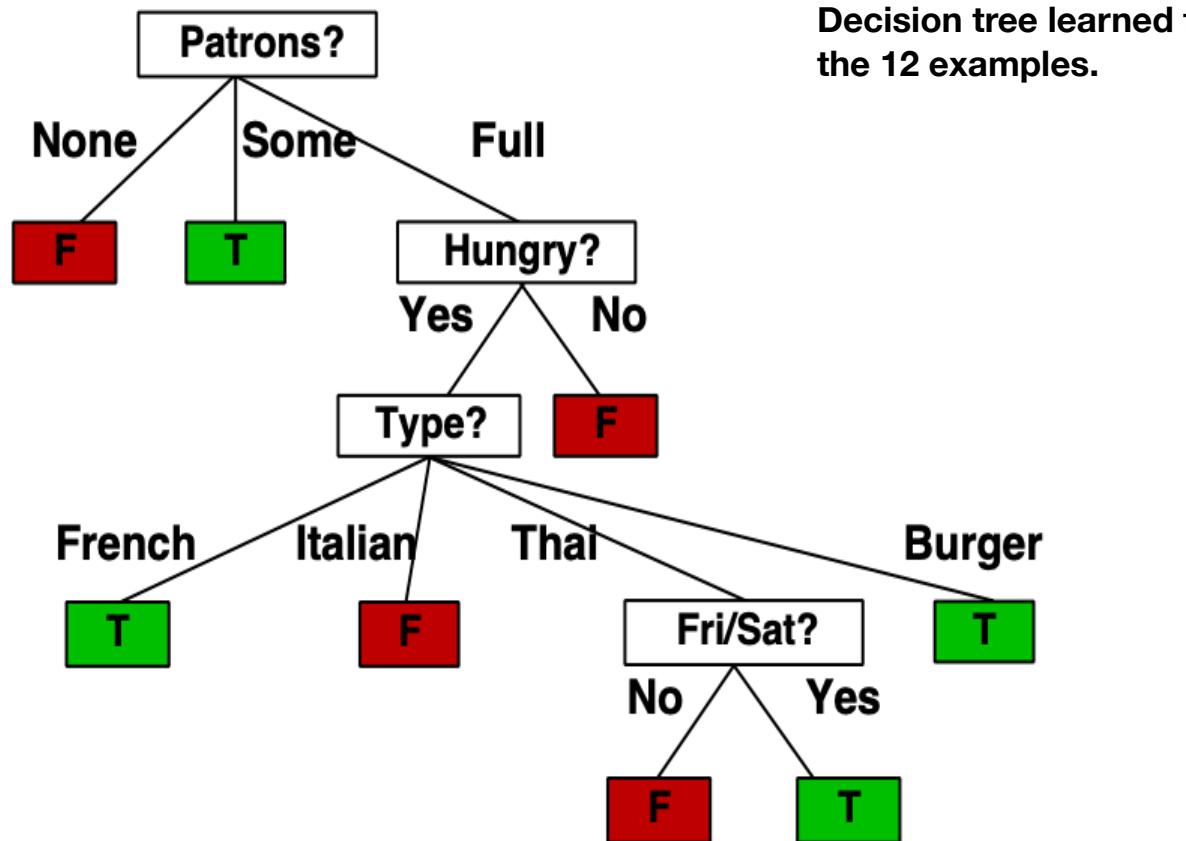


$$\sum_i \frac{p_i + n_i}{p + n} H\left(\left\langle \frac{p_i}{p_i + n}, \frac{n_i}{p_i + n_i} \right\rangle\right)$$

After splitting on Patrons,
split on Hungry

	Alt	Bar	F/S	Hun	Pat	Price	Pain	Res	Type	Est	Wait?
X ₁	T	F	F	T	Some	\$\$\$	F	T	French	0-10	T
X ₂	T	F	F	T	Full	\$	F	F	Thai	30-60	F
X ₃	F	T	F	F	Some	\$	F	F	Burger	0-10	T
X ₄	T	F	T	T	Full	\$	F	F	Thai	10-30	T
X ₅	T	F	T	F	Full	\$\$\$	F	T	French	> 60	F
X ₆	F	T	F	T	Some	\$\$	T	T	Italian	0-10	T
X ₇	F	T	F	F	None	\$	T	F	Burger	0-10	F
X ₈	F	F	F	T	Some	\$\$	T	T	Thai	0-10	T
X ₉	F	T	T	F	Full	\$	T	F	Burger	> 60	F
X ₁₀	T	T	T	T	Full	\$\$\$	F	T	Italian	10-30	F
X ₁₁	F	F	F	F	None	\$	F	F	Thai	0-10	F
X ₁₂	T	T	T	T	Full	\$	F	F	Burger	30-60	T

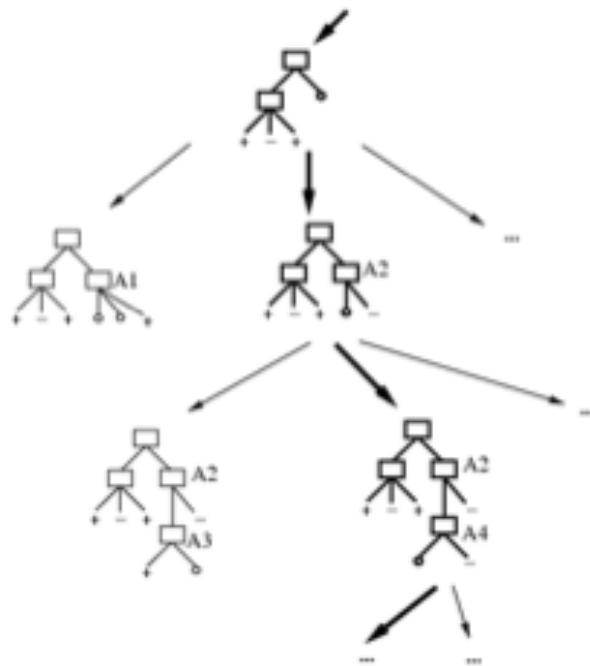
Induced Tree



Decision Trees

- Decision tree learning is a method for **approximating** discrete value target functions, in which the learned function is represented by a decision tree.
- Decision trees can also be represented by if-then-else rule
- Decision tree learning is one of the most widely used approach for inductive inference

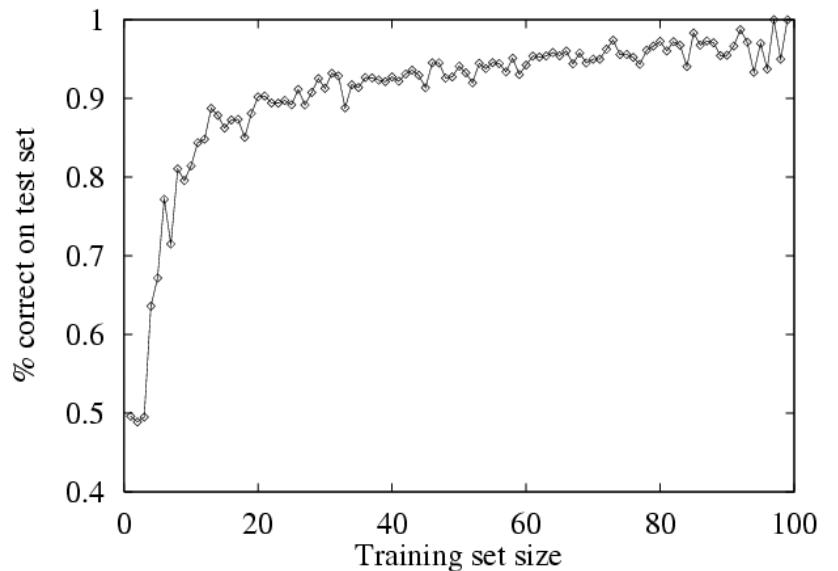
Which Tree Should We Output?



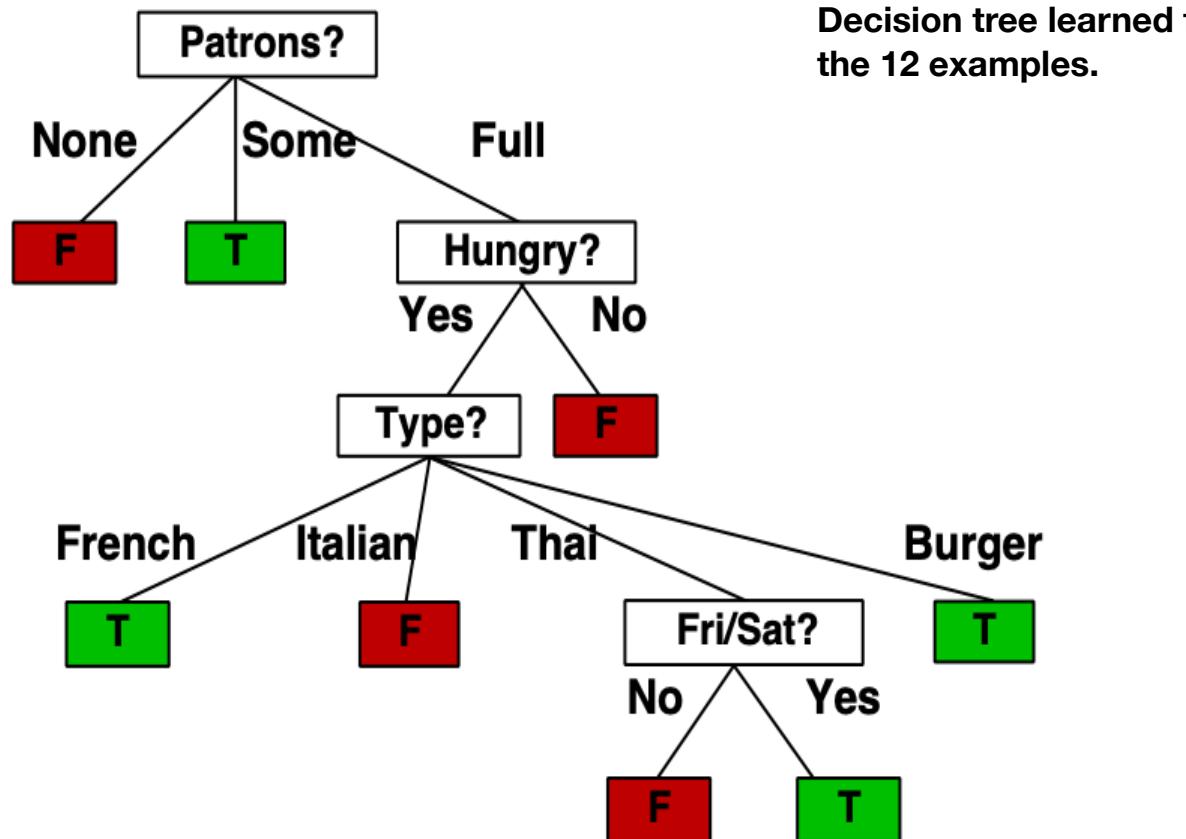
- Decision tree learning performs heuristic search through space of decision trees
- Stops at smallest acceptable tree
- Occam's razor: prefer simplest hypothesis that fits data

Performance measurement

- How do we know that $h \approx f$?
- 1. Use theorems of computational/statistical learning theory
- 2. Try h on a new **test set** of examples
 - (use **same distribution** over example space as training set)
- **Learning curve** = % correct on test set as a function of training set size

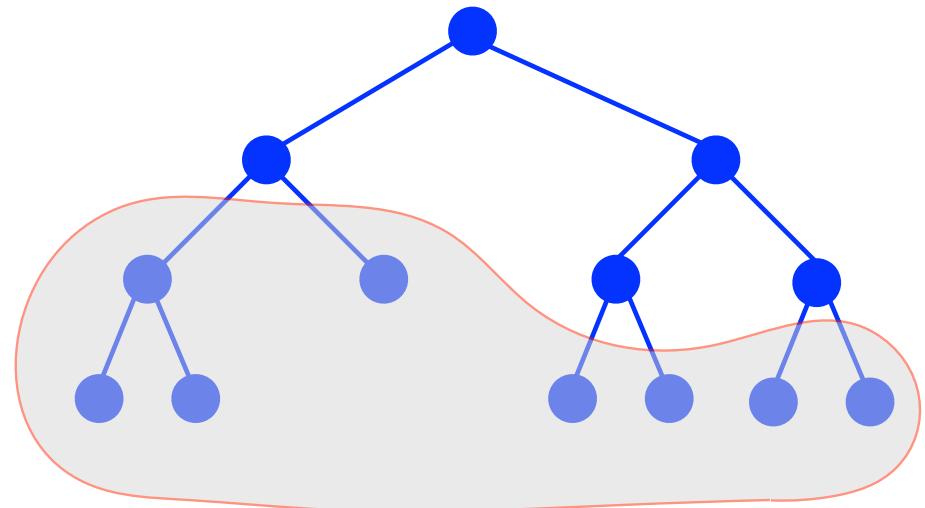


Induced Tree



Overfitting

- What if training data are noisy?
 - Misclassified examples
 - Incorrect measurements
- Making decision tree classify every example (including noise) could make it less accurate
 - Tries to classify bad examples
- ➡ Misclassifies correct examples in test set
- This is called *overfitting*
- Can improve accuracy by pruning branches created by trying to classify noise.



Tree Pruning To Minimise Error

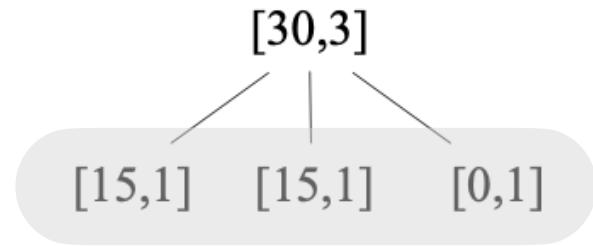
- Maximise expected accuracy, minimise expected probability of error
- For given tree, estimate its expected error
- Also estimate errors for variously pruned tree
- Choose tree with minimal estimated error

How to estimate error?

- One possibility: use “pruning set”
- Another possibility: estimate error probability from data in tree

Tree Pruning

- The top node of this tree has
 - 30 +ve examples
 - 3 -ve examples
- It is split 3 ways.
- If we decide to prune it, the top node becomes a leaf node and the decision value is the majority class, i.e. +ve.
- Only prune if the expected error is less than the expected error with the children



Laplace Error and Pruning

When a node becomes a leaf, all items will be assigned to the majority class at that node. We can estimate the error rate on the (unseen) test items using the [Laplace error](#):

$$E = 1 - \frac{n + 1}{N + k}$$

N = total number of training examples

n = number of training examples in the majority class

k = number of classes

If the average Laplace error of the children exceeds that of the parent node, we prune off the children.

How do we get the Laplace Error?

- Suppose a node has 99 +ve and 1 -ve: $\frac{n}{N} = \frac{99}{100}$ is the probability of the majority class.
- If we decide to prune, the expected error will be $1 - \frac{99}{100} = 0.01$
- This is a good estimate if N is large, but small better to rely on prior probability of class, i.e. $\frac{1}{k}$
- So the [Laplace error](#) adds a bias for small N . When N is large, correction is irrelevant

$$E = 1 - \frac{n+1}{N+k}$$

N = total number of training examples

n = number of training examples in the majority class

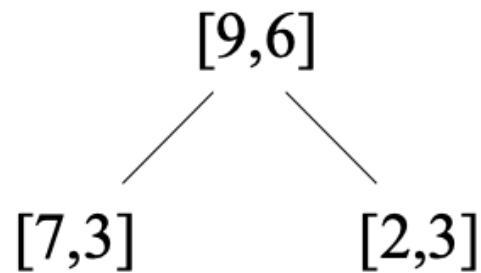
k = number of classes

Minimal Error Pruning

Should the children of this node be pruned or not?

Left child has class frequencies [7, 3]

$$E = 1 - \frac{n+1}{N+k} = 1 - \frac{7+1}{10+2} = 0.333$$



Right child has $E = 0.429$

Parent node has $E = 0.412$

Average for left and right child is:

$$E = \frac{10}{15} \times 0.333 + \frac{5}{15} \times 0.429 = 0.365$$

Since $0.365 < 0.412$, children should NOT be pruned

Minimal Error Pruning

Should the children of this node be pruned or not?

Left child has class frequencies [3, 2]

$$E = 1 - \frac{n+1}{N+k} = 1 - \frac{3+1}{5+2} = 0.429$$

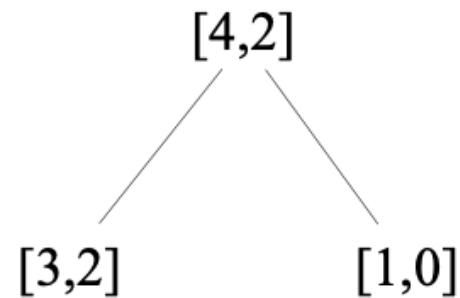
Right child has $E = 0.333$

Parent node has $E = 0.375$

Average for left and right child is:

$$E = \frac{5}{6} \times 0.429 + \frac{1}{6} \times 0.333 = 0.413$$

Since $0.413 > 0.375$, children should be pruned

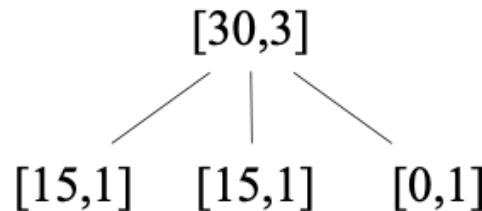


Minimal Error Pruning

Should the children of this node be pruned or not?

Left child has class frequencies [15, 1]

$$E = 1 - \frac{n+1}{N+k} = 1 - \frac{15+1}{16+2} = 0.111$$



Right child has $E = 0.333$

$$\text{Parent node has } E = \frac{4}{35} = 0.114$$

Average for left and right child is:

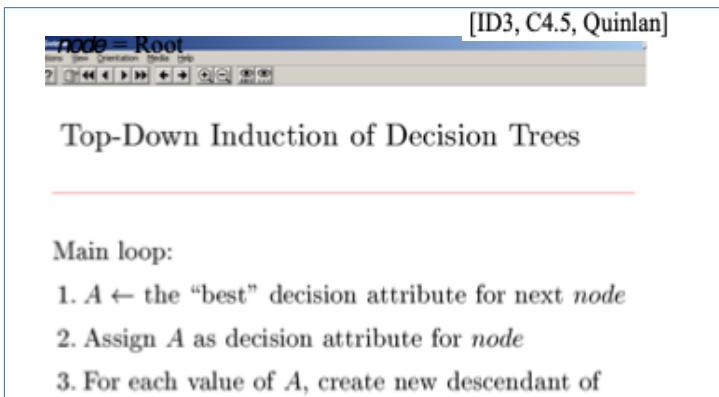
$$E = \frac{16}{33} \times 0.111 + \frac{16}{33} \times 0.111 + \frac{1}{33} \times 0.333 = 0.118$$

Since $0.118 > 0.114$, children should be pruned

Decision Tree Learning Algorithms

- ID3 Algorithm (Quinlan 1986) and it's successors C4.5 and C5.0
- *Employs a top-down induction*

[ID3, C4.5, Quinlan]



Top-Down Induction of Decision Trees

Main loop:

1. $A \leftarrow$ the “best” decision attribute for next *node*
2. Assign A as decision attribute for *node*
3. For each value of A , create new descendant of

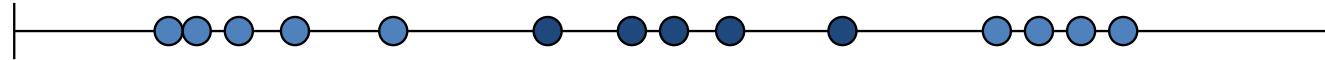


<http://www.rulequest.com/>

- Greedy search the space of possible decision trees.
- The algorithm never backtracks to reconsider earlier choices.

Numerical Attributes

- ID3 algorithm is designed for attributes that have discrete values.
 - How can we handle attributes with continuous numerical values?
- ➡ Must discretise values.



Problems Suitable for Decision Trees

- Instances are represented by attribute-value pairs
- Instances are described by a fixed set of attributes (e.g., Temperature) and their values (e.g., Hot).
 - Easiest domains for decision tree learning is when each attribute takes on a small number of disjoint possible values (e.g., Hot, Mild, Cold).
 - Extensions allow handling real-valued attributes as well (e.g., representing temperature numerically).
- The target function has discrete output values
- The training data
 - The training data may contain errors
 - The training data may contain missing attribute values

Some TDIDT Systems

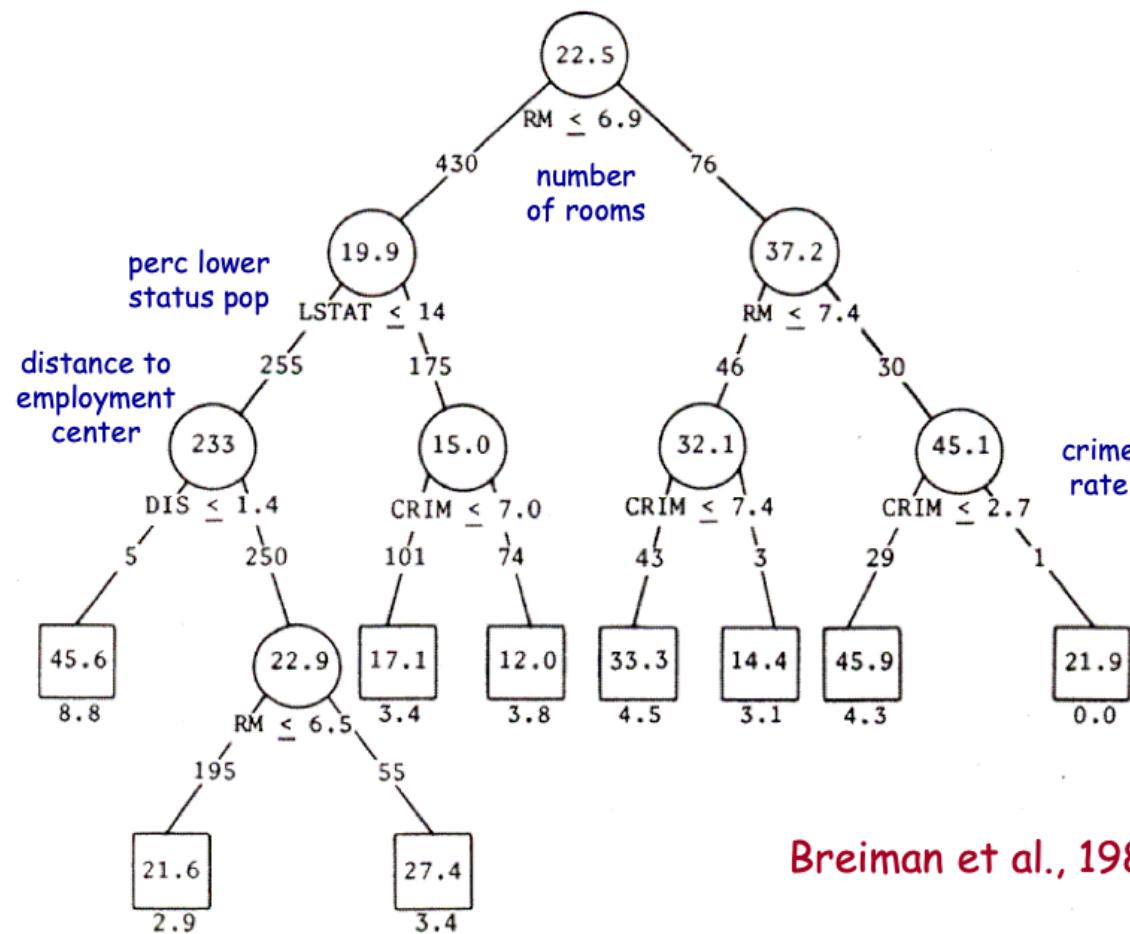
- ID3 (Quinlan 79)
- CART (Breiman et al. 84)
- Assistant (Cestnik et al. 87)
- C4.5 (Quinlan 93)
- C5 (Quinlan 97)
- Trees in WEKA (Witten, Frank 2000 - ...)
- scikit-learn

TDIDT - Top-Down Induction of Decision Trees

Inducing Regression Trees

- Like induction of decision trees
- Regression trees useful in continuous domains,
 - e.g. predict biomass of algae
- Decision trees: discrete class
- Regression trees: continuous, real-valued class

Example: Boston Housing Values



Breiman et al., 1984

Some Systems that Induce Regression Trees

- CART (Breiman et al. 1984)
- RETIS (Karalić 1992)
- M5 (Quinlan 1993)
- WEKA (Witten and Frank 2000-...)

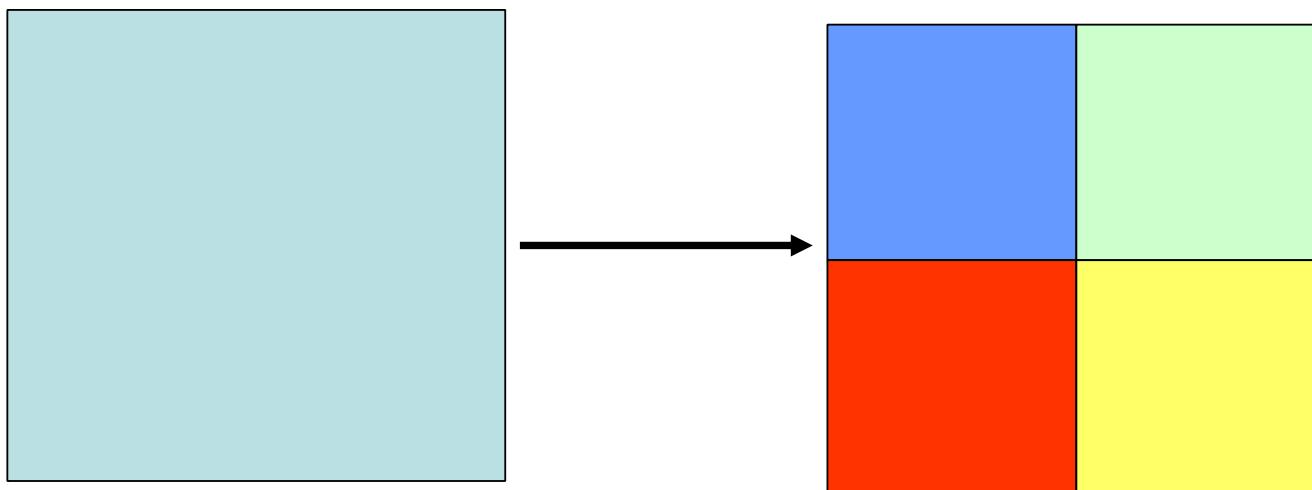
Training and Testing

- For classification problems, a classifier's performance is measured in terms of the *error rate*.
- The classifier predicts the class of each instance: if it is correct, that is counted as a *success*; if not, it is an *error*.
- The error rate is just the proportion of errors made over a whole set of instances, and it measures the overall performance of the classifier.

Training and Testing

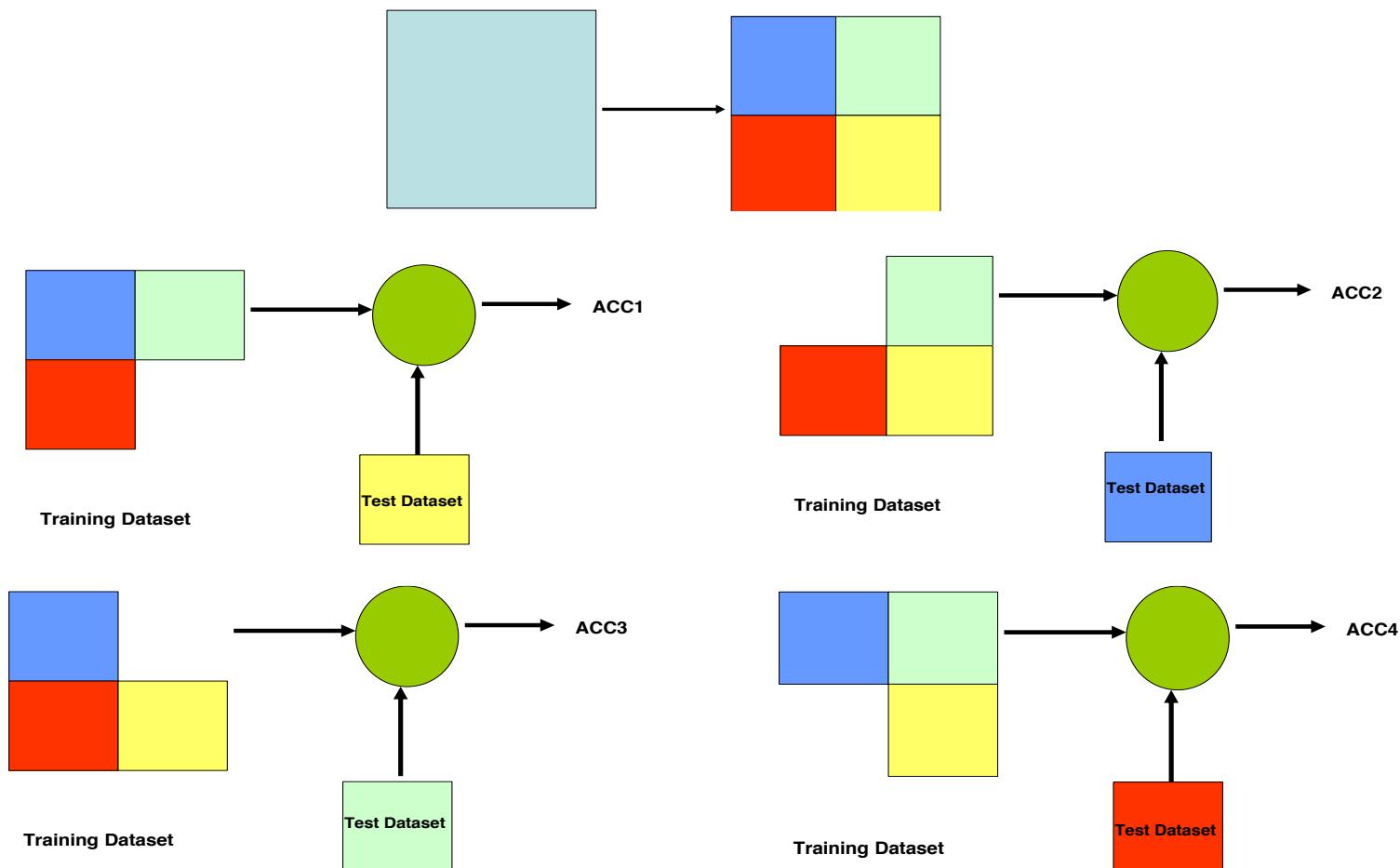
- Self-consistency test: when training and test dataset are the same
- Hold out strategy: reserve some examples for testing and use the rest for training (set part of that aside for validation, if required)
- K-fold Cross validation:
 - If don't have many examples
 - Partition dataset randomly into k equal-sized sets.
 - Train and test classifier k times using one set for testing and other $k - 1$ sets for training

4-Fold Cross-validation



$$\text{ACC} = (\text{ACC1} + \text{ACC2} + \text{ACC3} + \text{ACC4}) / 4$$

4-Fold Cross-validation



K-Fold Cross Validation

- Train multiple times, leaving out a disjoint subset of data each time for test.
- Average the test set accuracies.

Partition data into K disjoint subsets

for $k \in 1..K$:

$testData \leftarrow k_{th}$ subset

$h \leftarrow$ classifier trained on all data except for $testData$

$accuracy(k) =$ accuracy of h on $testData$

end

$FinalAccuracy =$ mean of the K recorded test set accuracies

Leave-One-Out Cross Validation

- This is just k-fold cross validation leaving out one example each iteration. Average the test set accuracies

Partition data into K disjoint subsets *each containing one example*

for $k \in 1..K$:

$testData \leftarrow k_{th}$ subset

$h \leftarrow$ classifier trained on all data except for $testData$

$accuracy(k) =$ accuracy of h on $testData$

end

$FinalAccuracy =$ mean of the K recorded test set accuracies

Summary

- Supervised Learning
 - Training set and test set
 - Try to predict target value based on input attributes
- Ockham's Razor
 - Tradeoff between simplicity and accuracy
- Decision Trees
 - Improve generalisation by building a smaller tree (using entropy)
 - Prune nodes based on Laplace error
 - Other ways to prune Decision Trees

References

- Poole & Mackworth, Artificial Intelligence: Foundations of Computational Agents, Chapter 7.
- Russell & Norvig, *Artificial Intelligence: a Modern Approach*, Chapter 18.1, 18.2, 18.3

Neural Networks

COMP3411/9814: Artificial Intelligence

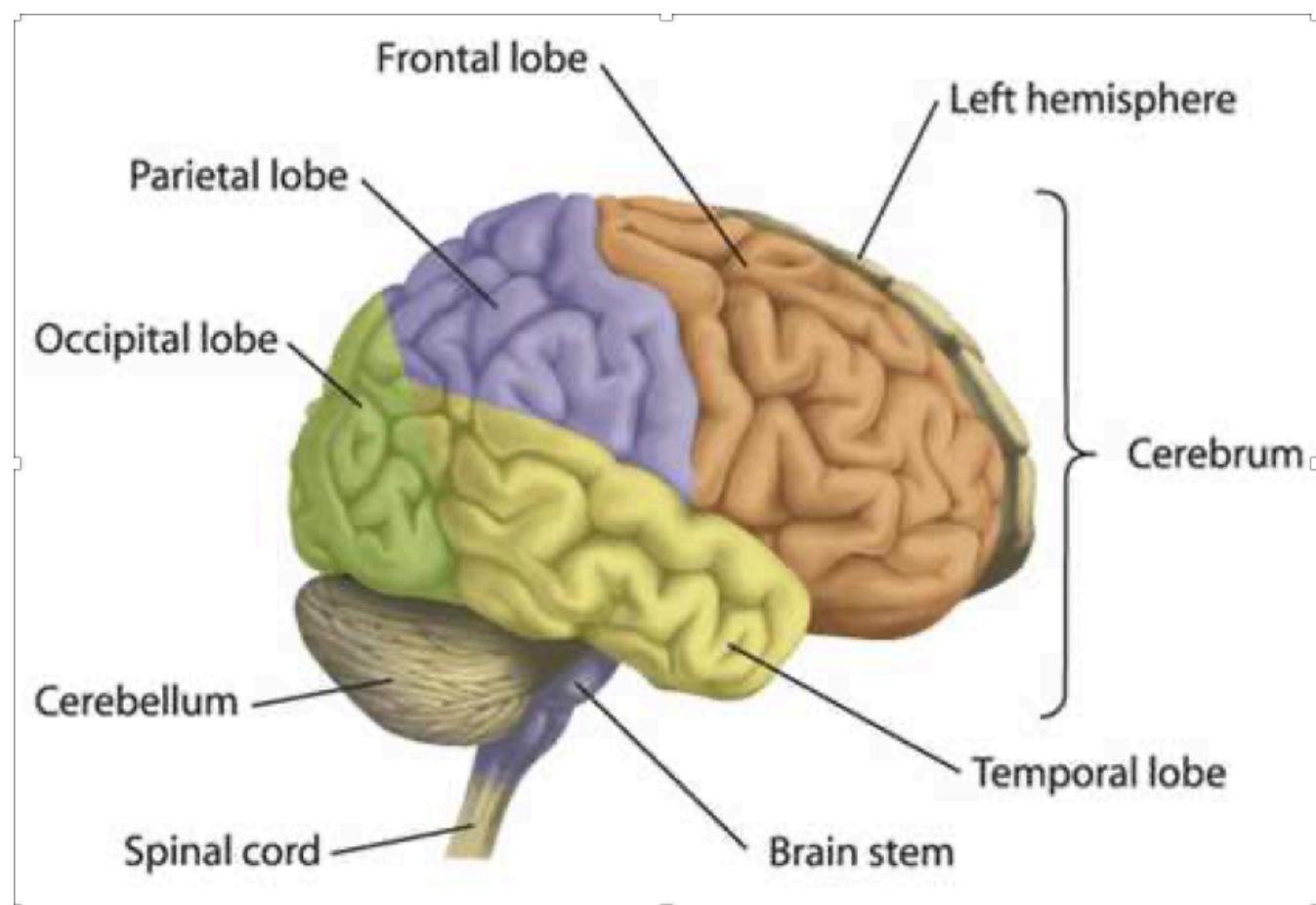
Lecture Overview

- Neurons – Biological and Artificial
- Perceptron Learning
- Linear Separability
- Multi-Layer Networks
- Back propagation
- Applications of neural networks

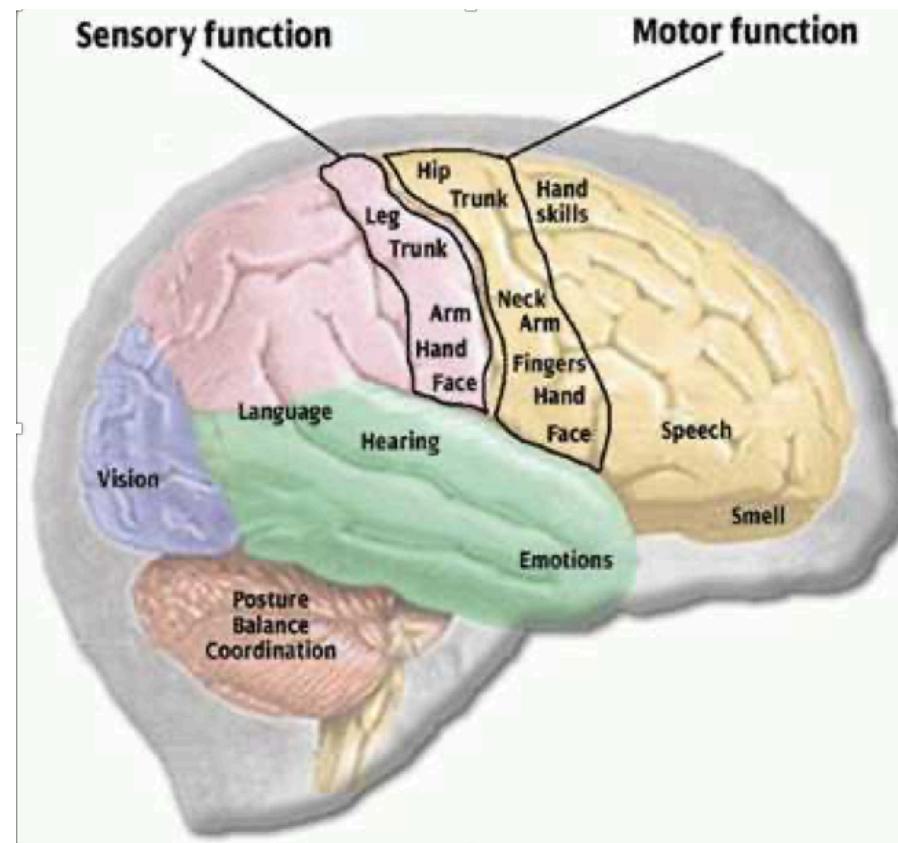
Sub-Symbolic Processing



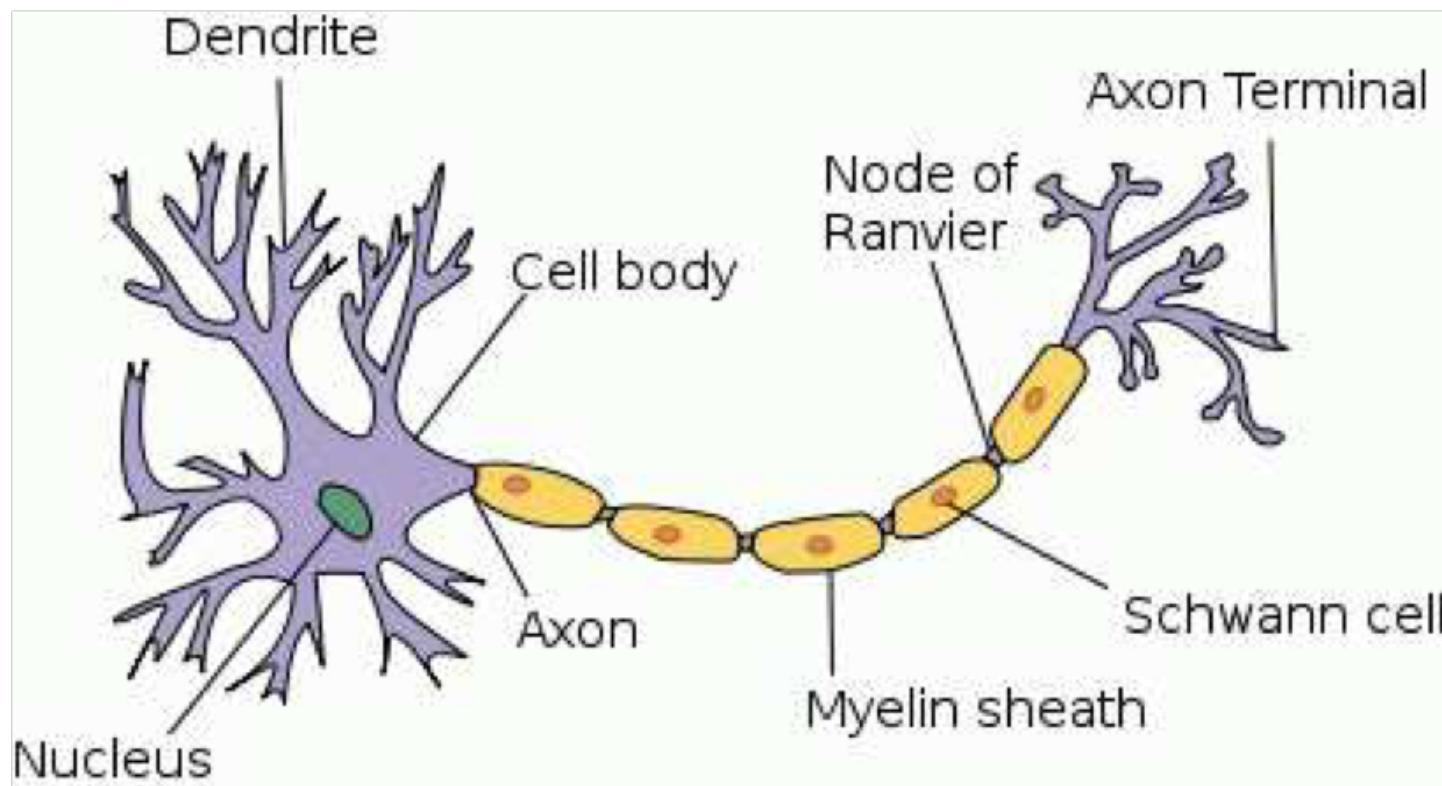
Brain Regions



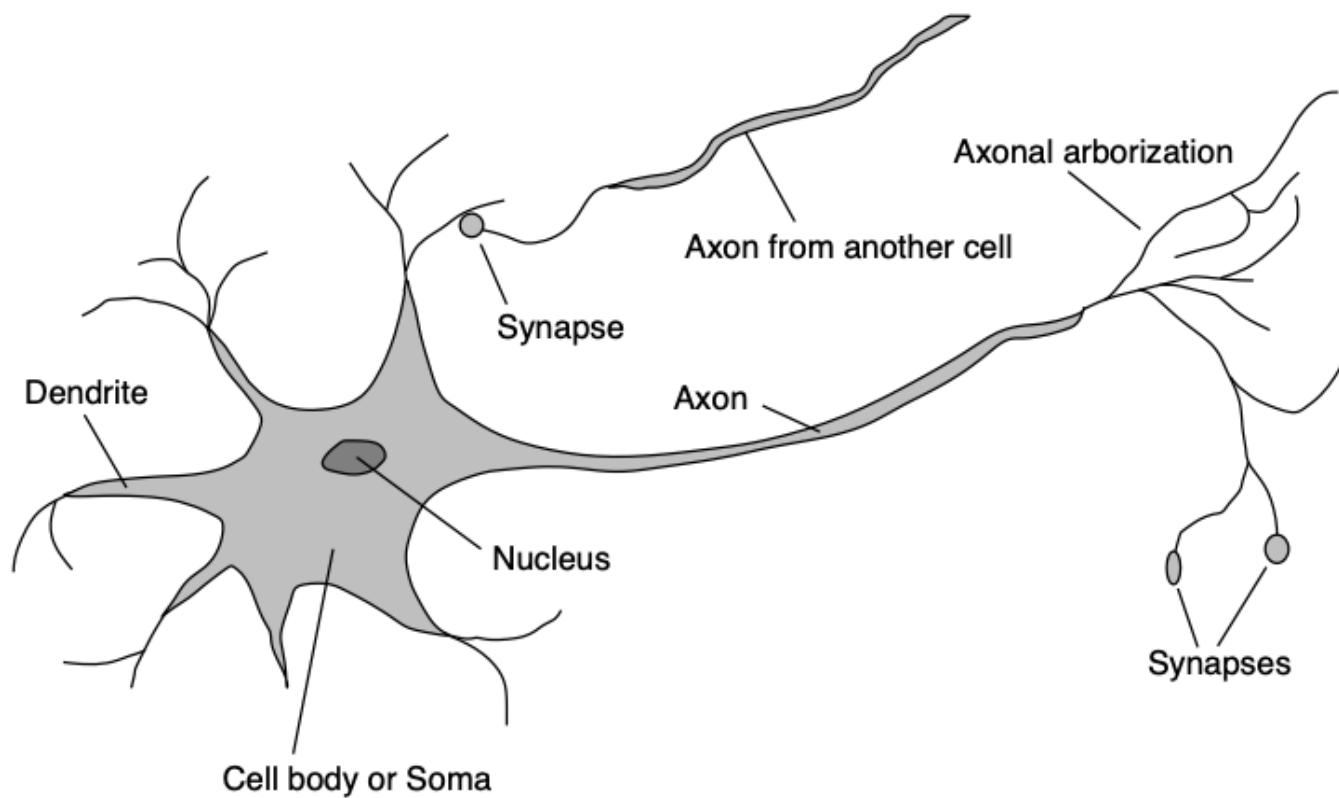
Brain Regions



Structure of a Typical Neuron



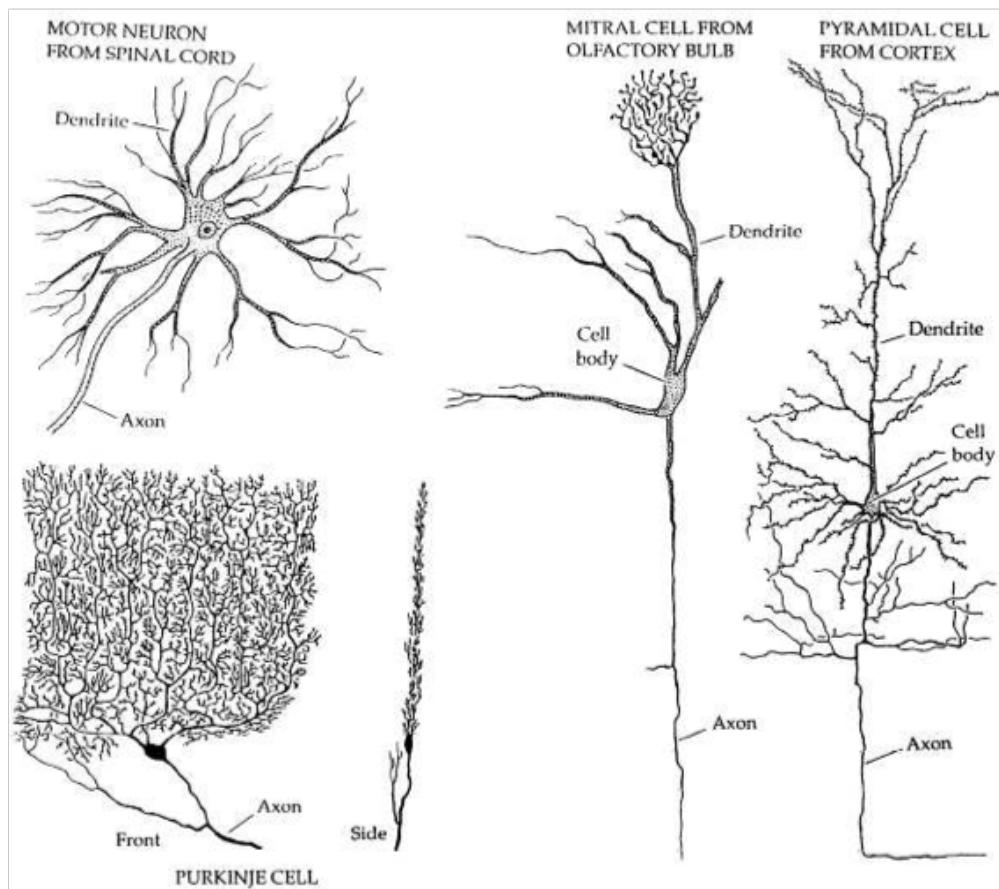
Brains



Biological Neurons

- The brain is made up of **neurons** (nerve cells) which have
 - a cell body (soma)
 - **dendrites** (inputs)
 - an **axon** (outputs)
 - **synapses** (connections between cells)
- Synapses can be **excitatory** or **inhibitory** and may change over time.
- When the inputs reach some threshold an **action potential** (electrical pulse) is sent along the axon to the outputs.

Variety of Neuron Types



The Big Picture

- Human brain has 100 billion neurons with an average of 10,000 synapses each
- Latency is about 3-6 milliseconds
- At most a few hundred “steps” in any mental computation, but **massively parallel**

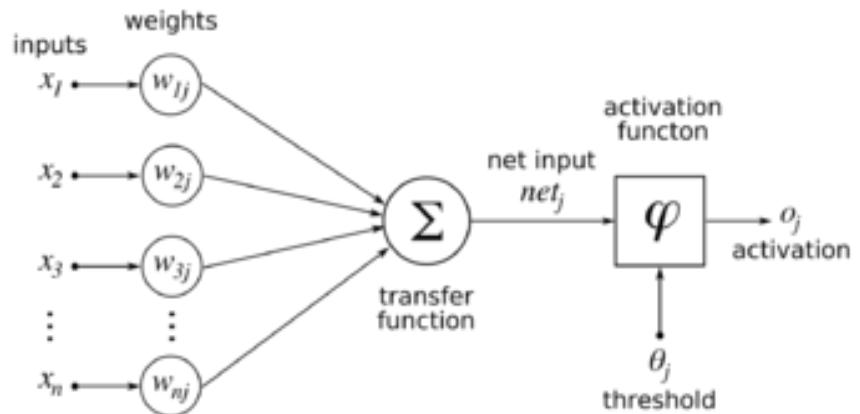
Artificial Neural Networks

- Information processing architecture loosely modelling the brain
- Consists of many interconnected processing units (**neurons**)
 - Work in parallel to accomplish a global task
- Generally used to model relationships between inputs and outputs or to find patterns in data

Artificial Neural Networks (ANN)

□ ANNs nodes have

- inputs edges with some **weights**
- outputs edges with **weights**
- **activation level** (function of inputs)



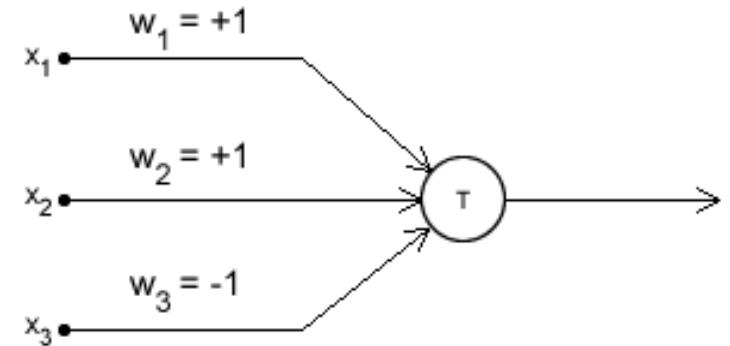
- Weights can be positive or negative and may change over time (learning).
- The **input function** is the weighted sum of the activation levels of inputs.
- The activation level is a non-linear **transfer function** g of this input:

$$\text{activation}_i = g(s_i) = g\left(\sum_j w_{ij} x_j \right)$$

Some nodes are inputs (sensing), some are outputs (action)

First artificial neurons: McCulloch-Pitts (1943)

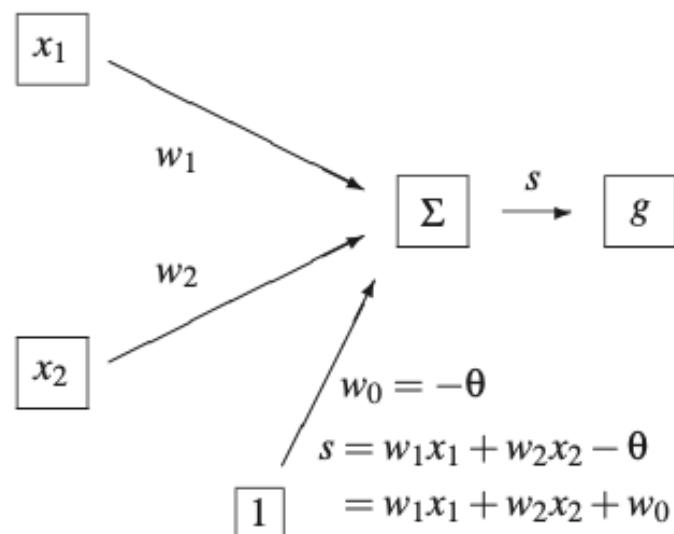
- McCulloch-Pitts model:
 - Inputs either 0 or 1.
 - Output 0 or 1.
 - Input can be either excitatory or inhibitory.
- Summing inputs
 - If input is 1, and is excitatory, add 1 to sum.
 - If input is 1, and is inhibitory, subtract 1 from sum.
- Threshold,
 - if sum < threshold, T, output 0.
 - Otherwise, output 1.



$$\text{sum} = x_1 \cdot w_1 + x_2 \cdot w_2 + x_3 \cdot w_3 + \dots$$

if $\text{sum} < T$ **then** output is 0
else output is 1.

McCulloch & Pitts Model of a Single Neuron



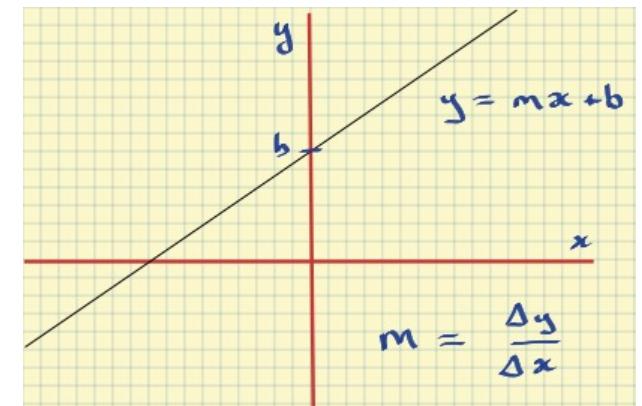
x_1, x_2 are inputs

w_1, w_2 are synaptic weights

θ is a threshold

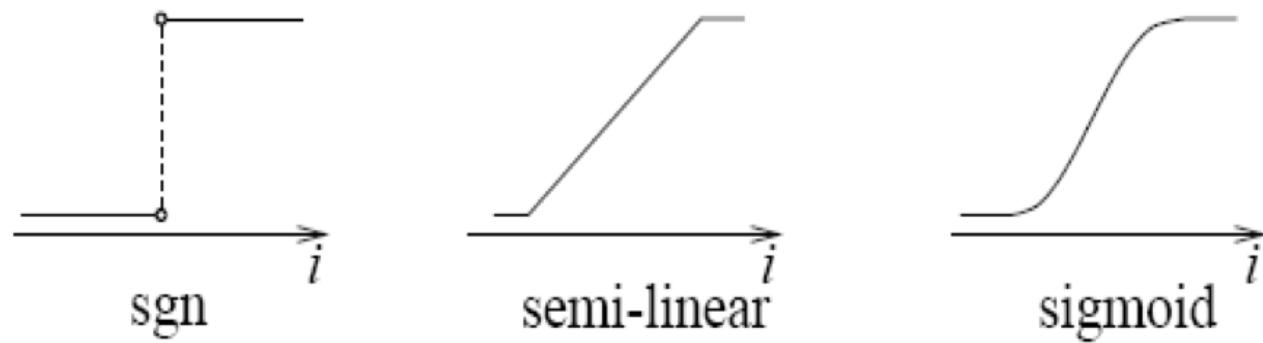
w_0 is a **bias** weight

g is transfer function



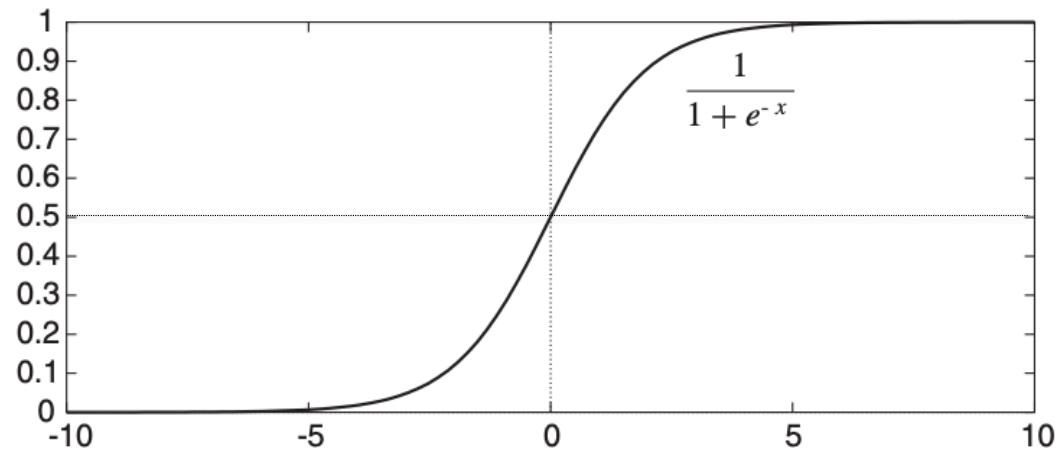
Activation Functions

Function $g(s)$ takes the weighted sum of inputs and produces output for node, given some threshold.



$$g(s) = \begin{cases} 1 & \text{if } s \geq 0 \\ 0 & \text{if } s < 0 \end{cases}$$

The sigmoid or logistic activation function

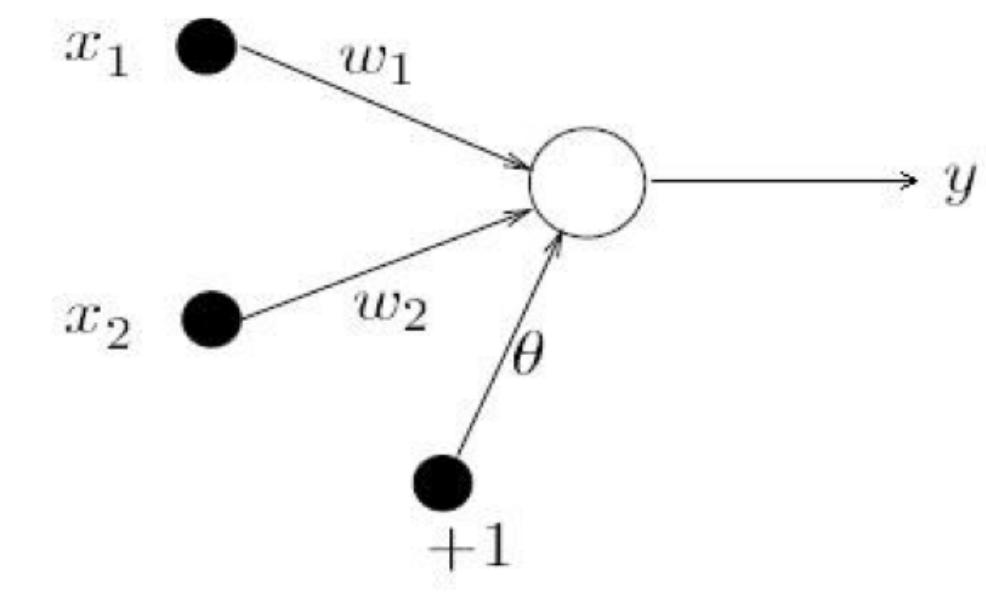


$$f(x) = \frac{1}{1 + e^{-x}}$$

Derivative $f'(x) = f(x)(1 - f(x))$ is the slope of the function

Simple Perceptron

The perceptron is a single layer feed-forward neural network.

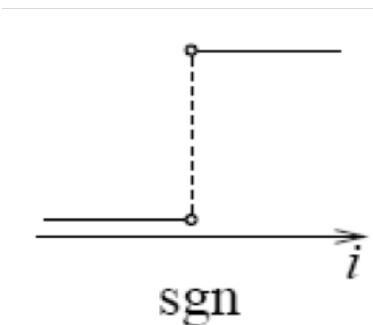


Simple Perceptron

Simplest output function

$$y = \text{sgn} \left(\sum_{i=1}^2 w_i x_i + \theta \right)$$

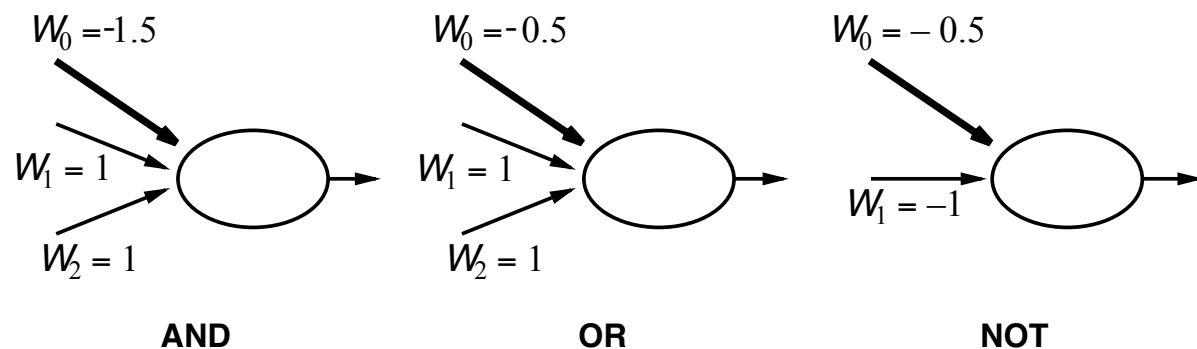
$$\text{sgn}(s) = \begin{cases} 1 & \text{if } s > 0 \\ -1 & \text{otherwise.} \end{cases}$$



Used to classify patterns said to be linearly separable

Implementing logical functions

McCulloch and Pitts - every Boolean function can be implemented:



Linear Separability

Examples:

AND $w_1 = w_2 = 1.0, \quad w_0 = -1.5$

OR $w_1 = w_2 = 1.0, \quad w_0 = -0.5$

NOR $w_1 = w_2 = -1.0, \quad w_0 = 0.5$

Can we train a perceptron net to learn a new function?
Yes, as long as it is linearly separable

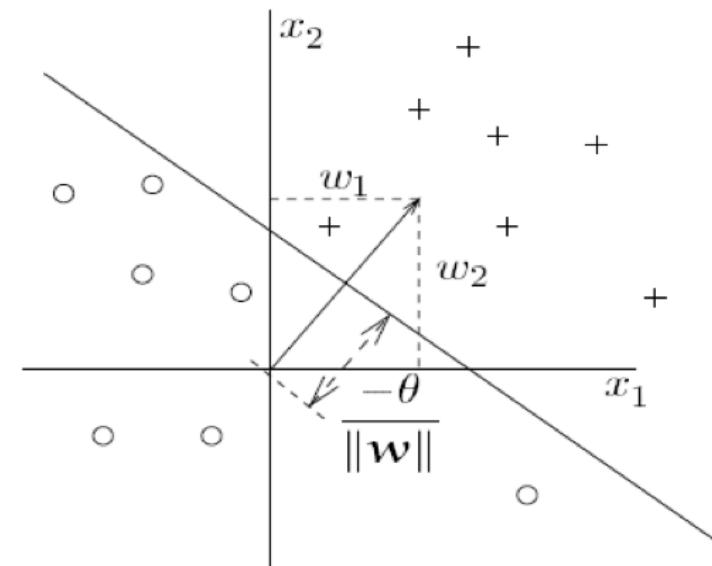
Linear Separability

The **bias** is proportional to the offset of the plane from the origin

The weights determine the slope of the line

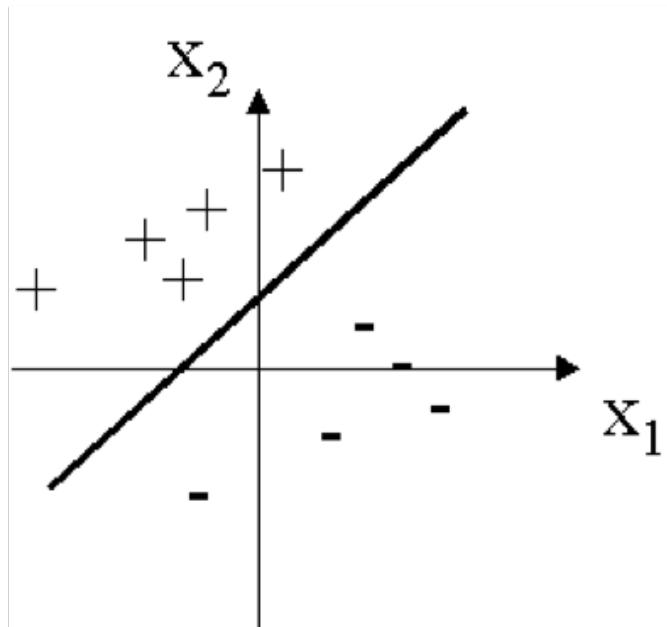
The weight vector is perpendicular to the plane

$$x_2 = -\frac{w_1}{w_2}x_1 - \frac{\theta}{w_2}$$



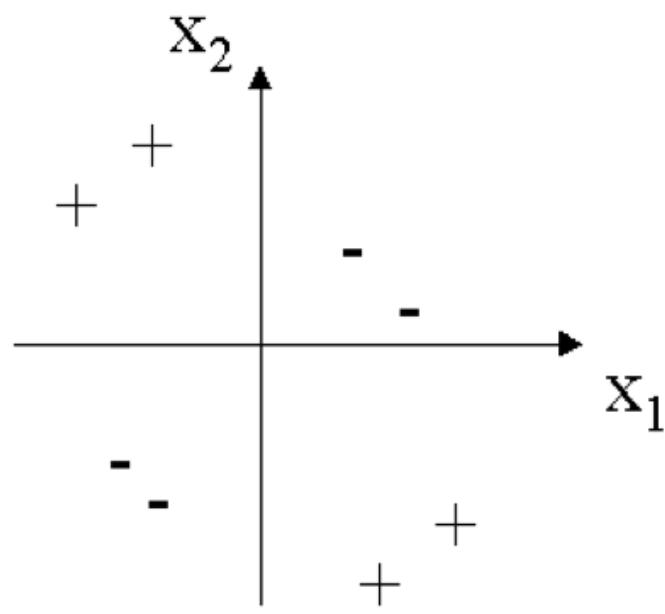
Linear Separability

What kind of functions can a perceptron compute?



Linearly Separable

$$w_1x_1 + w_2x_2 + \theta = 0$$



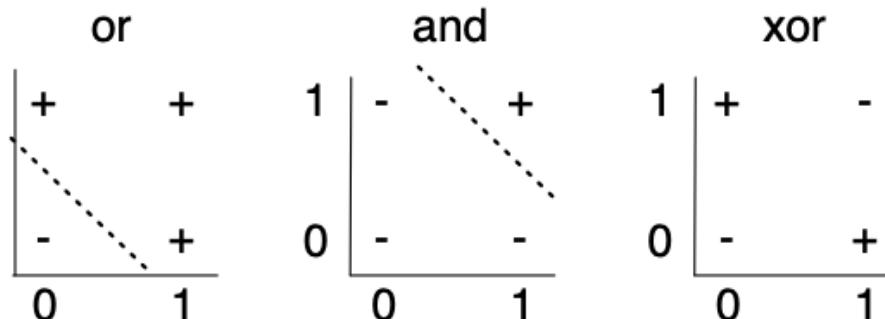
Not Linearly Separable

Linearly Separable

- **Linearly separable** if there is a hyperplane where classification is true on one side of hyperplane and false on other side
- For the sigmoid function, when the hyperplane is:

$$x_1 \cdot w_1 + \dots + x_n \cdot w_n = 0$$

separates the predictions > 0.5 and < 0.5 .

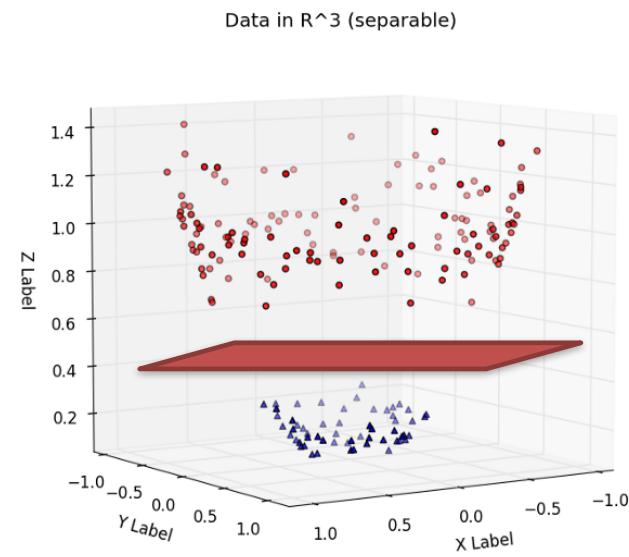
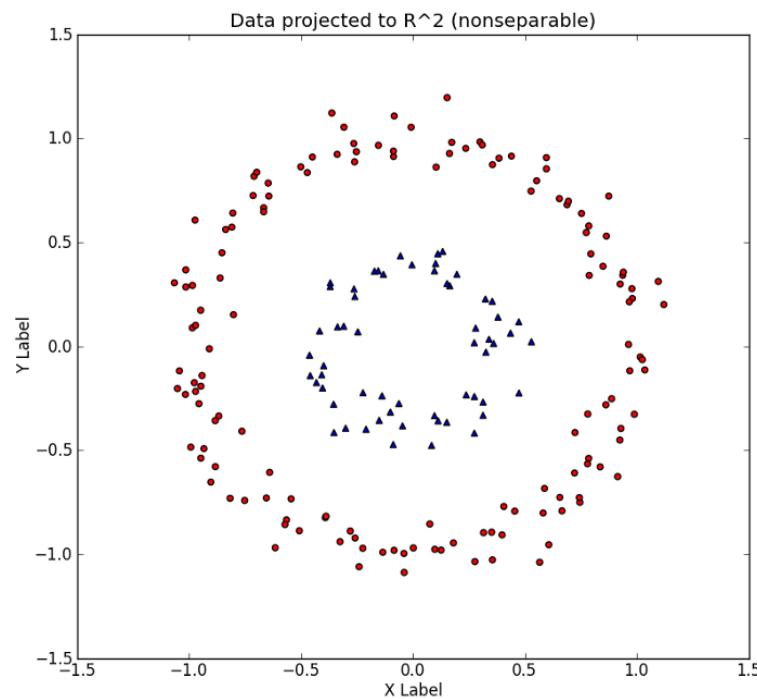


Variants in Linear Separators

- Which linear separator to use can result in various algorithms:
 - Perceptron
 - Logistic Regression
 - Support Vector Machines (SVMs) ...

Kernel Trick

- Project points onto an higher dimensional space
- Becomes linearly separable



Perceptron Learning Algorithm

- Want to train perceptron to classify inputs correctly
- Compare output of network with correct output and adjust the **weights** and **bias** to minimise the error
- So learning is parameter optimisation
- Can only handle linearly separable sets

Perceptron Learning Algorithm

- Training set: set of input vectors to train perceptron.
- During training, w_i and θ (bias) are modified
 - ▶ for convenience, let $w_0 = \theta$ and $x_0 = 1$
- η , is the *learning rate*, a small positive number
 - ▶ small steps lessen possibility of destroying correct classifications
- Initialise w_i to some values

Perceptron Learning Rule

- Repeat for each training example
 - Adjust the weight, w_i , for each input, x_i .

$$w_i \leftarrow w_i + \eta(d - y) \cdot x_i$$

$\eta > 0$ is the learning rate

d is the desired output

y is the actual output

- If output correct, no change
- If $d=1$ but $y=0$, w_i is increased when x_i is positive and decreased when x_i is negative (want to increase $\mathbf{w} \bullet \mathbf{x}$)
- If $d=0$ but $y=1$, w_i is decreased when x_i is positive and increased when x_i is negative (want to decrease $\mathbf{w} \bullet \mathbf{x}$)

Perceptron Convergence Theorem

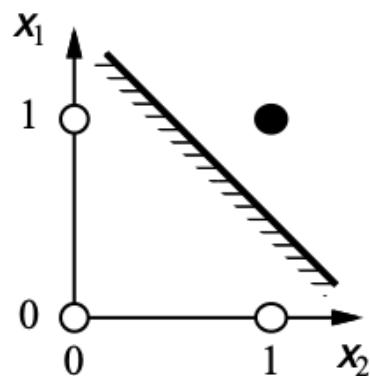
For any data set that is linearly separable, perceptron learning rule is guaranteed to find a solution in a finite number of iterations.

Historical Context

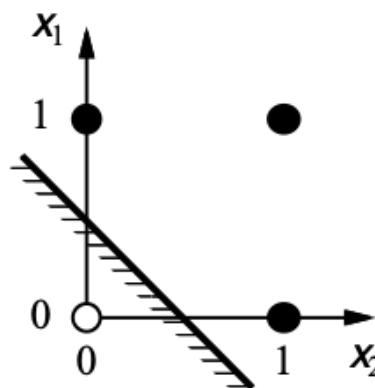
- In 1969, Minsky and Papert published book highlighting limitations of perceptrons
 - Funding agencies redirected funding away from neural network research,
 - preferring instead logic-based methods such as expert systems.
- Known since 1960's that any logical function could be implemented in a 2-layer neural network with step function activations.
- Problem was how to learn weights of multi-layer neural network from training examples
- Solution found in 1976 by Paul Werbos
 - not widely known until rediscovered in 1986 by Rumelhart, Hinton and Williams.

Limitations of Perceptrons

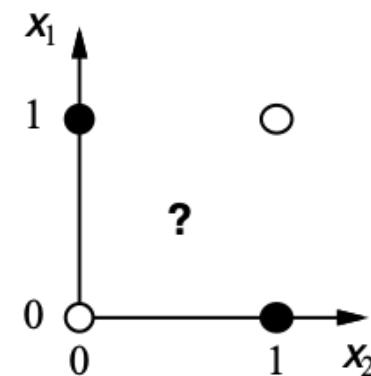
Problem: many useful functions are not linearly separable (e.g. XOR)



(a) x_1 **and** x_2



(b) x_1 **or** x_2



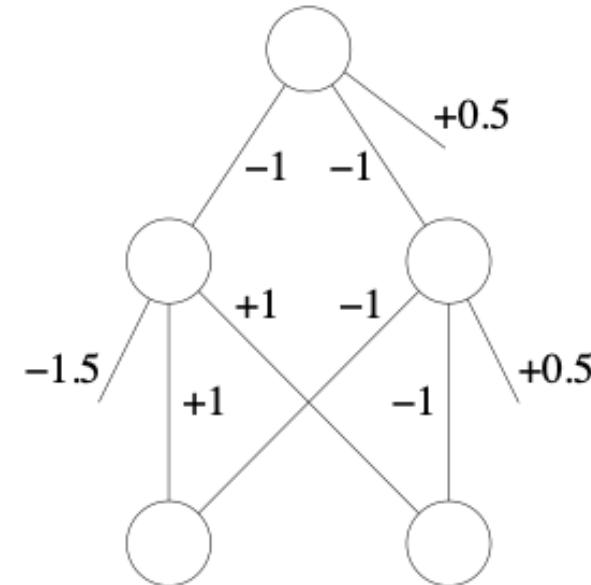
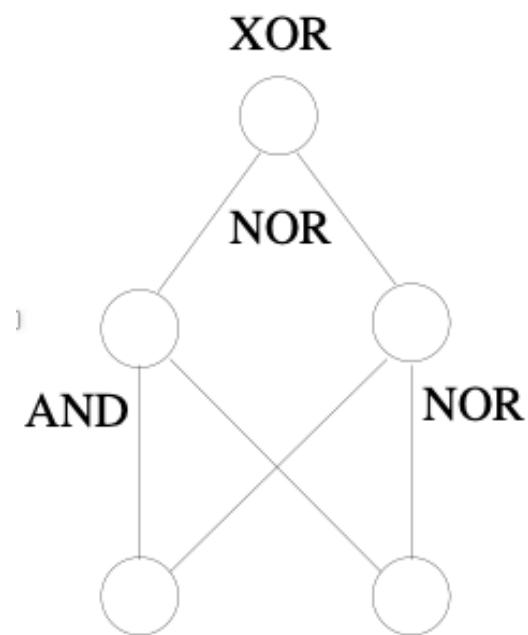
(c) x_1 **xor** x_2

Possible solution:

x_1 XOR x_2 can be written as: $(x_1 \text{ AND } x_2) \text{ NOR } (x_1 \text{ NOR } x_2)$

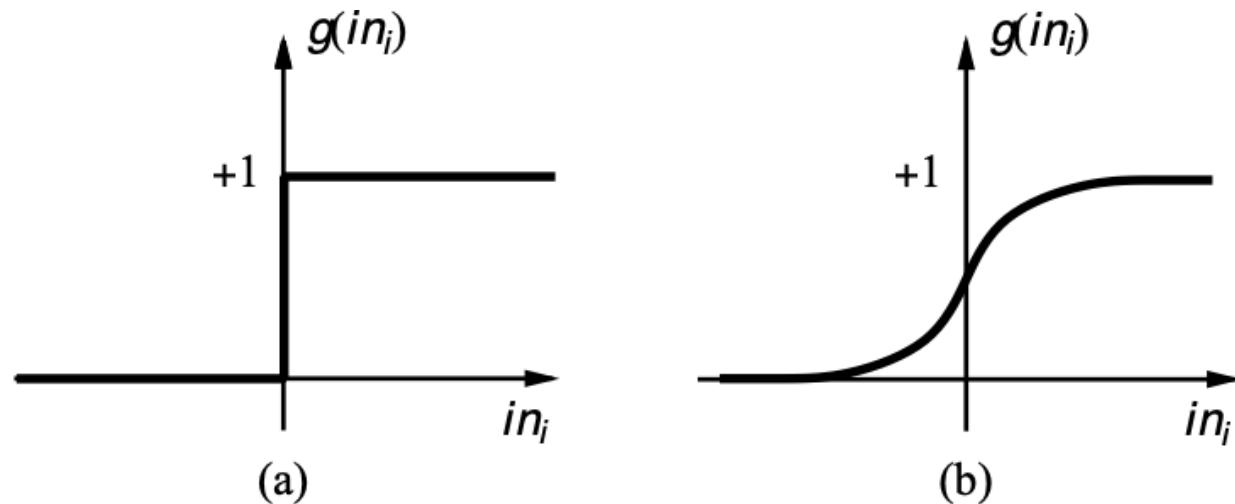
Recall that AND, OR and NOR can be implemented by perceptrons.

Multi-Layer Neural Networks



- Given an explicit logical function, we can design a multi-layer neural network by hand to compute that function.
- But, if we are just given a set of training data, can we train a multi-layer network to fit these data?

Activation functions

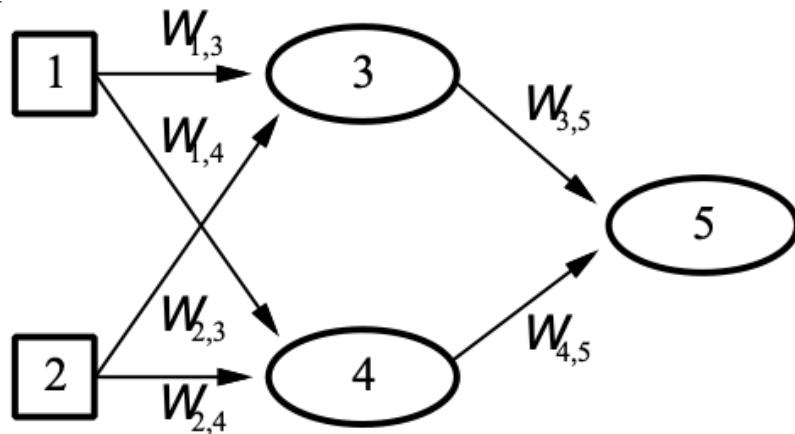


(a) is a step function or threshold function

(b) is a sigmoid function $\frac{1}{1 + e^{-x}}$

Changing the bias weight $w_{0,i}$ moves the threshold

Feed-Forward Example



$w_{i,j} \equiv$ weight between node i and node j

- Feed-forward network = a parameterised family of nonlinear functions:

$$\begin{aligned}a_5 &= g(W_{3,5} \cdot a_3 + W_{4,5} \cdot a_4) \\&= g\left(W_{3,5} \cdot g(W_{1,3} \cdot a_1 + W_{2,3} \cdot a_2) + W_{4,5} \cdot g(W_{1,4} \cdot a_1 + W_{2,4} \cdot a_2)\right)\end{aligned}$$

- Adjusting weights changes the function

ANN Training as Cost Minimisation

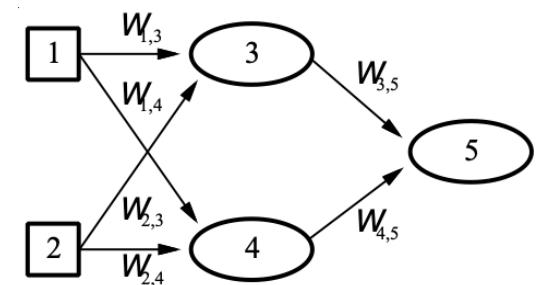
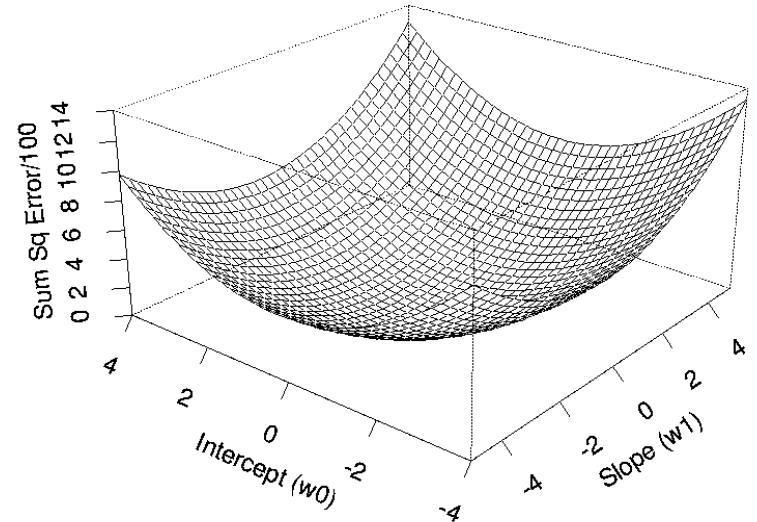
- Define error function Mean Squared Error, E

$$E = \frac{1}{2} \sum (d - y)^2$$

y = actual output

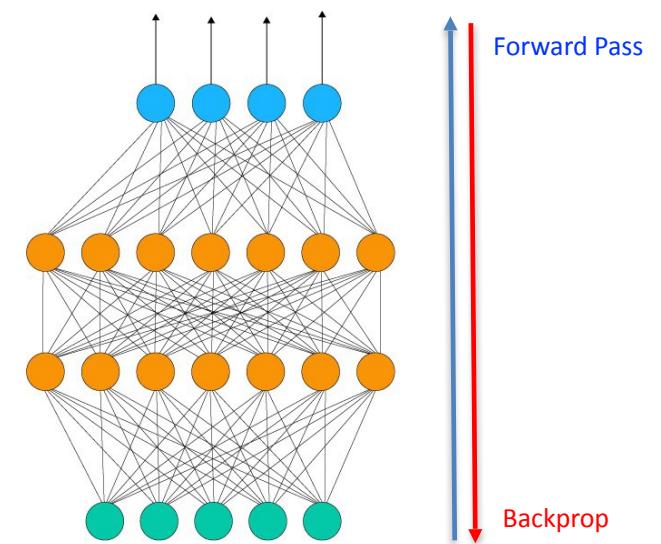
d = desired output

- Think of E as height of error landscape of weight space.
- Aim to find a set of weights for which E is very low.



Backpropagation

1. **Forward pass:** apply inputs to “lowest layer” and feed activations forward to get output
2. **Calculate error:** difference between desired output and actual output
3. **Backward pass:** Propagate errors back through network to adjust weights



Gradient Descent

$$E = \frac{1}{2} \sum (d - y)^2$$

If transfer functions are smooth, can use multivariate calculus to adjust weights by taking steepest downhill direction.

$$w \leftarrow w + \eta \frac{\partial E}{\partial w}$$

Parameter η is the **learning rate**

- How the cost function affects the particular weight
- Find the weight

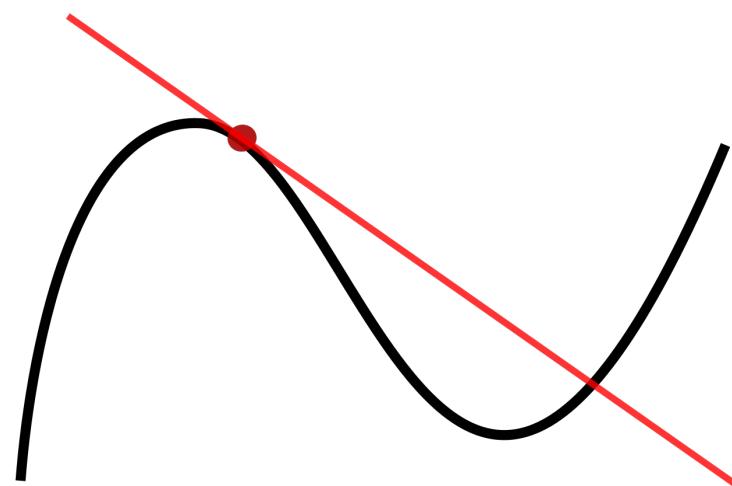
Derivative of a Function

The derivative of a function is the slope of the tangent at a point

$$y = f(x) = mx + b$$

$$m = \frac{\text{change in } y}{\text{change in } x} = \frac{\Delta y}{\Delta x}$$

Written $\frac{dy}{dx}$



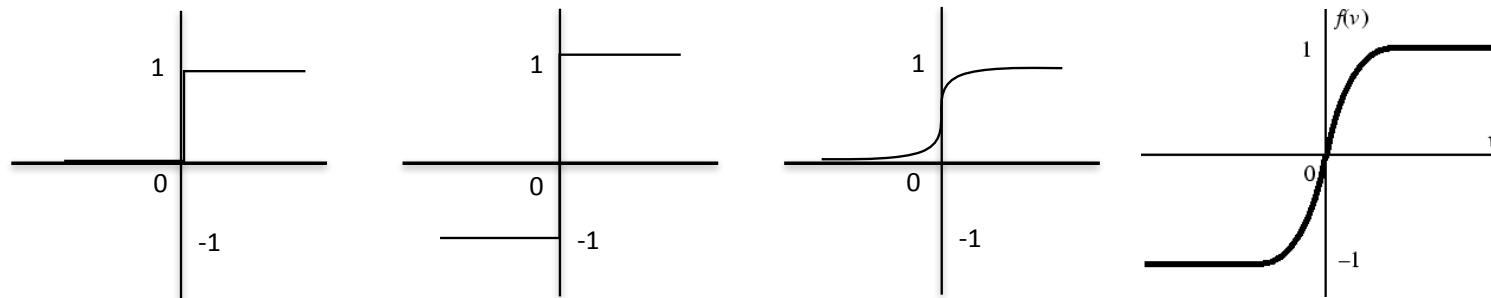
Partial Derivative

Derivative of a function of several variables with respect to one of this variables

If $z = f(x, y, \dots)$

Derivative with respect to x is written: $\frac{\partial z}{\partial x}$

Function must be continuous to be differentiable



Replace the (discontinuous) step function with a differentiable function, such as the sigmoid:

$$g(s) = \frac{1}{1 + e^{-s}}$$

or hyperbolic tangent

$$g(s) = \tanh(s) = \frac{e^s - e^{-s}}{e^s + e^{-s}} = 2\left(\frac{1}{1 + e^{-s}}\right) - 1 \quad (-1 \text{ to } 1)$$

Chain Rule

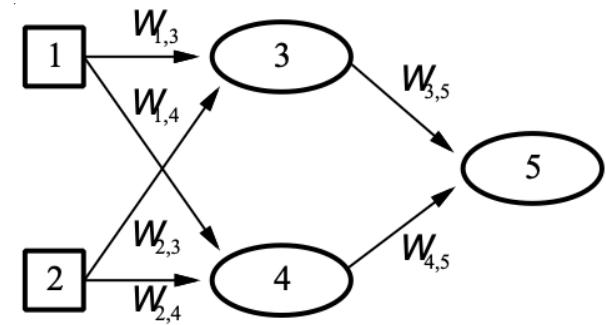
$$g(W_{3,5} \cdot g(W_{1,3} \cdot a_1 + W_{2,3} \cdot a_2) + W_{4,5} \cdot g(W_{1,4} \cdot a_1 + W_{2,4} \cdot a_2))$$

If

$$y = g(f(x)) \equiv \begin{cases} y = g(u) \\ u = f(x) \end{cases}$$

then

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial u} \cdot \frac{\partial u}{\partial x}$$



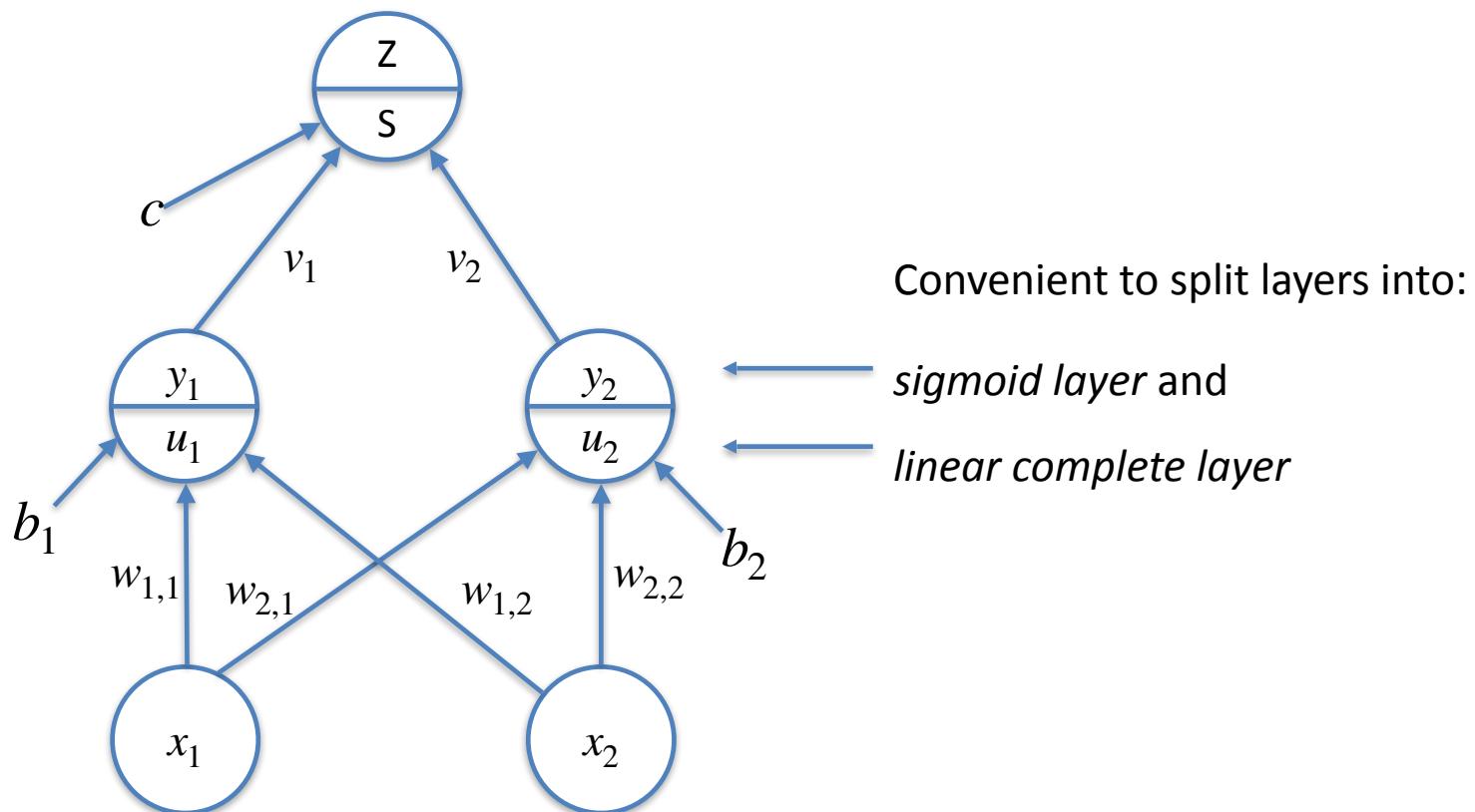
Used chain rule to compute partial derivatives efficiently.

Transfer function must be differentiable (usually sigmoid, or tanh).

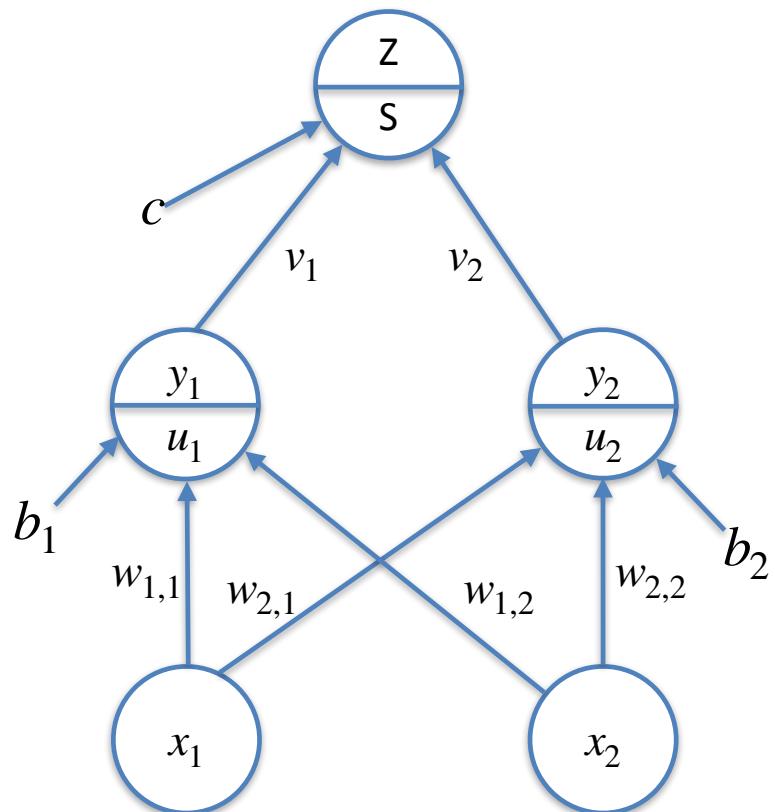
Note: if $z(s) = \frac{1}{1 + e^{-s}}$ derivative $z'(s) = z(1 - z)$

if $z(s) = \tanh(s)$ derivative $z'(s) = 1 - z^2$

Backpropagation



Forward Pass



$$u_1 = b_1 + w_{11}x_1 + w_{12}x_2$$

$$y_1 = g(u_1)$$

$$s = c + v_1y_1 + v_2y_2$$

$$z = g(s)$$

$$E = \frac{1}{2} \sum (z - t)^2 \quad (t = \text{target})$$

Backpropagation Algorithm

```
procedure BackpropLearner( $X_s$ ,  $Y_s$ ,  $E_s$ ,  $layers$ ,  $\eta$ )
    repeat
        for each example  $e$  in  $E_s$  in random order do
             $values[i] \leftarrow X_i(e)$  for each input unit  $i$ 
        for each layer from lowest to highest do
             $values \leftarrow OutputValues(layer, values)$ 
             $error \leftarrow SumSqError(Y_s(e), values)$ 
        for each  $layer$  from highest to lowest do
             $error \leftarrow Backprop(layer, error)$ 
    until termination
```

X_s : set of input features, $X_s = \{X_1, \dots, X_n\}$

Y_s : target features

E_s : set of examples from which to learn

$layers$: a sequence of layers

η : learning rate (gradient descent step size)

Backpropagation Algorithm

```
function g(x) = 1/(1 + e-x)
```

```
function SumSqError(Ys, predicted) = [ $\frac{1}{2} \sum (Ys[j] - predicted[j])^2$  for each output unit j]
```

```
function OutputValues(layer, input) // input is array with length ni  
    define input [n] to be 1  
    output[j] ← g( $\sum_{i=0}^n w_{ij} \cdot input[i]$ ) for each j // update output for layer  
    return output
```

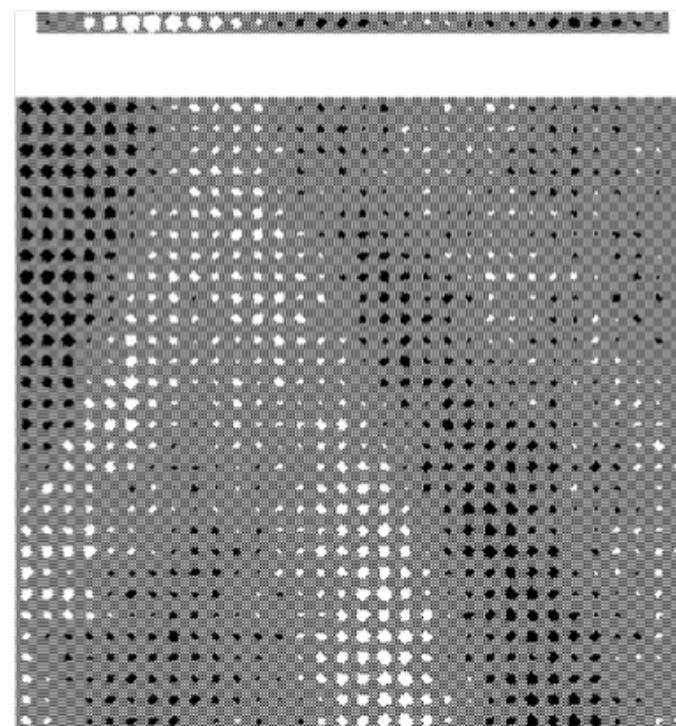
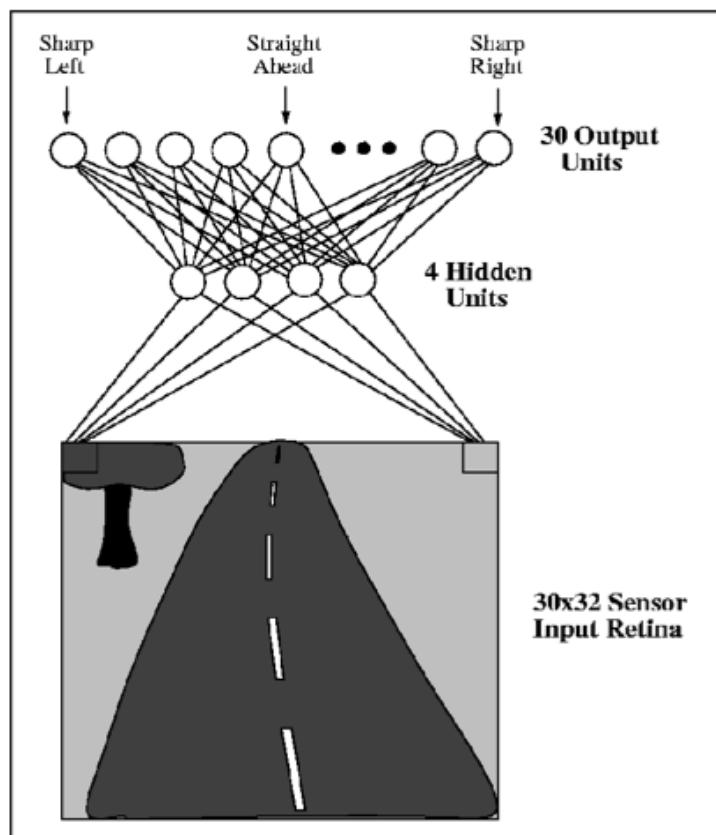
```
function Backprop(layer, error) // each layer has an input and output array  
    if output layer // error is array with length ni  
        delta = error  
    else  
        delta[i] ← output[i] · (1 – output[i]) · error[i] for each i  
        delta[i] ← output[i] · (1 – output[i]) · error[i] for each i  
        wji ← wji + η · delta[j] · input[i] for each i, j, for learning rate η // perceptron rule  
        delta[i] ←  $\sum_j w_{ji} \cdot delta[j]$  for each i  
    return delta
```

derivative

Neural Network – Applications

- Autonomous Driving
- Game Playing
- Credit Card Fraud Detection
- Handwriting Recognition
- Financial Prediction

ALVINN (First demo of long distance autonomous driving)



ALVINN

- Autonomous Land Vehicle In a Neural Network
- later version included a sonar range finder
 - 8×32 rangefinder input retina
 - 29 hidden units
 - 45 output units
- Supervised Learning, from human actions (Behavioural Cloning)
 - additional “transformed” training items to cover emergency situations
- Drove (mostly) autonomously from coast to coast in USA

Training Tips

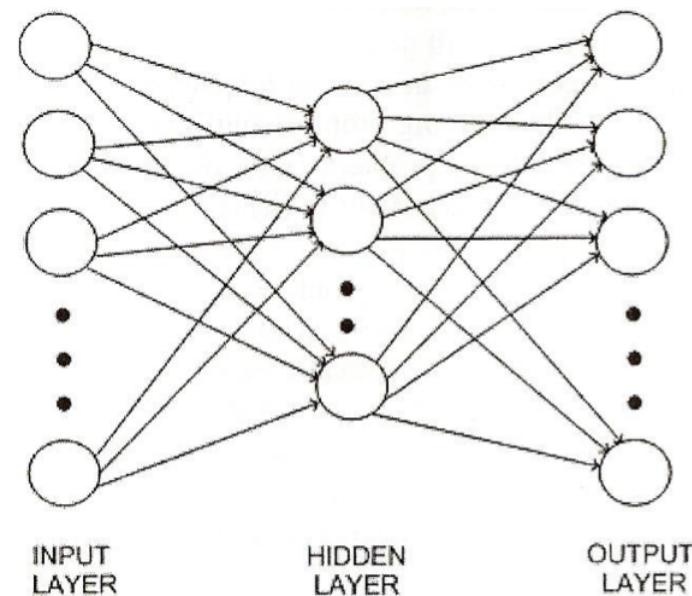
- Re-scale inputs and outputs to be in the range 0 to 1 or –1 to 1
- Initialise weights to very small random values
- On-line or batch learning
- Three different ways to prevent overfitting:
 - limit the number of hidden nodes or connections
 - limit the training time, using a validation set
 - weight decay
- Adjust the parameters: learning rate (and momentum) to suit the particular task

Neural Network Structure

Two main network structures

1. Feed-Forward
Network

2. Recurrent
Network

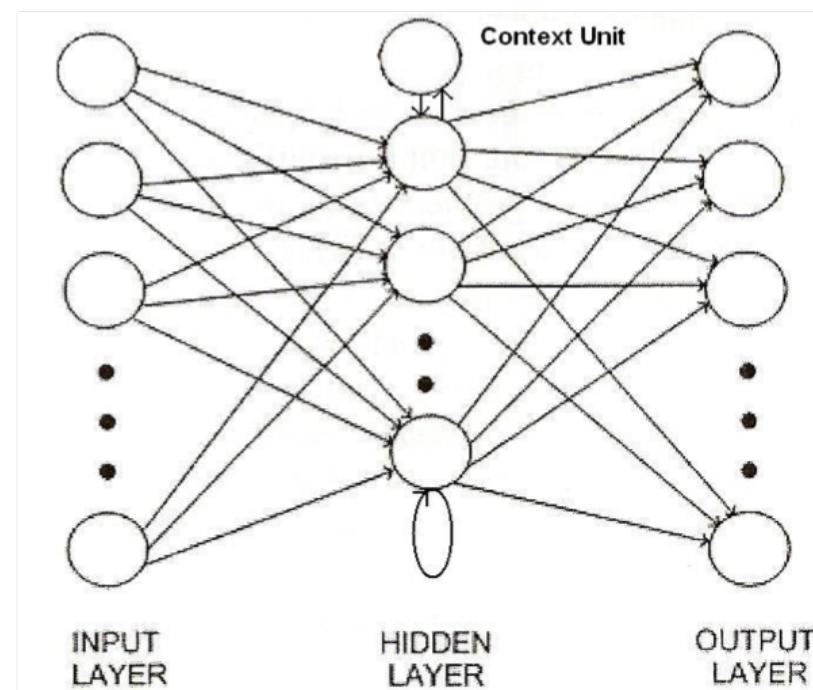


Neural Network Structure

Two main network structures

1. Feed-Forward
Network

2. Recurrent
Network



Neural Network Structures

Feed-forward network has connections only in one direction

- Every node receives input from “upstream” nodes; delivers output to “downstream” nodes
 - no loops.
- Represents a function of its current input
 - has no internal state other than the weights themselves.

Recurrent network feeds outputs back into its own inputs

- Activation levels of network form a dynamical system
 - may reach a stable state or exhibit oscillations or even chaotic behaviour
- Response of network to an input depends on its initial state
 - which may depend on previous inputs.
- Can support short-term memory

Neural Networks

- Multiple layers form a hierarchical model, known as **deep learning**
 - **Convolutional neural networks** are specialised for vision tasks
 - **Recurrent neural networks** are used for time series
- Typical real-world network can have 10 to 20 layers with hundreds of millions of weights
 - can take hours, days, months to learn on machines with thousands of cores

Summary

- Vector-valued inputs and outputs
- Multi-layer networks can learn non-linearly separable functions
- Hidden layers learn intermediate representation
 - ◆ How many to use?
- Prediction – Forward propagation
- Gradient descent (Back-propagation)
 - ◆ Local minima problems
- Kernel trick can be introduced through a deep belief network

References

- Poole & Mackworth, Artificial Intelligence: Foundations of Computational Agents, Chapter 7.
- Russell & Norvig, *Artificial Intelligence: a Modern Approach*, Chapters 18.6, 18.7 .

INDUCTIVE LOGIC PROGRAMMING

COMP3411/9814 Artificial Intelligence

Shape of Discriminator

- Machine Learning algorithms can be characterised by the way they divide up the attribute space.
- What is the shape of the surface that separates classes?

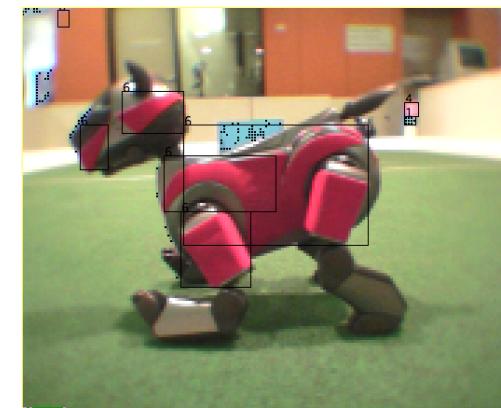
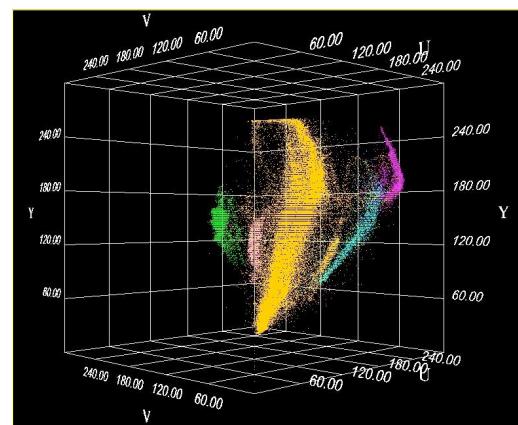
Learning in Perception

```
105, 117, 113, orange  
105, 116, 112, orange  
102, 117, 113, orange  
102, 116, 114, orange  
103, 117, 111, orange  
103, 117, 112, orange  
103, 118, 110, orange  
99, 117, 112, orange  
98, 116, 118, orange  
99, 116, 117, orange  
106, 111, 114, orange  
114, 115, 123, yellow  
128, 111, 124, yellow  
150, 112, 121, yellow  
173, 111, 117, yellow  
171, 110, 110, yellow  
145, 112, 108, yellow  
121, 111, 110, yellow  
106, 111, 112, orange  
107, 112, 112, orange  
104, 114, 114, orange  
100, 115, 114, orange  
100, 117, 117, orange  
98, 115, 113, orange  
100, 114, 116, orange  
97, 117, 112, orange  
102, 115, 109, orange  
104, 118, 109, orange  
100, 114, 108, orange  
97, 115, 110, orange  
101, 114, 110, orange  
99, 116, 113, orange  
98, 116, 113, orange
```

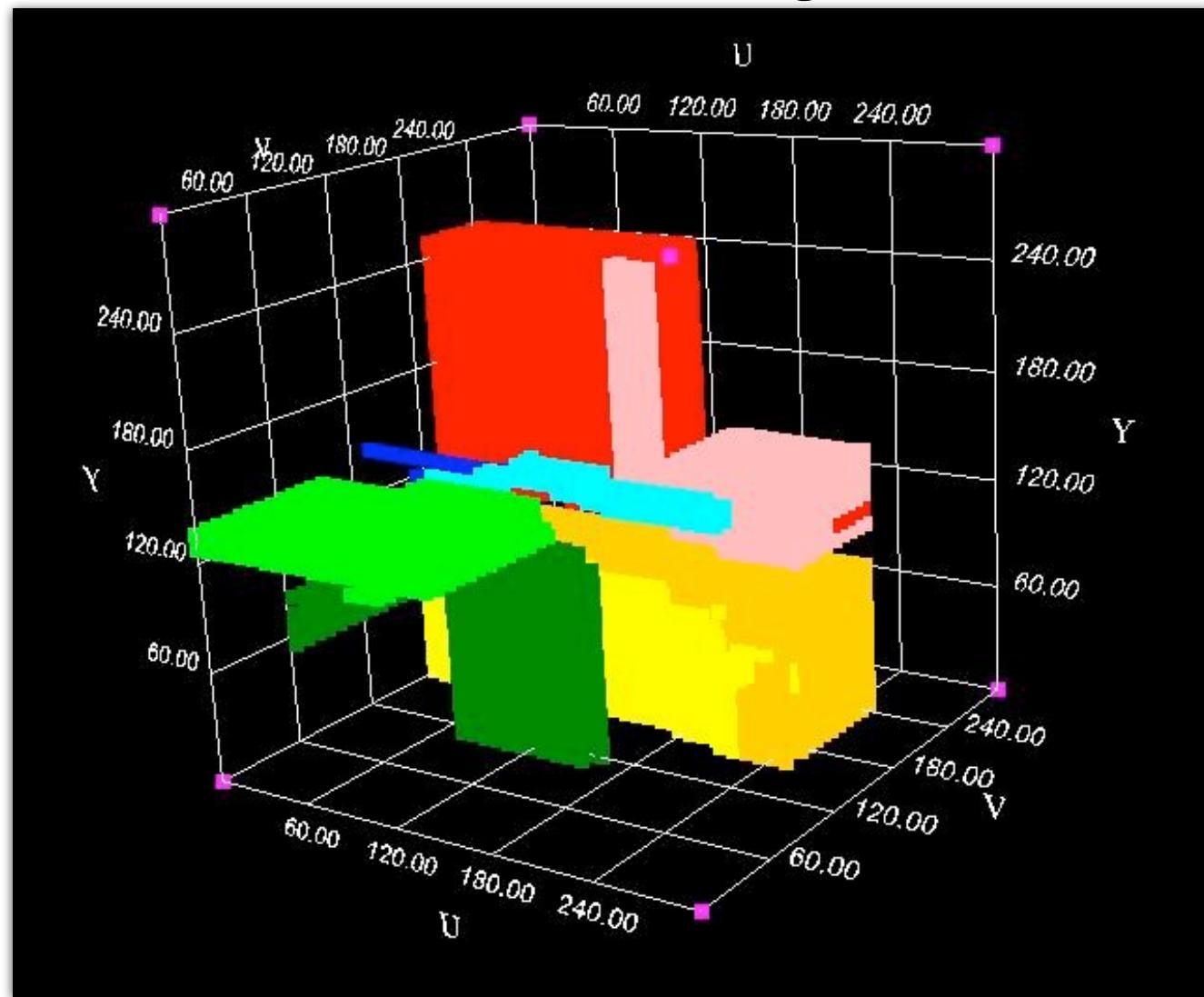
C4.5

Warning: In practice data sets and decision trees are much larger than this example!

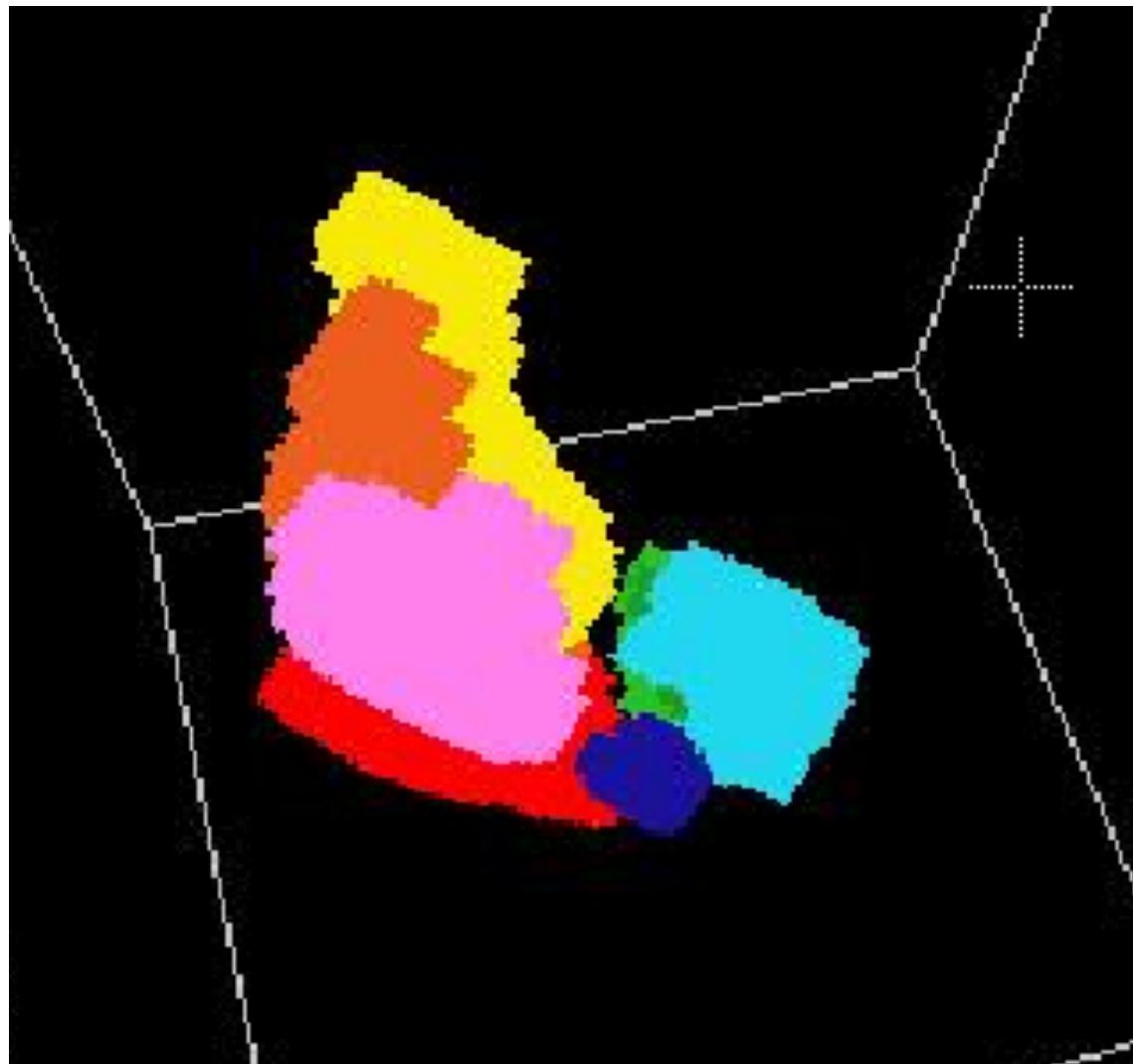
```
if (u <= 107)  
    yellow;  
else  
    if (v <= 100)  
        orange;  
    else  
        if (y <= 136)  
            orange;  
        else  
            yellow;
```



Colour Classes using C4.5



Nearest Neighbour



Description Language

- A concept can also be represented by sentences in a description language.
- May be if-then-else, or rules, like Horn clauses (Prolog):

The colour decision tree can be written as:

```
yellow :- u <= 107.  
yellow :- h > 107, v <= 100, y > 136.  
orange :- u > 107, v <= 100, y <= 136.
```

Generalisation Ordering

- If we can define a generalisation ordering on a language, learning can be done by syntactic transformations.
- E.g

$$class \leftarrow size = large \tag{1}$$

is a generalisation of

$$class \leftarrow size = large \wedge colour = red \tag{2}$$

because (2) describes a more constrained set

Subsumption

A clause C_1 *subsumes*, or is more general than, another clause C_2 if there is a substitution σ such that $C_2 \supseteq C_1 \sigma$.

The least general generalisation of

$class \leftarrow size = large$
$class \leftarrow size = large \wedge colour = red$

$$p(g(a), a) \tag{3}$$

and $p(g(b), b) \tag{4}$

is $p(g(X), X). \tag{5}$

Under the substitution $\{a / X\}$ (5) is equivalent to (3).

Under the substitution $\{b / X\}$ (5) is equivalent to (4).

Inverse Substitution

The least general generalisation of

$$p(g(a), a)$$

and $p(g(b), b)$

is $p(g(X), X).$

and results in the inverse substitution $\{X / \{a, b\}\}$

Least General Generalisation

E.g.

The result of heating this bit of iron to 419°C was that it melted.

The result of heating that bit of iron to 419°C was that it melted.

The result of heating any bit of iron to 419°C was that it melted.

We can formalise this as:

`melted(bit1) :- bit_of_iron(bit1), heated(bit1, 419).`

`melted(bit2) :- bit_of_iron(bit2), heated(bit2, 419).`

`melted(X) :- bit_of_iron(X), heated(X, 419).`

Least General Generalisation

- Find a substitution so that there is no other clause that is more general

LGG of Clauses

$q(g(a)) :- p(g(a), h(b)), r(h(b), c), r(h(b), e).$

$q(x) :- p(x, y), r(y, z), r(h(w), z), s(a, b).$

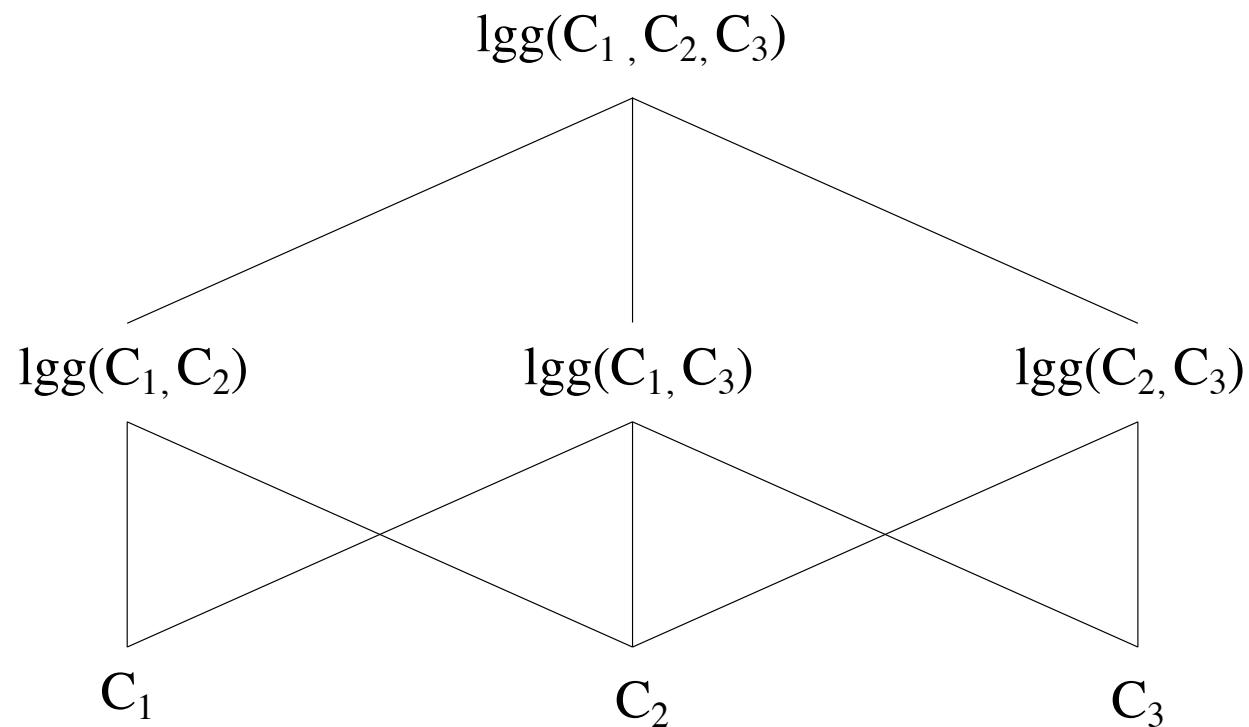
results in an LGG:

$q(X) :- p(X, Y) , r(Y, Z) , r(h(U), Z) , r(Y, V) , r(h(U), V)$

with inverse substitutions:

$\{X/(g(a), x), Y/(h(b), y), Z/(c, z), U/(b, w), V/(e, z)\}$

LGG of sets of clauses



Background Knowledge

- Background knowledge can assist learning
- It must be possible to interpret a concept description as a recognition procedure.
- If the description of chair has been learned, then it should be possible to refer to chair in other concept descriptions.
- E.g. the chair “program” will recognise the chairs in an office scene.

Saturation

Given a set of clauses, the body of one of which is completely contained in the bodies of the others, such as:

$$X \leftarrow A \wedge B \wedge C \wedge D \wedge E$$

$$Y \leftarrow A \wedge B \wedge C$$

we can *saturate* the first clause:

$$X \leftarrow A \wedge B \wedge C \wedge D \wedge E \wedge Y$$

Saturation Example

Suppose we are given two instances of a concept `cuddly_pet`,

`cuddly_pet(X) ← fluffy(X) ∧ dog(X)`

`cuddly_pet(X) ← fluffy(X) ∧ cat(X)`

and:

`pet(X) ← dog(X)`

`pet(X) ← cat(X)`

Saturated clauses are:

`cuddly_pet(X) ← fluffy(X) ∧ dog(X) ∧ pet(X)`

`cuddly_pet(X) ← fluffy(X) ∧ cat(X) ∧ pet(X)`

LG_G is

`cuddly_pet(X) ← fluffy(X) ∧ pet(X)`

Relative Least General Generalisation (RLGG)

- Apply background knowledge to *saturate* example clauses.
- Find LGG of saturated clauses

```
heavier(A, B) :- denser(A, B), larger(A, B).
```

```
fall_together(hammer, feather) :-  
    same_height(hammer, feather),  
    denser(hammer, feather),  
    larger(hammer, feather).
```

```
fall_together(hammer, feather) :-  
    same_height(hammer, feather),  
    denser(hammer, feather),  
    larger(hammer, feather),  
    heavier(hammer, feather).
```

GOLEM

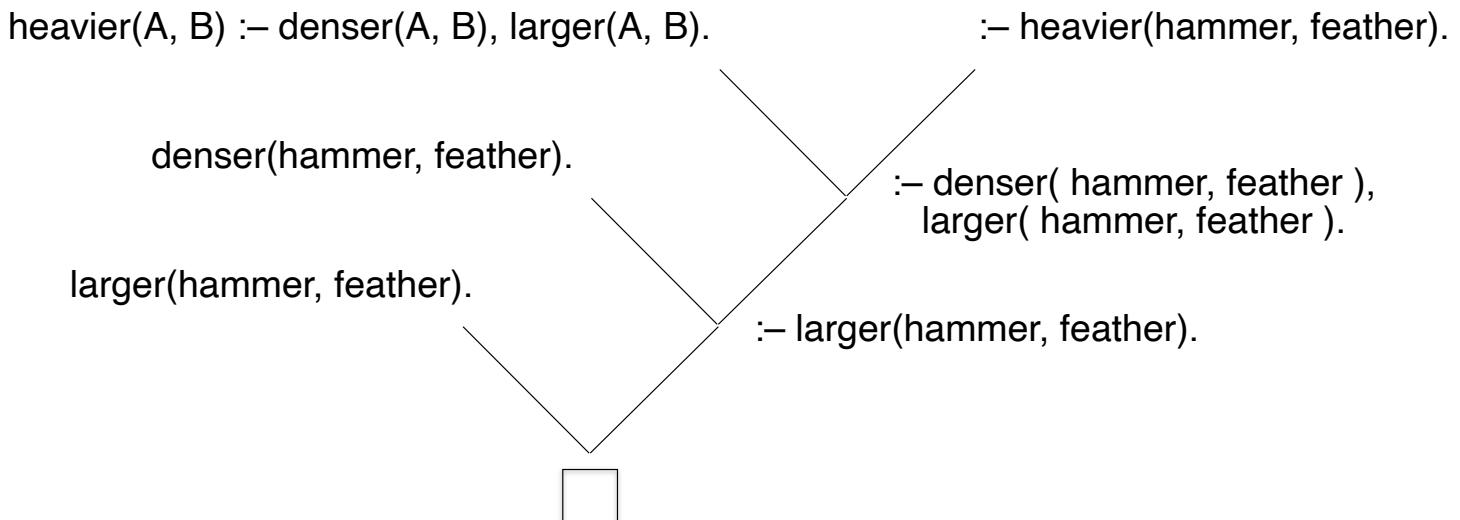
- LGG is very inefficient for large numbers of examples
- GOLEM uses a *hill-climbing* as an approximation
 - Randomly select pairs of examples
 - Find LGG's and pick the one that covers most positive examples and excludes all negative examples, call it **S**.
 - Randomly select another set of examples
 - Find all LGG's with S
 - Pick best one
 - Repeat as long as cover of positive examples increases.

Inverting Resolution

- Resolution provides an efficient means of deriving a solution to a problem, giving a set of axioms which define the task environment.
- Resolution takes two terms and resolves them into a most general unifier.
- Anti-unification finds the *least general generalisation* of two terms.

Resolution Proofs

```
larger(hammer, feather).  
denser(hammer, feather).  
heavier(A, B) :- denser(A, B), larger(A, B).  
heavier(hammer, feather)?
```



Absorption

Given a set of clauses, the body of one of which is completely contained in the bodies of the others, such as:

$$X \leftarrow A \wedge B \wedge C \wedge D \wedge E$$

$$Y \leftarrow A \wedge B \wedge C$$

we can hypothesise:

$$X \leftarrow Y \wedge D \wedge E$$

$$Y \leftarrow A \wedge B \wedge C$$

Intra-construction

This is the distributive law of Boolean equations. Intra-construction takes a group of rules all having the same head, such as:

$$X \leftarrow B \wedge C \wedge D \wedge E$$

$$X \leftarrow A \wedge B \wedge D \wedge F$$

and replaces them with:

$$X \leftarrow B \wedge D \wedge Z$$

$$Z \leftarrow C \wedge E$$

$$Z \leftarrow A \wedge F$$

Intra-construction automatically creates a new term in its attempt to simplify descriptions.

Automatic Programming

```
member(blue, [blue]).  
member(eye, [eye, nose, throat]).  
  
Is member(A, [A|B]) always true? y  
  
Is member(A, [B|C]) always true? n  
  
member(2,[1,2,3,4,5,6]).  
  
Is member(A,[B,A|C]) always true? y  
  
Is member(A,[B|C]) :- member(A,C) always true? y
```

Generalisation:

```
member(A, [A|B]).  
member(A, [B|C]) :- member(A, C).
```

Problems with Incremental Learning

- Experiments can never validate a world model.
- Experiments usually involve noisy data, they can cause damage to the environment, they may cause misleading side-effects.
- A robot may have an incomplete theory and incorrect model.
- Need to be able to handle exceptions.
- Need to be able to repair knowledge base.
- If concepts are represented by Horn clauses, we can use a program debugger (declarative diagnosis).

Repairing Theories

Set the theory T to $\{ \}$

repeat

 Examine the next example

repeat

while the theory T is too general (covers -ve example) do

 Specialise T

while the theory is too specific (doesn't cover +ve example) do

 Generalise T

until the conjecture T is neither too general nor too

 specific with respect to the known facts

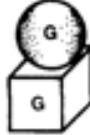
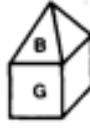
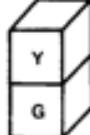
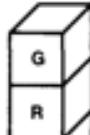
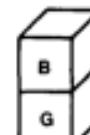
 Output T

forever

Exceptions

Multi-level Counterfactuals

- Form a cover for +ve examples
 - If -ve examples are also covered, for a new cover of -ve examples and add it as an exception
 - If +ve examples are excluded now, reverse process
1. (ON .X .Y)(GREEN .Y)(CUBE .Y)
 2. (ON .X .Y)(GREEN .Y)(CUBE .Y)~((BLUE .X) ~(PYRAMID .X))

POSITIVE INSTANCES		NEGATIVE INSTANCES	
(ON T1 T2) (SPHERE T1) (GREEN T1) (CUBE T2) (GREEN T2)		(ON T10 T11) (SPHERE T10) (BLUE T10) (CUBE T11) (GREEN T11)	
P1	V1		
(ON T3 T4) (PYRAMID T3) (BLUE T3) (CUBE T4) (GREEN T4)		(ON T12 T13) (SPHERE T12) (GREEN T12) (CUBE T13) (BLUE T13)	
P2	V2		
(ON T5 T6) (CUBE T5) (YELLOW T5) (CUBE T6) (GREEN T6)		(ON T14 T15) (ON T15 T16) (CUBE T14) (YELLOW T14) (CUBE T15) (BLUE T15) (CUBE T16) (GREEN T16)	
P3	V3		
(ON T7 T8) (ON T8 T9) (CUBE T7) (GREEN T7) (CUBE T8) (RED T8) (CUBE T9) (GREEN T9)		(ON T17 T18) (CUBE T17) (BLUE T17) (CUBE T18) (GREEN T18)	
P4	V4		

Exceptions or Noise?

- If there is noise, then exceptions will start to track noise, causing, "overfitting".
- Must have a stopping criterion that prevents clause from growing too much.
- Some -ve examples may still be covered and some +ve examples may not.
- Use Minimum Description Length heuristic.

Minimum Description Length

- Devise an encoding that maps a theory (set of clauses) into a bit string.
- Also need an encoding for examples.
- Number of bits required to encode theory should not exceed number of bits to encode +ve examples.

Compaction

- Use a measure of compaction to guide search.
- More than one compaction operator applicable at any time.
- A measure is applied to each rule to determine which one will result in the greatest compaction.
- The measure of compaction is the reduction in the number of symbols in the set of clauses after applying an operator.
- Each operator has an associated formula for computing this reduction.
- Best-first search.

Summary

- If a concept can be represented by sentences in a description language, concepts can be learned by generalising sentences in language
- Machine Learning as search through the space of possible sentences for the most compact that best covers +ve examples and excludes –ve examples
 - Least general generalisation
 - Inverse resolution
 - Automatic Programming

References

- G. D. Plotkin. A note on inductive generalization. In B. Meltzer and D. Michie, editors, *Machine Intelligence 5*, pages 153–163. Edinburgh University Press, 1970.
- G. D. Plotkin. A further note on inductive generalization. In B. Meltzer and D. Michie, editors, *Machine Intelligence 6*. Elsevier, New York, 1971.
- C. A. Sammut and R. B. Banerji. Learning concepts by asking questions. In R. S. Michalski, J. Carbonell, and T. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach*, Vol 2, pages 167–192. Morgan Kaufmann, Los Altos, California, 1986.
- S. Muggleton. Inductive logic programming. *New Generation Computing*, 8:295–318, 1991.
- S. Muggleton and C. Feng. Efficient induction of logic programs. In *First Conference on Algorithmic Learning Theory*, Tokyo, 1990. Omsha.
- S. Muggleton, W.-Z. Dai, C. Sammut, A. Tamaddoni-Nezhad, J. Wen, and Z.-H. Zhou. Meta-interpretive learning from noisy images. *Machine Learning*, 107(7):1097 – 1118, 2018.

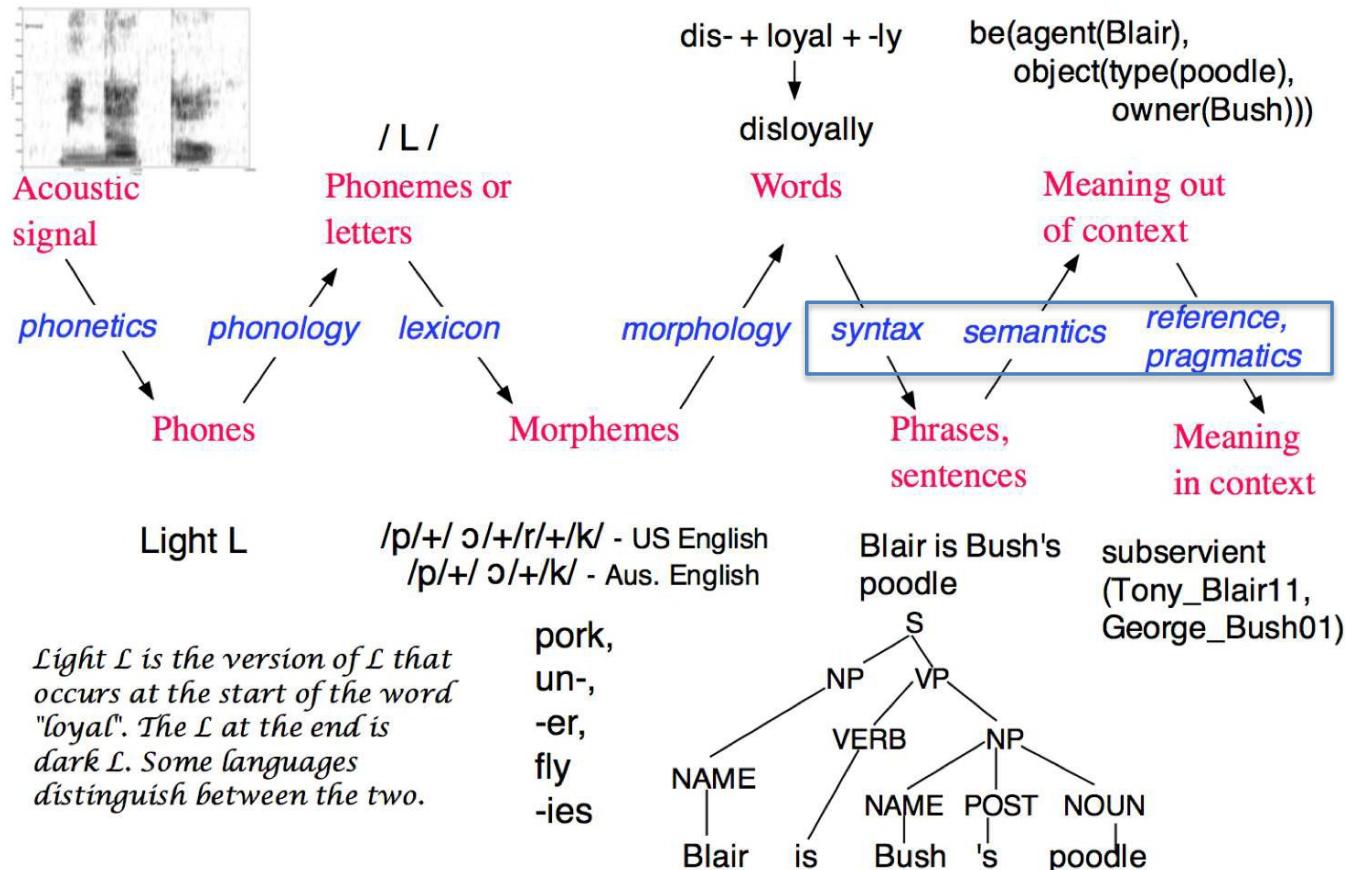
COMP3411: Artificial Intelligence

Grammars and Parsing

This Lecture

- Overview of Natural Languages
- Syntax and Grammar for Natural Languages

Linguistics Landscape



Natural Langue Processing

- Syntax
 - Linguistic Knowledge
 - Grammars and Parsing
 - Probabilistic Parsing
- Semantics
 - Semantic Interpretation and Logical Form
- Pragmatics
 - Discourse Processing
 - Speech Act Theory
 - (Spoken) Dialogue Systems

Related Disciplines

- Linguistics
 - Study of language in the abstract and particular languages
- Psycholinguistics
 - Psychological models of human language processing
- Neurolinguistics
 - Neural models of human language processing
- Logic
 - Study of formal reasoning

NLP Applications

- Chatbots
 - Customer service, e.g. CBA, Amtrak, Lyft, Spotify, Whole Foods
- Personal Assistants
 - Siri, Alexa, Google Assistant
- Information Extraction
 - Financial reports, news articles
- Machine (Assisted) Translation
 - Weather reports, EU contracts, Canada Hansard
- Social Robotics
 - Home care robots

Central Problem – Ambiguity

- Natural languages exhibit **ambiguity**

“The fisherman went to the bank” (lexical)

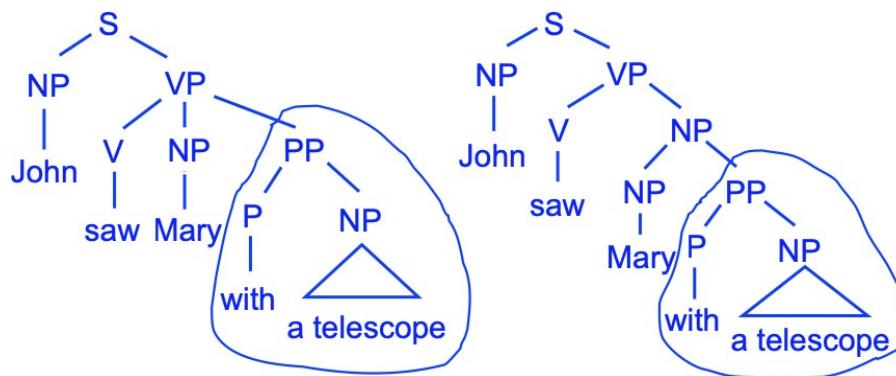
“The boy saw a girl with a telescope” (structural) “Every student took an exam” (semantic)

“The table won’t fit through the doorway because it is too [wide/narrow]” (pragmatic)

- Ambiguity makes it difficult to interpret meaning of phrases/ sentences
 - But also makes inference harder to define and compute
 - Resolve ambiguity by mapping to unambiguous representation

Structural Ambiguity

“John saw Mary with a telescope”



- Different interpretation → different representation

“John sold a car to Mary” and “Mary was sold a car by John”

- Same interpretation → same representation

Syntax

- Linguistic Knowledge and Grammars
- Context Free Grammars
- Parsing
 - Top Down Parsing
 - Bottom Up Parsing
 - Chart Parsing
 - Deterministic Parsing
 - Probabilistic Parsing

Sentence vs Utterance

- Sentence is a group of words that convey some meaning
- An utterance is a group of words between pauses in speech.

Lexical Items (Basic Words)

- Open class (can add new words)
 - Nouns: denote objects (e.g. cat, John, justice)
 - Verbs: denote actions, events (e.g. buy, break, believe)
 - Adjectives: denote properties of objects (e.g. red, large)
 - Adverbs: denote properties of events (e.g. quickly)
- Closed class (function words, mostly fixed in the language)
 - Prepositions: at, in, of, on, . . .
 - Articles: the, a, an
 - Conjunctions: and, or, if, then, than, . . .

Sentence Forms

- Declarative (indicative)
 - Bart is listening.
- Yes/No question (interrogative)
 - Is Bart listening?
- Wh-question (interrogative)
 - When is Bart listening?
- Imperative (command)
 - Listen, Bart!
- Subjunctive (conditional or imaginary situations)
 - If Bart were listening, he might hear something useful.

Noun Phrases

- Noun phrases: occur as “subject” with a range of “predicates”
 - (noun phrase) ate the bone
 - (noun phrase) saw the bird in the sky
 - (noun phrase) believes that $2 + 2 = 4$
- Examples
 - John, The dog, The big ugly dog, The man in the red car, The oldest man in the world with a beard, The oldest man who lives in China, . . .
- Sentences need not “make sense”

Verb Phrases

- Verb phrases: occur as “predicate” with a range of “subjects”
 - John (verb phrase)
 - The dog (verb phrase)
 - Any noun phrase (verb phrase)
- Examples
 - ate the bone
 - saw the bird in the sky
 - believes that $2 + 2 = 4$
- Verb phrase depends on noun phrase

Inside Noun Phrases

- Within noun phrase
 - Main item (the head of the phrase): noun
 - Optional specifiers
 - Determiners (articles, demonstratives, quantifiers)
 - Adjectives and other nouns
 - Mandatory arguments
 - Depend on head (e.g. capital (of France))
 - Optional modifiers
 - Adjectival phrases (e.g. larger than Spain)
 - Prepositional phrases (e.g. in the park)
 - Relative clauses (e.g. who likes beer)
 - Order specifiers, head, modifiers in English (e.g. firstly, ...)

Inside Verb Phrases

- Within verb phrase
 - Main item (the head of the phrase): verb
 - Optional specifiers
 - Auxiliary verbs (e.g. do, does, will, might, . . .)
 - Adverbs (e.g. quickly)
 - Mandatory arguments
 - depend on head (e.g. bought (a book) (for Henry))
 - Optional modifiers
 - Adverbial phrases (e.g. more quickly than Henry)
 - Notice similar structure to noun phrases

Prepositional Phrases

- Within prepositional phrase
 - Main item (the head of the phrase): preposition
 - Mandatory arguments
 - (noun phrase) (e.g. in the park)
 - Nouns, verbs, etc. are just the heads of phrases

Context Free Grammars

- Terminal symbols (lexical items)
- Nonterminal symbols (grammatical categories)
- Start symbol (a nonterminal) e.g. (sentence)
- Rewrite rules
 - nonterminal → sequence of nonterminals, terminals
 - e.g. (sentence) → (noun phrase) (verb phrase)
- Open question: is English context free?

Typical (Small) Grammar

$S \rightarrow NP\ VP$

$NP \rightarrow [Det]\ Adj^* N [AP\mid PP\mid Rel\ Clause]^*$

$VP \rightarrow V [NP] [NP] PP^*$

$AP \rightarrow Adj\ PP$

$PP \rightarrow P\ NP$

$Det \rightarrow a \mid an \mid the \mid \dots$

$N \rightarrow John \mid Mary \mid park \mid telescope \mid \dots$

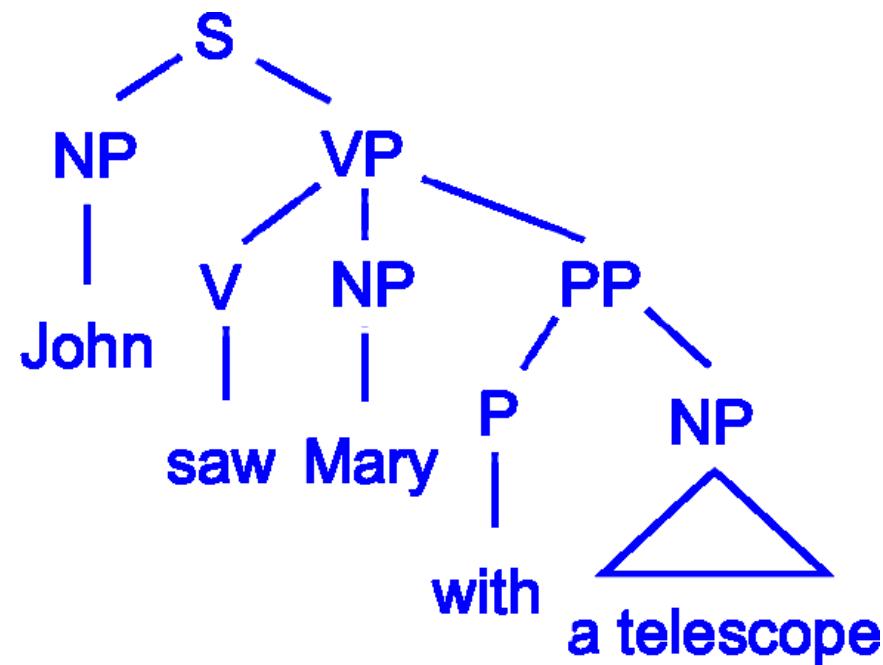
$V \rightarrow saw \mid likes \mid believes \mid \dots$

$Adj \rightarrow hot \mid hotter \mid \dots$

$P \rightarrow in \mid \dots$

Extra notation: * is “0 or more”; [. . .] is “optional”

Syntactic Structure



Syntactically ambiguous = more than one parse tree

(Leftmost) Derivation of Example

S
⇒ NP VP

⇒ N VP

⇒ John VP

⇒ John V NP PP

⇒ John saw NP PP

⇒ John saw N PP

⇒ John saw Mary PP

⇒ John saw Mary P NP

⇒ John saw Mary with NP

⇒ John saw Mary with Det N

⇒ John saw Mary with a N

⇒ John saw Mary with a telescope

⇒ means “rewrites as”

S → NP VP
NP → [Det] Adj* N [AP | PP | Rel Clause]*
VP → V [NP] [NP] PP*
AP → Adj PP
PP → P NP
Det → a | an | the | ...
N → John | Mary | park | telescope | ...
V → saw | likes | believes | ...
Adj → hot | hotter | ...
P → in | ...

Rightmost Derivation

S

\Rightarrow NP VP

\Rightarrow NP V NP PP

\Rightarrow NP V NP P NP

\Rightarrow NP V NP P Det N

\Rightarrow NP V NP P Det telescope

\Rightarrow NP V NP P a telescope

\Rightarrow ...

\Rightarrow ...

\Rightarrow ...

$S \rightarrow NP VP$

$NP \rightarrow [Det] Adj^* N [AP | PP | Rel Clause]^*$

$VP \rightarrow V [NP] [NP] PP^*$

$AP \rightarrow Adj PP$

$PP \rightarrow P NP$

$Det \rightarrow a | an | the | ...$

$N \rightarrow John | Mary | park | telescope | ...$

$V \rightarrow saw | likes | believes | ...$

$Adj \rightarrow hot | hotter | ...$

$P \rightarrow in | ...$

Parsing

- Aim is to compute a derivation of a sentence
 - produces parse tree
- Methods
 - Top down
 - Start with S , apply rewrite rules until sentence reached
 - Bottom up
 - Start with sentence, apply rewrite rules “in reverse” until S is reached
 - Chart parsing
 - Chart records parsed fragments and hypotheses
 - Can mix top down and bottom up strategies

Top-Down Parsing

- Use a stack to record working hypothesis
- Start with S as only symbol on stack
- At each step
 - Rewrite top of stack T using grammar rule $T \rightarrow \text{RHS}$
 - i.e. replace T by RHS (in reverse order), OR
 - Match word on top of stack to next word in sentence
 - Apply backtracking on failure
 - Accept sentence when stack is empty and all words in sentence matched; reject sentence when no rules to try
 - Produces leftmost derivation

Example

STACK	INPUT
S	John saw Mary with a telescope
NP VP	John saw Mary with a telescope
N VP	John saw Mary with a telescope
John VP	John saw Mary with a telescope
VP	saw Mary with a telescope
V NP PP	saw Mary with a telescope
Saw NP PP	saw Mary with a telescope
NP PP	Mary with a telescope
...	...
...	...

Bottom Up Parsing

- Use a stack to record parsed (left-right) fragment
- Start with stack empty
- At each step
 - Rewrite sequence at top of stack using rule $T \rightarrow \text{RHS}$ i.e. replace RHS (in reverse) by T, OR
 - Move word from input to stack
- Apply backtracking on failure
- Accept sentence when input empty and stack contains S; reject sentence when no more rules to try
- Produces rightmost derivation (in reverse)

Example

STACK	INPUT
	John saw Mary with a telescope
John	saw Mary with a telescope
N	saw Mary with a telescope
NP	saw Mary with a telescope
NP saw	Mary with a telescope
NP V	Mary with a telescope
NP V Mary	with a telescope
NP V N	with a telescope
...	...
...	...

Chart Parsing

- Top-down and bottom-up parsers may have to repeat parsing because they backtrack
- E.g. “The old man the boats”:
 - “man” can be a noun or a verb
 - Initially “old man” might be put together as a noun phrase,
 - but then we see that “man” is the verb
- A chart parser maintains a table or graph of all possible parsed fragments to avoid backtracking

“Garden Path” Sentence

Grammar:

1. $S \rightarrow NP\ VP$
2. $NP \rightarrow ART\ N$
3. $NP \rightarrow ART\ ADJ\ N$
4. $VP \rightarrow V\ NP$

Lexicon:

- the: ART
old: ADJ, N
man: N, V
boat: N

Sentence: 1 The 2 old 3 man 4 the 5 boat 6

Chart Parser Example

Grammar:

1. $S \rightarrow NP VP$
2. $NP \rightarrow ART N$
3. $NP \rightarrow ART ADJ N$
4. $VP \rightarrow V NP$

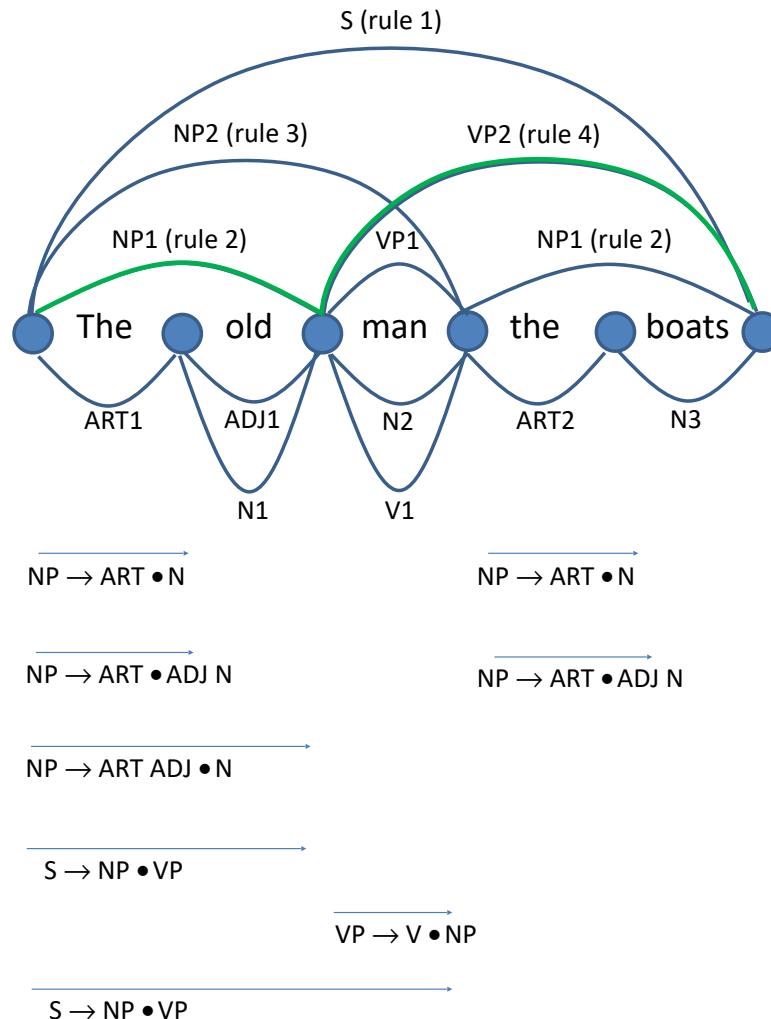
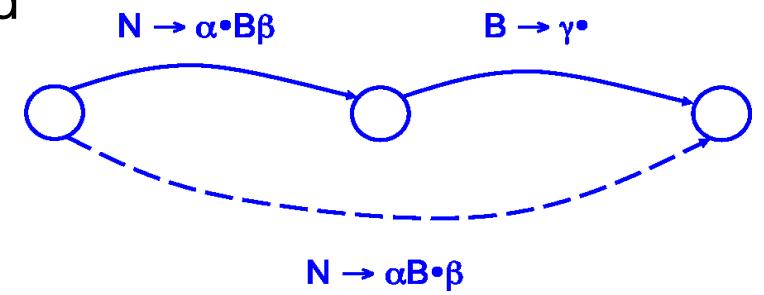
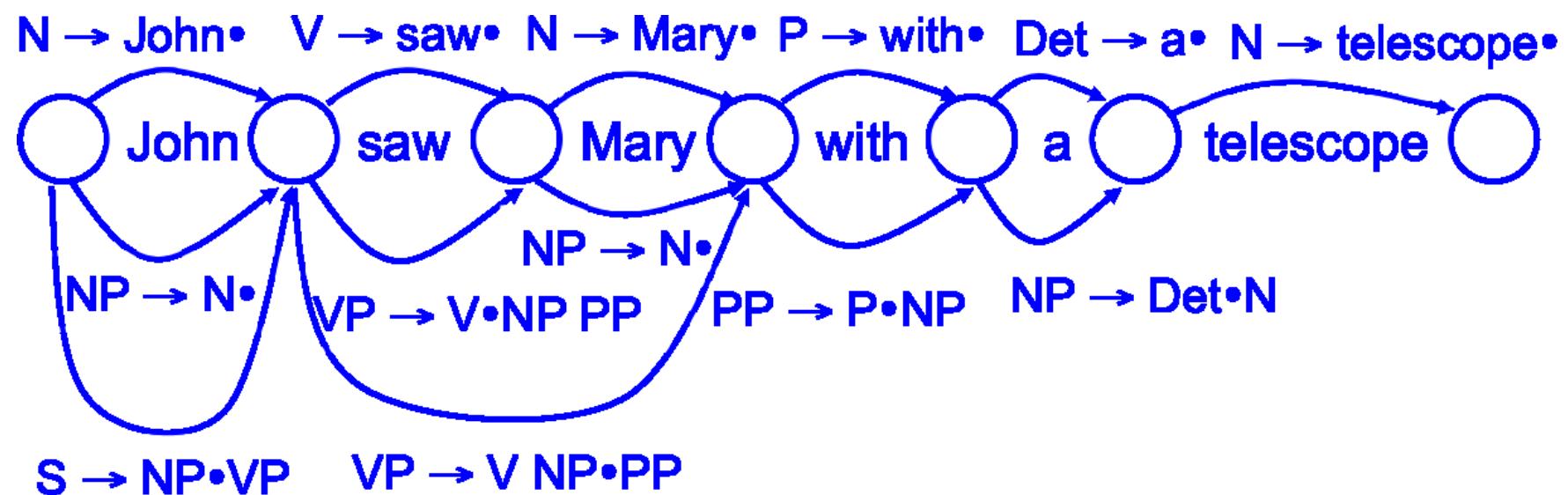


Chart Parsing

- Use a chart to record parsed fragments and hypotheses
- Hypotheses $N \rightarrow a \cdot \beta$ where $N \rightarrow a\beta$ is a grammar rule means “trying to parse N as $a\beta$ and have so far parsed a ”
- One node in chart for each word gap, start and end
- One arc in chart for each hypothesis
- At each step, apply fundamental rule
 - If chart has $N \rightarrow a \cdot B\beta$ from n_1 to n_2 and $B \rightarrow \gamma \cdot$ from n_2 to n_3
 - add $N \rightarrow aB \cdot \beta$ from n_1 to n_3
- Accept sentence when $S \rightarrow a \cdot$ is added from start to end
- Can produce any sort of derivation



Example Chart



Comparing Parsing Methods

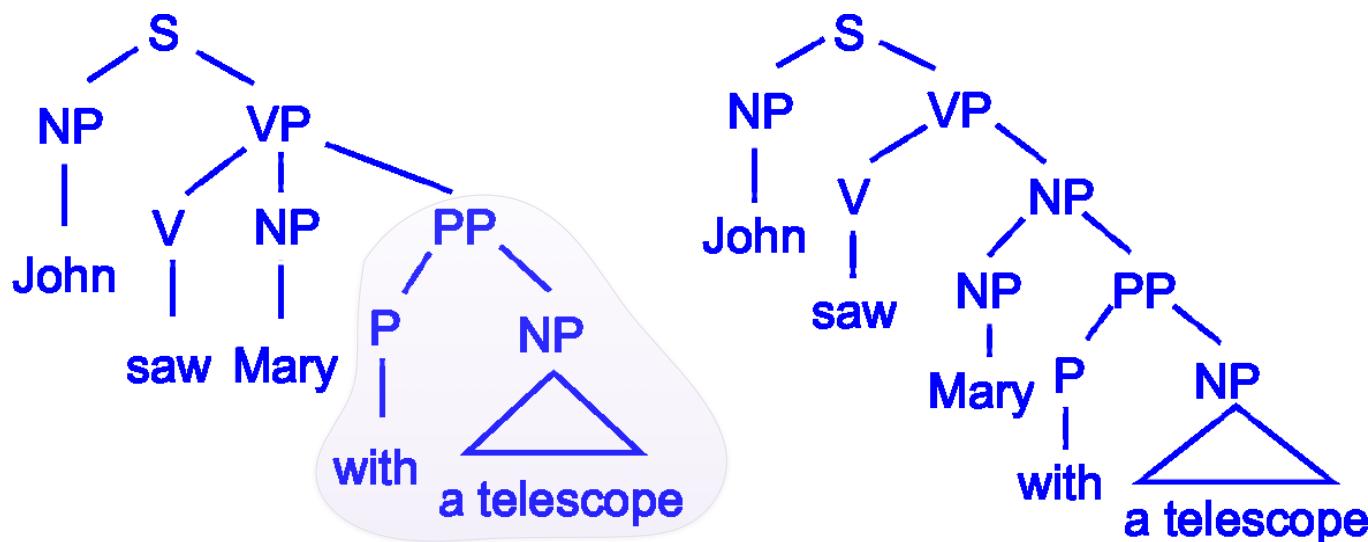
- Top Down Parsing
 - Simple, Memory efficient
 - Much repeated work, may loop infinitely
- Bottom Up Parsing
 - Less repeated work, harder to control
- Chart Parsing
 - Memory inefficient (especially with features)
 - No repeated work, difficult to control

Deterministic Parsing

- Motivation
 - People don't notice ambiguity . . .
 - But sometimes have trouble
 - “The horse raced past the barn fell”
“We painted all the walls with cracks”
“The man kept the dog in the house”
 - Can we do what the “human parser” does?

Heuristics

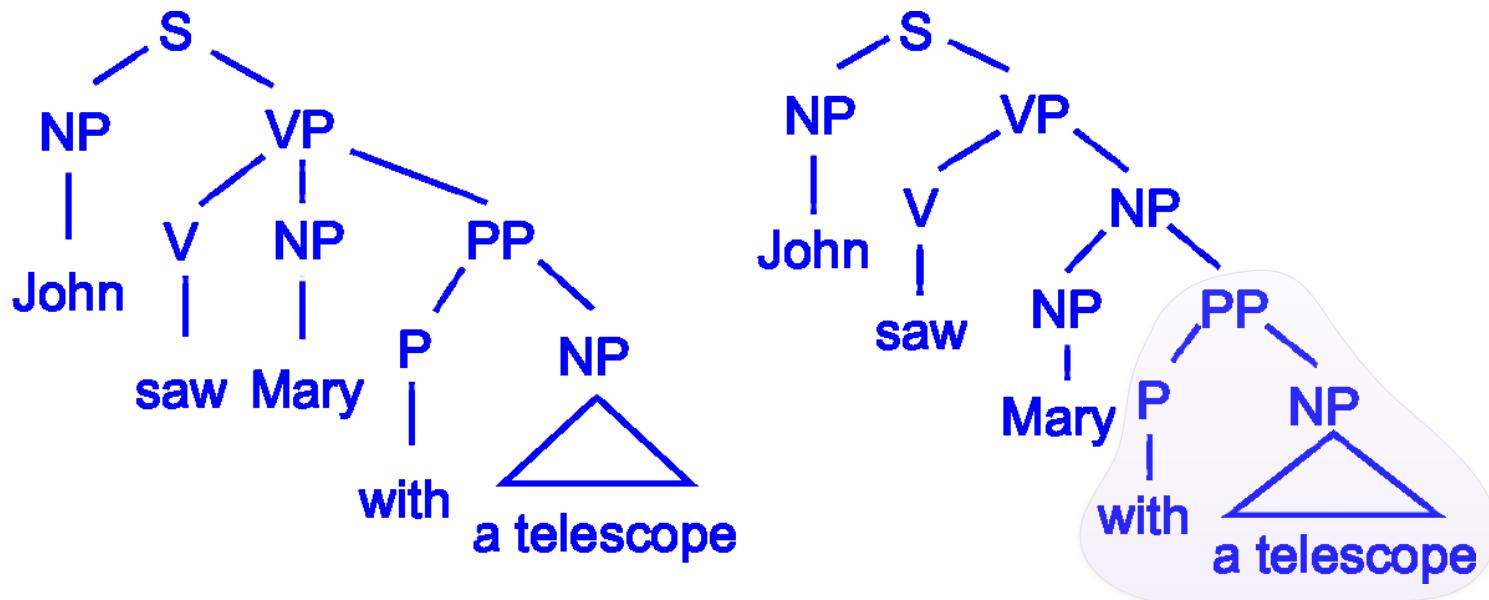
Minimal Attachment



Minimise size of parse tree

Heuristics

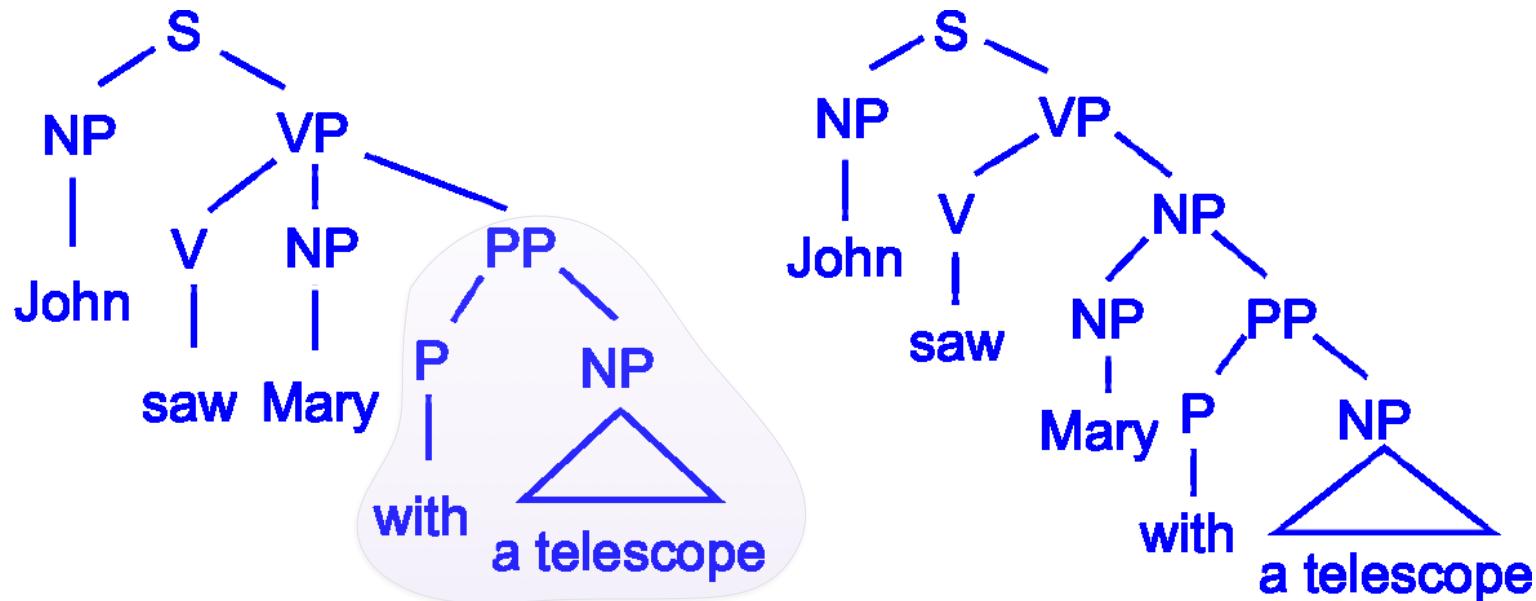
Right Association



Always attach to rightmost (lower) nodes

Heuristics

Lexical Preference



Try to fill most common sub-categorisation frame

Probabilistic Context Free Grammars

- Associate probabilities with grammar rules
 - Requires parsed corpus (e.g. Penn Treebank)
 - Count number of times rule used in parsing corpus sentences
- Probability of parse tree
 - \prod_r probability rule $r \times \prod_w$ probability of word w given category
 - Assuming independence

Probabilistic Chart Parsing

- Start with probabilities calculated by part of speech tagger
- Multiply probabilities when applying fundamental rule
- Best-First Chart Parsing
 - Examine most likely constituents first (priority queue)
 - Never constructs constituents with lower probability than parse

Summary

- Syntactic Knowledge
 - Grammatical categories defined by distribution
 - Much determined by properties of lexical items
- Context Free Grammars
 - Useful and powerful formalism
 - Relatively efficient parsers
 - Limited when dealing with complex phenomena
- Parsing
 - Top down method is easy to understand, but not efficient
 - Bottom up method is more efficient

COMP3411: Artificial Intelligence

Semantics and Pragmatics



This Lecture

- Semantics
 - ▶ Features and Augmented Grammars
 - ▶ Semantic Interpretation

- Pragmatics
 - ▶ Discourse Structure
 - ▶ Speech Act Theory
 - ▶ Dialogue Management



Agreement

- Number agreement
 - ▶ Which country borders France?
 - ▶ Which countries border France?
 - ▶ *Which country border France?
 - ▶ *Which countries borders France?
- Case
 - ▶ I saw him
 - ▶ *Him saw I
- To capture these facts, need lexical knowledge



Noun Features

Pronoun	Person	Number	Gender	Case
I	first	sing		nom
you	second	sing/plural		nom/acc
we	first	plural		nom
us	first	plural		acc
he	third	sing	masculine	nom
she	third	sing	feminine	nom
it	third	sing	neuter	nom/acc
him	third	sing	masculine	acc
her	third	sing	feminine	acc
they	third	plural		nom
them	third	plural		acc

Nominative: refers to subject
Accusative: refers to object



Verb Forms

Verb	Form	Example
cry	base	
cries	simple present	He cries
cried	simple past	He cried
crying	present participle	He is crying
cried	past participle	He has cried

Tense	Verb sequence	Example
future	will + infinitive	He will cry
present perfect	has + past participle	He has cried
future perfect	will + have + past participle	He will have cried
past perfect	had + past participle	He had cried



Verb Forms

Progressive	Verb sequence	Example
present	is + present participle	He is crying
past	was + present participle	He was crying
future	will + be + past participle	He will be crying
present perfect	has + been + pres participle	He has been crying
future perfect	will + have + been + past participle	He will have been crying
past perfect	had + been + pres participle	He had been crying



Subcategorization

[]	Jack laughed
[NP]	Jack found a key
[NP, NP]	Jack gave Sue the paper
[VP(inf)]	Jack wants to fly
[NP, VP(inf)]	Jack told the man to go
[VP(ing)]	Jack keeps hoping for the best
[NP, VP(ing)]	Jack caught Sam looking at his desk
[NP, VP(base)]	Jack watched Sam look at his desk

- Verb form determines mandatory sentence constituents
 - VP(inf) – infinitive is the base form of the verb
-



Augmented Context Free Grammars

- Each symbol has a collection of features
- Grammar rules constrain feature values
 - ▶ Use unification to enforce constraints, as in Prolog
- Features (mainly) derived from lexical items
- Some also from grammar rules (e.g. Case)
- Simple example
 - ▶ $S(\text{number: N}) \rightarrow NP(\text{number: N}) VP(\text{number: N})$
 - ▶ Enforce number agreement by unification (matching)



Typical (Small) Grammar

$S(Agr) \rightarrow NP(Agr), VP(VForm, Agr)$

$NP(Agr) \rightarrow ART(Agr), N(Agr)$

$NP(Agr) \rightarrow PRO(Agr)$

$VP(VForm, Agr) \rightarrow V(VForm, Agr, []) \quad \# \text{ subcat feature}$

$VP(VForm, Agr) \rightarrow V(VForm, Agr, [NP]), NP(Agr)$

$VP(VForm, Agr) \rightarrow V(VForm, Agr, [VP(inf)]), VP(inf, _)$

$VP(VForm, Agr) \rightarrow V(VForm, Agr, [ADJP]), ADJP$

$VP(inf, Agr) \rightarrow to, VP(base, Agr)$

$ADJP \rightarrow ADJ([])$

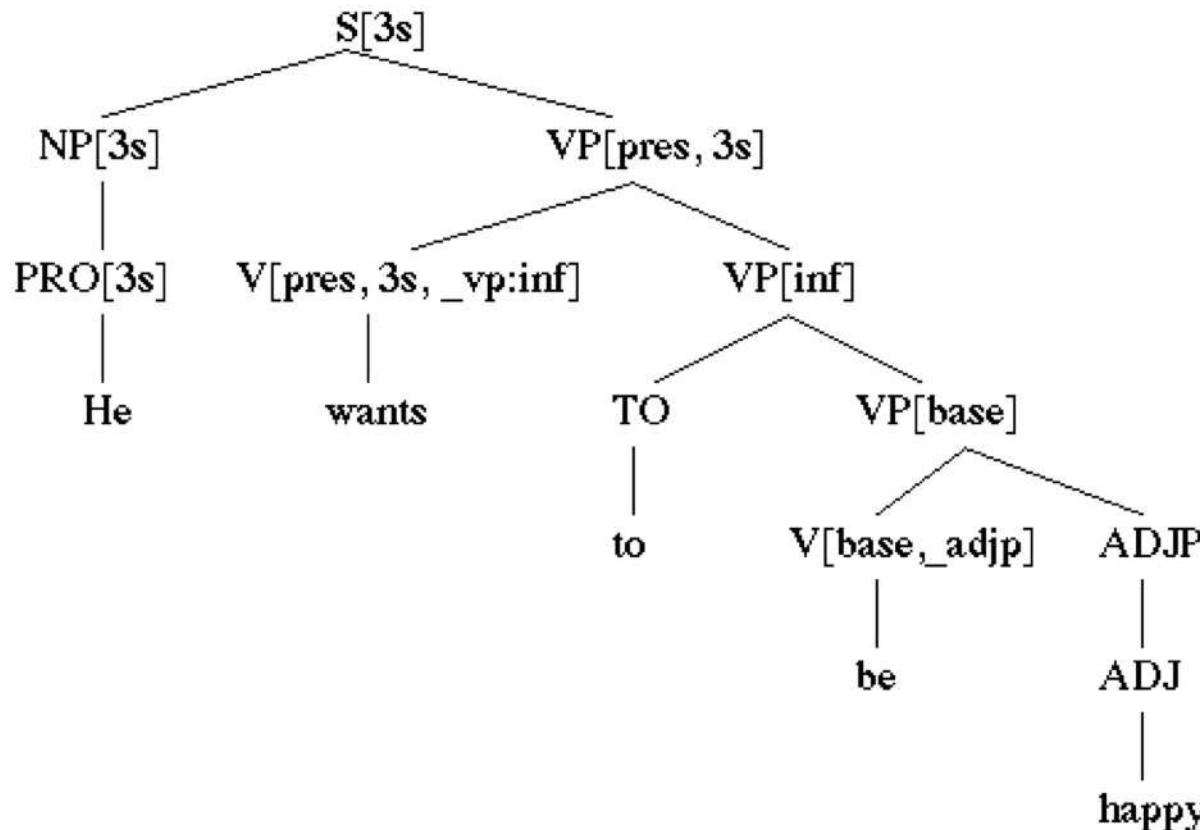
$ADJP \rightarrow ADJ([VP(inf)]), VP(inf, _)$

Convention is to unify (match) arguments in rule with same name

Note: $_$ is a variable that matches anything



Example



Note: Not all arguments are specified on tree nodes



Semantic Interpretation

- Logical form (LF) captures underlying “meaning”
 - ▶ Depends on purpose – no one “true” meaning
- Logical form should resolve semantic ambiguity
- Compute LF of sentence from LF of constituents
- Treat logical form as another feature
- Example: John sold a car to Mary
$$\text{event}(e, \text{Sell}) \wedge \text{occur}(e, \text{past}) \wedge \text{agent}(e, \text{John}) \wedge \text{co-agent}(e, \text{Mary}) \\ \wedge \text{object}(e, \{c: \text{car}\})$$



Thematic Roles

- Agent (intentional actor)
- Object/Theme (object on which action performed)
- Patient (animate object affected psychologically)
- Co-agent
- Instrument
- Beneficiary
- Location
- Source
- Destination

Often hard to distinguish!



Assigning Thematic Roles

- (Semantic) **selection restrictions** given by verb
 - ▶ e.g. agent of ‘break’ is animate
- Prepositions indicate likely role
 - ▶ e.g. ‘with’ implies instrument or co-agent
 - ▶ e.g. ‘by’ implies location or agent
- Problem examples
 - ▶ The window broke
 - ▶ My car drinks petrol

Simple method but doesn’t always work ⇒ probabilities



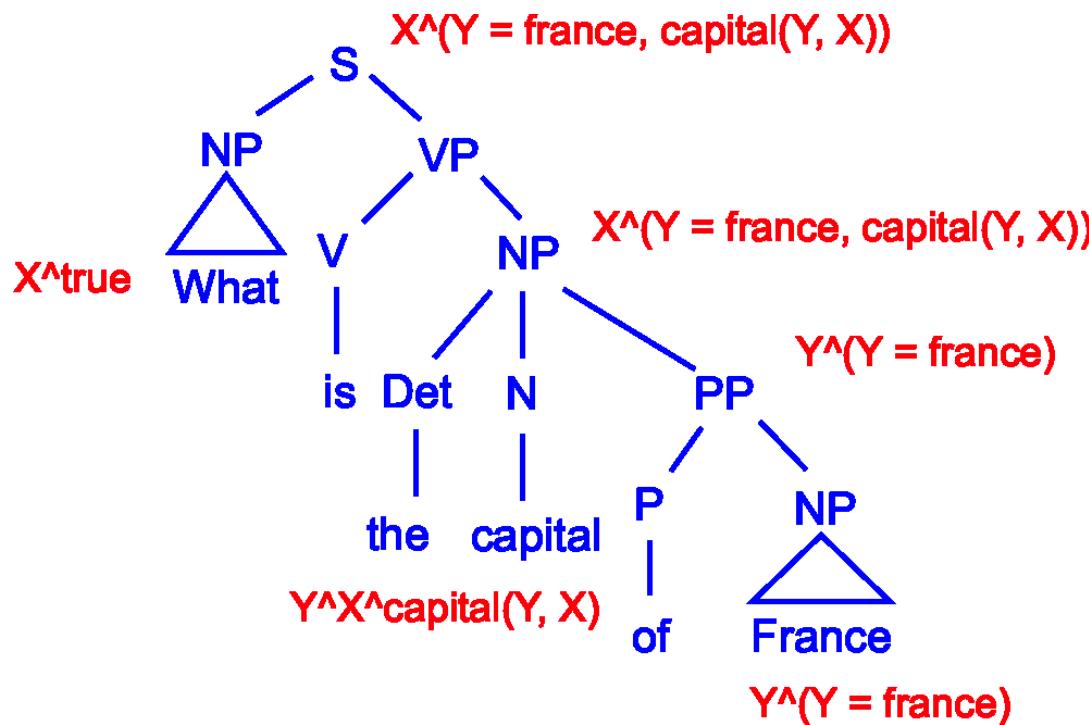
Logical Form (Chat-80)

- Logical Form (LF) is just another feature
 - ▶ Formulae of the form X^F where X is a variable and F is a formula
 - ▶ Read “the X such that F”
- May need more than matching to compute logical forms



Example Logical Form

What is the capital of France?



Example Grammar

$X^*(Y = \text{france} \wedge \text{capital}(Y, X))$

$S(X^*(NPForm \wedge VPForm)) \rightarrow NP(X^*NPForm), VP(X^*VPForm)$

$NP(Form) \rightarrow N(Form)$ the capital (of France)

$NP(X^*(PPForm \wedge NForm)) \rightarrow ART, N(Y^*X^*NForm), PP(Y^*PPForm)$

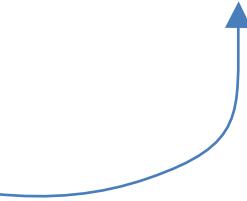
$N(X^*true) \rightarrow \text{what}$

$N(X^*(X = \text{france})) \rightarrow \text{france}$

$N(X^*Y^*(\text{capital}(X, Y))) \rightarrow \text{capital}$

$VP(Form) \rightarrow V(\text{be}), NP(Form)$

$PP(Form) \rightarrow P(\text{of}), NP(Form)$



Summary

- Disambiguation is central problem in NLP
- Use logical form language to resolve semantic ambiguity
- Augmented grammars can capture agreement and logical form
 - ▶ Focus on lexical knowledge
- No one “right” logical form language
 - ▶ Case frames, First-order logic, ...



Pragmatics

- Discourse Processing
 - ▶ Reference Resolution
 - ▶ Discourse Structure
- Speech as Rational Action
 - ▶ Speech Act Theory
 - ▶ Spoken Dialogue Systems



Reference Resolution

Jack lost his wallet in his car.

He looked for it for several hours.

Jack forgot his wallet.

Sam did too.

Jack forgot his wallet.

He looked for someone to borrow money from.

Sam did too.

I found a red pen and a pencil.

The pen didn't work.

I saw two bears.

Bill saw some too.



Reference Resolution

We bought a desk.

The drawer was broken.

Reserve a flight to Brisbane for me.

Reserve one for Norman too.

Mary gave John five dollars.

It was more than she gave Sue. One of them was counterfeit.

Each person took a handout.

Then she threw it away.

John didn't marry a Swedish blonde.

She was Danish/She had brown hair/She's living with him.



Discourse Entities

- The possible antecedents of pronouns
 - ▶ Noun phrases explicitly mentioned in recent discourse
 - ▶ A set of (implied) discourse entities
 - e.g. the handouts each person took, the set of people
 - ▶ An object related to (evoked by) a discourse entity
 - e.g. the drawer of the desk
 - ▶ Fillers of roles in stereotypical scenarios
 - e.g. waiters in restaurants
- Assume discourse is divided into “local contexts”



Focus Hypothesis

- At any time, there is a discourse entity that is the preferred antecedent for pronouns in the current local context – the discourse **focus**
 1. If any object in the local context is referred to by a pronoun in the current sentence, then the focus of the current sentence must also be pronominalized
 2. The focus is the most preferred discourse entity in the local context that is referred to by a pronoun
 3. Maintaining the focus is preferred to changing the focus
- Order possible antecedents subject > object > rest

Example

Jack left for the party late. (focus = Jack)

When **he** arrived, Sam met **him** at the door. (focus = **he/Jack**)

He decided to leave early. (focus = **he/Jack**)

Jack saw **him** in the park. (focus = **him**)

He was riding a bike. (focus = **he/him**)

While Jack was walking in the park, **he** met Sam. (focus = **he/Jack**)

He invited **him** to the party. (focus = Jack or Sam)

Discourse Structure

- E: So you have the engine assembly finished.
Now attach the rope to the top of the engine.
By the way, did you buy petrol today?
- A: Yes. I got some when I bought the new lawnmower wheel.
I forgot to take my can with me, so I bought a new one.
- E: Did **it** cost much?
- A: No, and I could use another anyway.
- E: OK. Have you got **it** attached yet?

Tracking focus isn't enough

Discourse Segments

- Recency-based technique for reference resolution
- Fixed time and location or simple progression
- Fixed set of speakers/hearers
- Fixed set of background assumptions
- Intentional view
 - ▶ Segment elements contribute to same **discourse purpose**
- Informational view
 - ▶ Segment elements are related temporally, causally, etc.

Hierarchical Structure

SEG1

Jack and Sue went to buy a new lawnmower since their old one was stolen.

SEG2

Sue had seen the man who took it and she had chased them down the street, but they'd driven away in a truck.

After looking in the store, they realized they couldn't afford one.

SEG3

By the way, Jack lost his job last month so he's been short of cash recently. He has been looking for a new one, but so far hasn't had any luck.

Anyway, they finally found a used one at a garage sale.

Attentional Stack

- Stack corresponding to segment hierarchy at point in time
 - ▶ e.g. [SEG1, SEG2] or [SEG1, SEG3]
- Stack update on starting SEG3
 - ▶ Either push new segment
 - giving [SEG1, SEG2, SEG3]
 - ▶ Or close current segment and push new segment
 - giving [SEG1, SEG3]

Managing the Attentional Stack

- Extending a segment
 - ▶ All references can be resolved in current segment
 - ▶ Same tense or same tense without perfect aspect
- Creating a new segment
 - ▶ Change in tense (progression of discourse)
 - ▶ Cue phrase indicating digression
- Closing a segment
 - ▶ Discourse purpose of new segment part of immediate parent

Speech Acts

- Speech as goal-directed rational activity
 - ▶ e.g. promise, threaten, warn, order, advise, state request, inform, assert, deny, apologize, thank, greet, criticize, dare, hope, congratulate, welcome, bless, curse, toast, challenge, announce, declare, question, . . .
- Sometimes explicit in utterances, use of so-called **performative** verbs

Speech act type is utterance's illocutionary* force

* Communicative effect

Actions Involved in Speech Acts

- Locutionary act
 - ▶ Physical act of saying something
- Illocutionary act
 - ▶ Speech act performed in making utterance
- Perlocutionary act
 - ▶ Effect on hearers actions thoughts, beliefs, etc.

Communication involves intention recognition

Characterising Illocutionary Acts

	request	warn
propositional content	Future act A of H	Future event or state E
preparatory conditions	H able to do A S believes H able to do A Not obvious to both S, H that H will do A anyway	S has reason to believe E will occur and is not in H's interest Not obvious to both S, H that E will occur
sincerity conditions	S wants H to do A	S believes E not in H's best interest
essential conditions	Counts as an attempt to get H to do A	Counts as undertaking to the effect that E not in H's best interest

Heavily oriented towards speaker's intentions

Indirect Speech Acts

- Use of one kind of illocutionary act to perform another
 - ▶ Can you pass the salt?
 - ▶ Do you know the time?
 - ▶ You are standing on my foot
 - ▶ Why don't you leave now?
 - ▶ Would you like a game of tennis?
 - ▶ If I were you, I'd sell that car
 - ▶ Now would you mind getting off my foot!
- Even harder to recognize speaker's intention

Spoken Dialogue Systems

- Feasible now with good speech recognition
 - ▶ Speaker dependent or domain specific
- Based on limiting possible dialogue structure
 - ▶ Frames with slots that need filling
 - ▶ Graphs representing possible transitions
 - ▶ Rules for defining actions based on prior context
 - ▶ Limited range of subdialogues (e.g. clarification)

Aim is to perform as little reasoning as possible

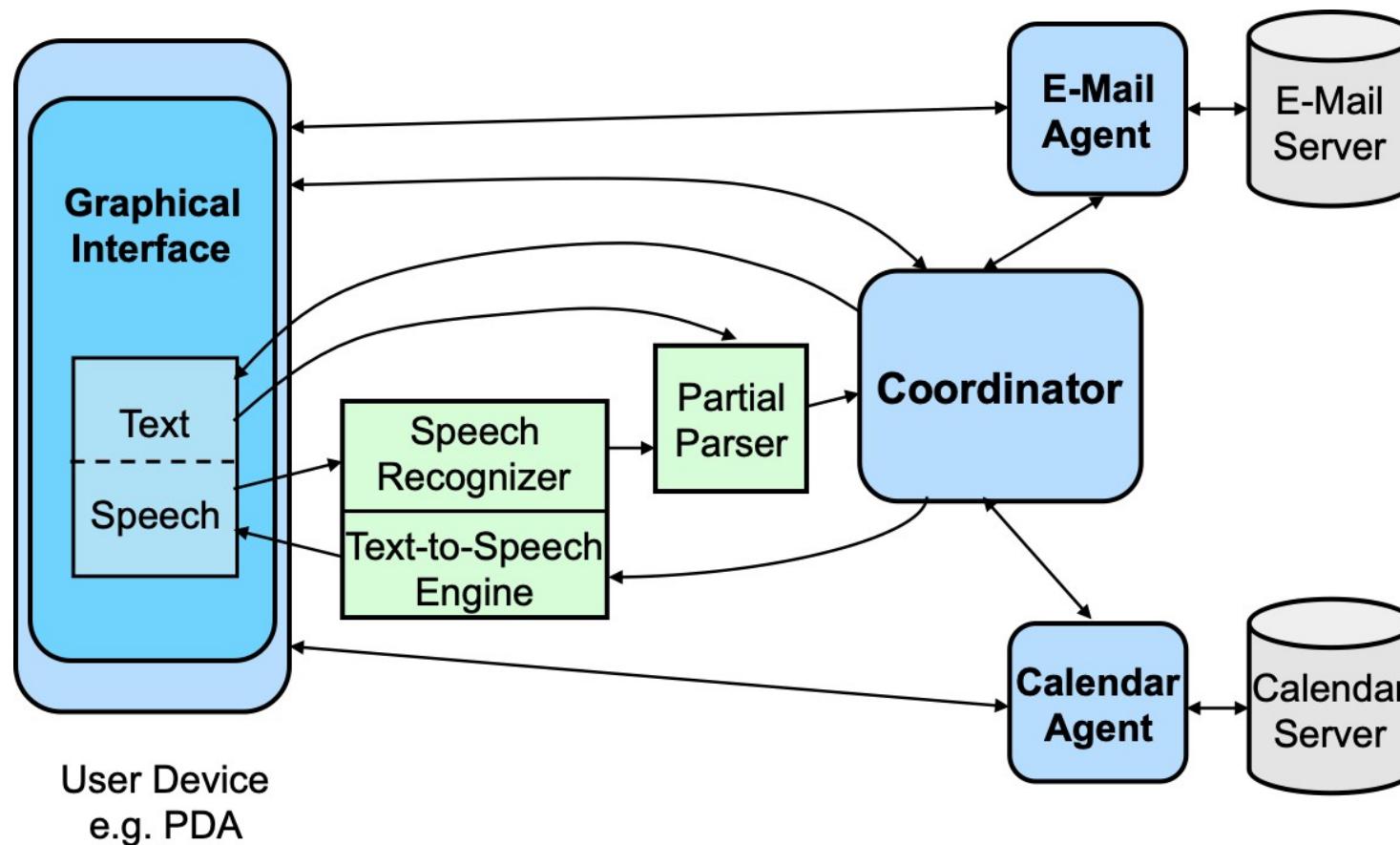
Initiative

- System Initiative
 - ▶ System “controls” dialogue by prompting user for information
 - ▶ Useful for specific tasks
 - Booking flights, Ordering pizza, Placing bets
- User Initiative
 - ▶ User “controls” dialogue by questions, commands
 - ▶ Useful for simple tasks, e.g. web search, training and simulation
- Mixed Initiative
 - ▶ Mixture of system initiative and user initiative

Smart Personal Assistant

- Integrated collection of personal (task) assistants
- Each assistant specializes in a task domain
 - ▶ E-mail and calendar management
- Users interact through a range of devices
 - ▶ PDAs, desktops, iPhone?
- Focus on usability
 - ▶ Multi-modal natural language dialogue
 - ▶ Adapt to users device, context, preferences

System Architecture



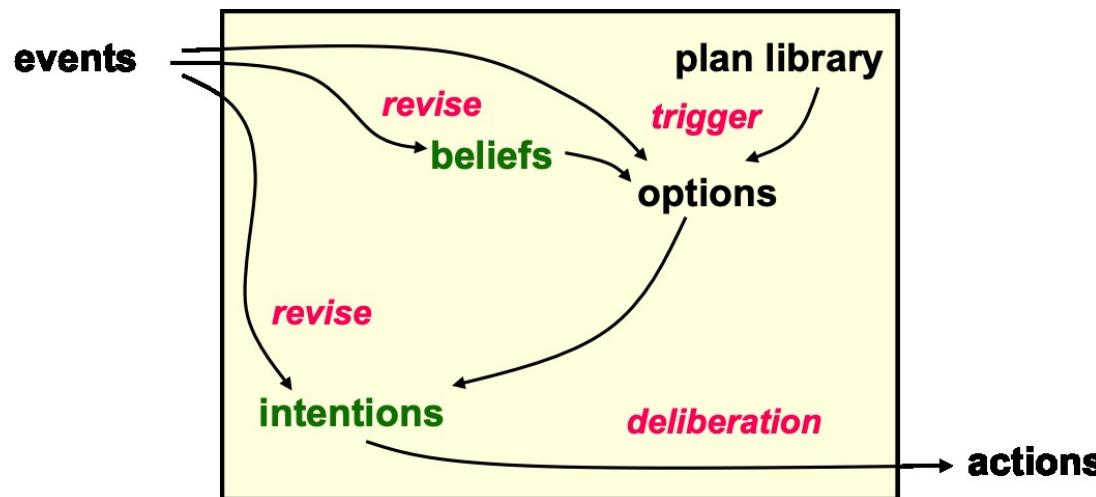
Partial Parsing

- Full parsing is inappropriate
 - ▶ Limited accuracy of speech recognition
 - ▶ Regular use of short-form expressions
 - ▶ Unconstrained language vocabulary
 - e.g. “Are there any new messages from . . . ”
- Shallow syntactic frame

clause	
connective	Expresses the relation of the clauses
type	Question, declaration, imperative, ...
subject	Syntactic subject
predicate	Main verb
direct object	Main object of the predicate
indirect object	Possible second object
complement phrase	Other information e.g. time, location

BDI Agent Architecture

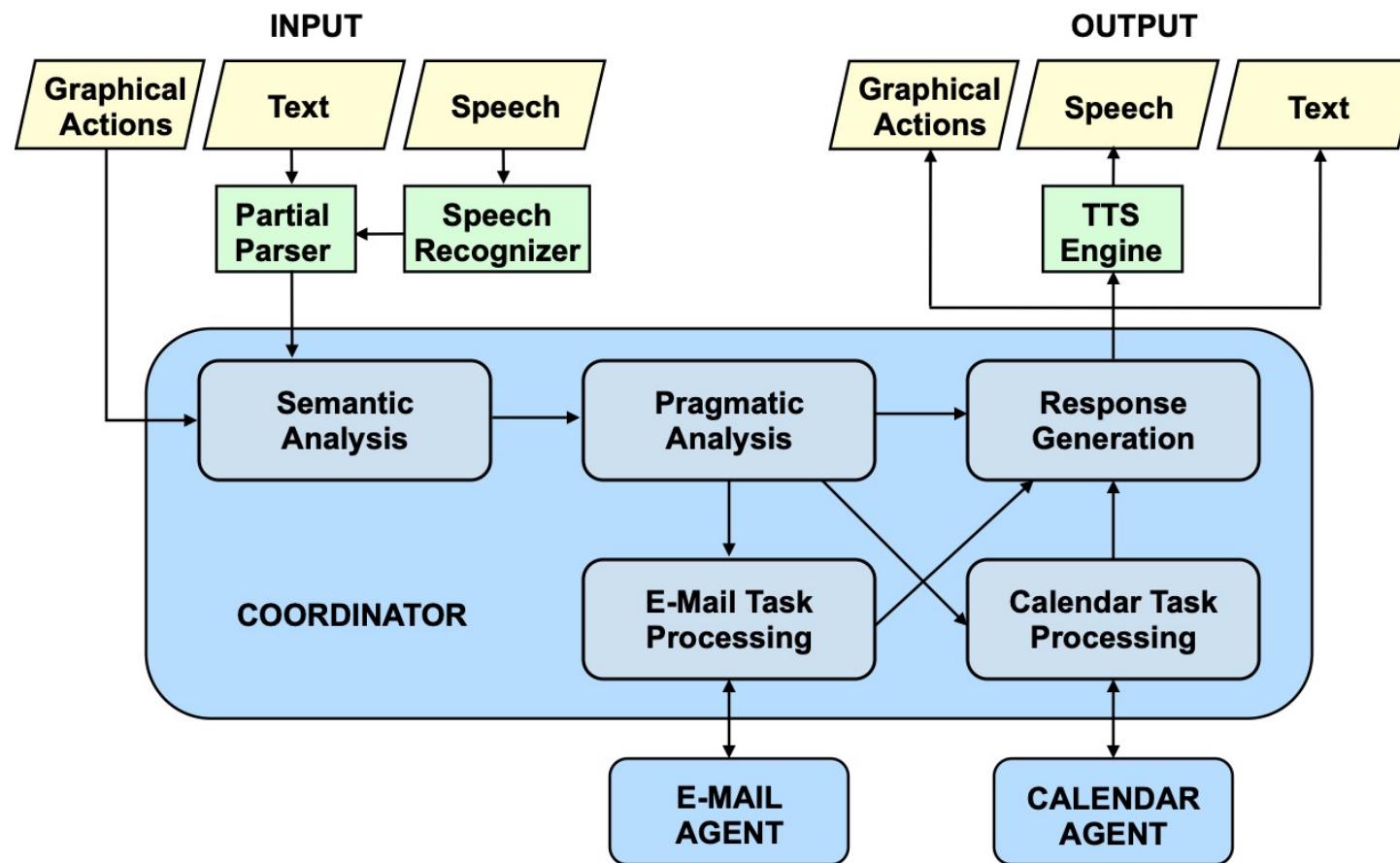
- Beliefs, desires, intentions explicit
 - ▶ Pre-defined plans for achieving goals
- Interpreter cycle PRS (Procedural Reasoning System)
 - ▶ Event-driven selection and execution of plans



Dialogue Management Beliefs

- Dialogue model
 - ▶ Discourse history (stack of conversational acts)
 - ▶ Salient list (ranked list of recently mentioned objects)
- Domain knowledge
 - ▶ Supported tasks (for each task assistant)
 - ▶ Domain-specific vocabularies for task interpretation
- User model
 - ▶ User context information (device, modalities, . . .)

Dialogue Management Plans



Summary

- Dialogue systems are current “hot” topic
- Feasible because of high-quality speech recognition
 - ▶ Questions over accuracy, usability of Siri, Alexa, etc.
- Industry impetus is automation of routine interactions to reduce costs
 - ▶ Lot of hype compared to actual deployed systems
- Simple dialogue management techniques include graphs of dialogue actions and rules based on dialogue context
 - ▶ Current possible interactions not very sophisticated

Definite Clause Grammars

COMP3411/COMP9814 - Artificial Intelligence

Logic Grammars

- A grammar rule is a formal device for defining sets of sequences of symbols.
- Sequence may represent a statement in a programming language.
- Sequence may be a sentence in a natural language such as English.

BNF Notation

- A BNF grammar specification consists of *production rules*.

$\langle s \rangle ::= a b$

$\langle s \rangle ::= a \langle s \rangle b$

- First rule says that whenever s appears in a string, it can be rewritten with the sequence ab .
- Second rule says that s can be rewritten with a followed by s followed by b .

BNF Notation

- s is a non-terminal symbol.
- a and b are terminal symbols.
- A grammar rule can generate a string, e.g.

$s \rightarrow a s b$

$a s b \rightarrow a a s b b$

$a a s b b \rightarrow a a a b b b$

Grammar for a robot arm

- Two commands for a robot arm are: *up* and *down*, i.e. move one step up or down respectively.

`<move> ::= <step>`

`<move> ::= <step> <move>`

`<step> ::= up`

`<step> ::= down`

- The grammar is recursive and has a termination rule.

Definite Clause Grammars

- Prolog has a DCG grammar notation that parses sequences of symbols in a list.

```
s --> [a], [b].
```

```
move --> step.
```

```
s --> [a], s, [b].
```

```
move --> step, move.
```

```
?- s([a, a, b, b], X).
```

```
step --> [up].
```

```
X = []
```

```
step --> [down].
```

```
?- s([a, c, b], X).
```

```
false.
```

DCGs are translated into Prolog

```
s --> [a], [b].  
s --> [a], s, [b].
```

[a, a, b, b]	(2)
[a, b, b]	(2)
[b]	(1)
[]	

```
move --> step.  
move --> step, move.  
  
step --> [up].  
step --> [down].
```



```
s([a, b|X], X).  
s([a|X], Y) :-  
    s(X, [b|Y]).
```

```
move(X, Y) :-  
    step(X, Y).
```

```
move(X, Z) :-  
    step(X, Y),  
    move(Y, Z).
```

```
step([up|X], X).  
step([down|X], X).
```

A simple subset of English

sentence --> noun_phrase, verb_phrase.

noun_phrase --> determiner, noun.

verb_phrase --> verb, noun_phrase.

determiner --> [a].

determiner --> [the].

noun --> [cat].

noun --> [mouse].

verb --> [scares].

verb --> [hates].

E.g.

the cat scares the mouse

the mouse hates the cat

the mouse scares the mouse

Context Dependence

- Programming languages usually use context free grammars.
 - Type of symbol is determined completely by its position in sentence.
- Natural language is often context dependent.
 - Correctness of one symbol depends on type of other symbols in sentence.
 - E.g. number in English.

Context Dependence

noun --> [cats].

verb --> [hate].

- Adding these rules to the grammar makes the following sentence legal:

the mouse hate the cat.

- Additional constraints must be added to the grammar to ensure that the number of all the parts of speech is consistent.

Context Dependence

```
sentence -->
    noun_phrase(Number),
    verb_phrase(Number).

noun_phrase(Number) -->
    determiner(Number),
    noun(Number).

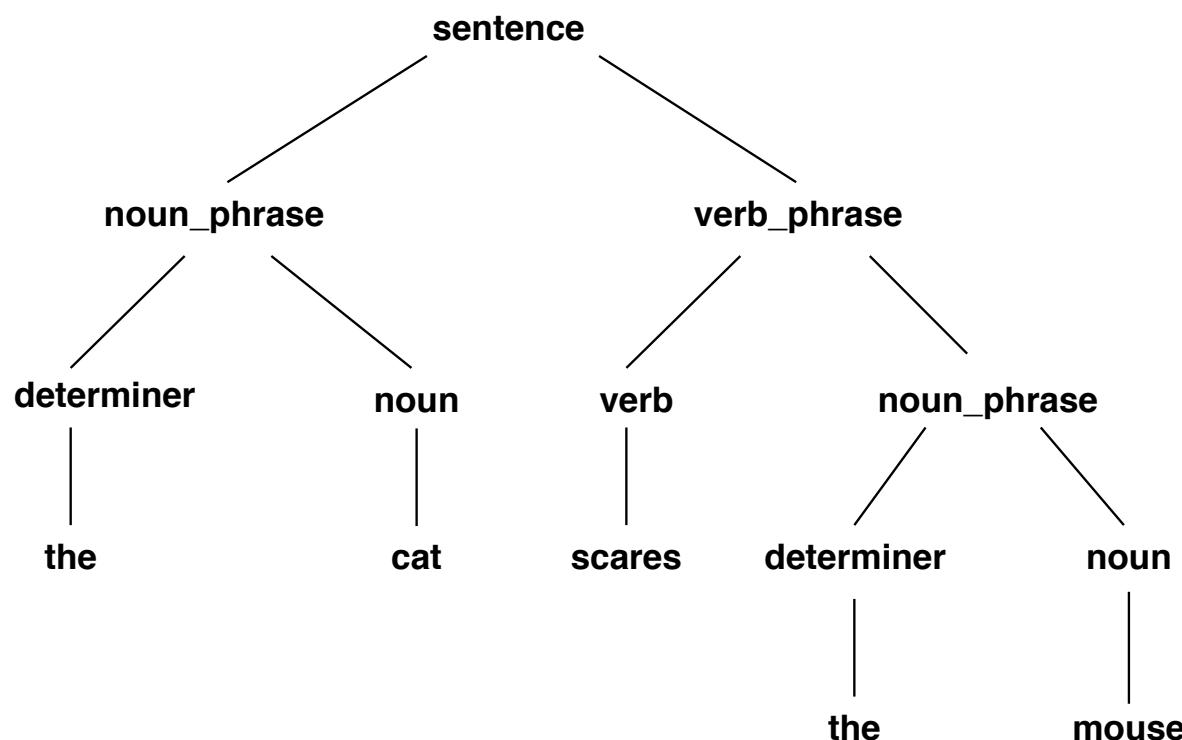
verb_phrase(Number) -->
    verb(Number),
    noun_phrase(_).

determiner(singular) --> [a].
determiner(_) --> [the].

noun(singular) --> [cat].
noun(singular) --> [mouse].
noun(plural) --> [mice].

verb(singular) --> [hates].
verb(plural) --> [hate].
```

Parse Trees



Parse Trees

- Leaves are labelled by the terminal symbols of the grammar
- Internal nodes are labelled by non-terminals
- The parent-child relation is specified by the rules of the grammar.

Parse Trees

```
sentence(sentence(NP, VP)) -->  
    noun_phrase(Number, NP),  
    verb_phrase(Number, VP).
```

```
noun_phrase(Number, noun_phrase(Det, Noun)) -->  
    determiner(Number, Det),  
    noun(Number, Noun).
```

```
verb_phrase(Number, verb_phrase(V, NP)) -->  
    verb(Number, V),  
    noun_phrase(_, NP).
```

```
determiner(singular, determiner(a)) --> [a].  
determiner( _, determiner(the)) --> [the].
```

```
noun(singular, noun(cat)) --> [cat].  
noun(plural, noun(cats)) --> [cats].  
noun(singular, noun(mouse)) --> [mouse].  
noun(plural, noun(mice)) --> [mice].
```

```
verb(singular, verb(scares)) --> [scares].  
verb(singular, verb(hates)) --> [hates].  
verb(plural, verb(hate)) --> [hate].
```

DCG Translation

```
sentence(sentence(NP, VP) -->  
        noun_phrase(Number, NP),  
        verb_phrase(Number, VP)).
```



```
Input list           Remainder  
sentence(sentence(A, D), B, F) :-  
    noun_phrase(C, A, B, E),  
    verb_phrase(C, D, E, F).
```

From parse tree to meaning

```
?- sentence(X, [the, mouse, hates, the, cat], Y).
```

```
X = sentence(  
            noun_phrase(determiner(the), noun(mouse)),  
            verb_phrase(verb(hates),  
                        noun_phrase(determiner(the), noun(cat))))
```

```
Y = []
```

- In a two step understanding system, the parse tree returned from the grammar rules could be passed to a semantic analyser.

Defining the meaning of a sentence

```
sentence(VP) -->
    noun_phrase(Actor),
    verb_phrase(Actor, VP).

noun_phrase(NP) -->
    proper_noun(NP).

verb_phrase(Actor, VP) -->
    intrans_verb(Actor, VP).
verb_phrase(Subject, VP) -->
    trans_verb(Subject, Object, VP),
    noun_phrase(Object).

intrans_verb(Actor, paints(Actor)) --> [paints].
trans_verb(Subject, Object, likes(Subject, Object)) --> [likes].

proper_noun(john) --> [john].
proper_noun(annie) --> [annie].
```

Defining the meaning of a sentence

```
?- sentence(X, [john, paints], Y).
```

```
C|>sentence(_0)
C||>noun_phrase(_1)
C|||>proper_noun(_1)
E|||<proper_noun(john)
E||<noun_phrase(john)
C||>verb_phrase(john, _0)
C|||>intrans_verb(john, _0)
E|||<intrans_verb(john, paints(john))
E||<verb_phrase(john, paints(john))
E|<sentence(paints(john))
```

```
X = paints(john)
```

```
Y = []
```

Defining the meaning of a sentence

```
?- sentence(X, [john, likes, annie], _).  
  
C |>sentence(_0)  
C ||>noun_phrase(_1)  
C |||>proper_noun(_1)  
E |<proper_noun(john)  
E |<noun_phrase(john)  
C |>verb_phrase(john, _0)  
C ||>intrans_verb(john, _0)  
R |>verb_phrase(john, _0)  
C ||>trans_verb(john, _9, _0)  
E |<trans_verb(john, _9, likes(john, _9))  
C |>noun_phrase(_9)  
C ||>proper_noun(john)  
R |>proper_noun(annie)  
E |<proper_noun(annie)  
E |<noun_phrase(annie)  
E |<verb_phrase(john, likes(john, annie))  
E |<sentence(likes(john, annie))  
  
X = likes(john, annie)
```

The Determiner 'a'

- 'A person paints' does *not* mean *paints(person)*.
- In this sentence 'person' is not a specific person. The correct meaning should be:
 $\text{exists}(X, \text{person}(X) \ \& \ \text{paints}(X))$
- The general form for dealing with 'a'
 $\text{exists}(X, \text{person}(X) \ \& \ \text{Assertion})$

The determiner 'every'

E.g.

Every student studies

$\text{all}(X, \text{student}(X) \rightarrow \text{studies}(X))$

'every' indicates the presence of a *universally quantified* variable.

Relative Clauses

E.g.

Every person that paints admires Monet

Can be expressed in a logical form as:

For all X, if X is a person and X paints then X admires Monet.

in Prolog:

`all(X, person(X) & paints(X) -> admires(X, monet))`

in general:

`all(X, Property1 & Property2 -> Assertion)`

The Complete Grammar

```
?- op(700, xfy, &).  
?- op(800, xfy, ->).
```

```
determiner(X, Property, Assertion, all(X, (Property -> Assertion))) --> [every].  
determiner(X, Property, Assertion, exists(X, (Property & Assertion))) --> [a].
```

```
noun(X, man(X)) --> [man].  
noun(X, woman(X)) --> [woman].  
noun(X, person(X)) --> [person].
```

```
proper_noun(john) --> [john].  
proper_noun(annie) --> [annie].  
proper_noun(monet) --> [monet].
```

```
trans_verb(X, Y, likes(X, Y)) --> [likes].  
trans_verb(X, Y, admires(X, Y)) --> [admires].
```

```
intrans_verb(X, paints(X)) --> [paints].
```

The Complete Grammar

```
sentence(S) -->
    noun_phrase(X, Assertion, S),
    verb_phrase(X, Assertion).

noun_phrase(X, Assertion, S) -->
    determiner(X, Property12, Assertion, S),
    noun(X, Property1),
    rel_clause(X, Property1, Property12).
noun_phrase(X, Assertion, Assertion) -->
    proper_noun(X).

verb_phrase(X, Assertion) -->
    trans_verb(X, Y, Assertion1),
    noun_phrase(Y, Assertion1, Assertion).
verb_phrase(X, Assertion) -->
    intrans_verb(X, Assertion).

rel_clause(X, Property1, (Property1 & Property2)) -->
    [that],
    verb_phrase(X, Property2).
rel_clause(_, Property, Property).
```

The Complete Grammar

```
sentence(S) -->
    noun_phrase(X, Assertion, S),
    verb_phrase(X, Assertion).

noun_phrase(X, Assertion, S) -->
    determiner(X, Property12, Assertion, S),
    noun(X, Property1),
    rel_clause(X, Property1, Property12).

noun_phrase(X, Assertion, Assertion) -->
    proper_noun(X).

verb_phrase(X, Assertion) -->
    trans_verb(X, Y, Assertion1),
    noun_phrase(Y, Assertion1, Assertion).

verb_phrase(X, Assertion) -->
    intrans_verb(X, Assertion).

rel_clause(X, Property1, (Property1 & Property2)) -->
    [that],
    verb_phrase(X, Property2).

rel_clause(_, Property, Property).
```

```
determiner(X, Property, Assertion, all(X, (Property -> Assertion))) --> [every].
```

The Complete Grammar

```
sentence(S) -->
    noun_phrase(X, Assertion, S),
    verb_phrase(X, Assertion).

noun_phrase(X, Assertion, S) -->
    determiner(X, Property12, Assertion, S),
    noun(X, Property1),
    rel_clause(X, Property1, Property12).
noun_phrase(X, Assertion, Assertion) -->
    proper_noun(X).

verb_phrase(X, Assertion) -->
    trans_verb(X, Y, Assertion1),
    noun_phrase(Y, Assertion1, Assertion).
verb_phrase(X, Assertion) -->
    intrans_verb(X, Assertion).

rel_clause(X, Property1, (Property1 & Property2)) -->
    [that],
    verb_phrase(X, Property2).
rel_clause(_, Property, Property).
```

Variable Bindings

```
noun_phrase(X, Assertion, S) -->
    determiner(X, Property12, Assertion, S),
    noun(X, Property1),
    rel_clause(X, Property1, Property12).
```

```
determiner(X, Property, Assertion, all(X, (Property -> Assertion))) --> [every].
```

Result

```
?- sentence(X, [every, person, that, paints, admires, monet], _).  
X = all(_24512, person(_24512) & paints(_24512) -> admires(_24512, monet))
```

Annotations

```
noun_phrase(NP) --> proper_noun(NP), {asserta(history(NP))}.
```

- Annotations allow you to write any Prolog code you like to support the processing of the grammar.
- Anything in between '{' and '}'
- Asserta stores a clause in Prolog's data base.
 - The new clause becomes the *first* in the database

Assert and Asserta add clauses to the database

```
?- assert(f(a)).  
?- assert(f(b)).
```

```
?- listing(f/1).
```

```
f(a).  
f(b).
```

```
?- asserta(f(a)).  
?- asserta(f(b)).
```

```
?- listing(f/1).
```

```
f(b).  
f(a).
```

Retract deletes clauses

```
?- assert(f(a)).  
?- assert(f(b)).  
  
?- listing(f/1).  
  
f(a).  
f(b).
```

```
?- retract(f(X)).  
X = a ;  
X = b.  
  
?- listing(f/1).  
:- dynamic f/1.  
  
true.
```

Frames

- Objects are represented by a list of properties and values
- E.g.
 - object(shirt, [colour(green)]).
 - event(1, [
 - actor(john),
 - action(buy),
 - object(shirt, [colour(green)]),
 - location(myers)])
- “Understanding” a sentence means filling in the slots.

Resolving References

- Use Prolog annotations in grammar to build frame's property list.
- *new_event* asserts events in reverse order.
- Database events can be used to resolve references

Resolving References

Suppose history is:

```
history(object(John, [isa(person), gender(masculine), number(singular)])).  
history(object(annie, [isa(person), gender(feminine), number(singular)])).
```

Simple example of pronoun resolution:

```
pronoun(Resolvent) --> [he],  
    {resolve([gender(masculine), number(singular)], Resolvent)}.  
pronoun(Resolvent) --> [she],  
    {resolve([gender(feminine), number(singular)], Resolvent)}.
```

```
resolve(Properties, Name) :-  
    history(object(Name, Props)),  
    subset(Properties, Props).
```

Tutorials COMP3411/9814 20T1

Week 3 Solutions

Question 1 - The route-finding problem using the Romania map from Russell & Norvig

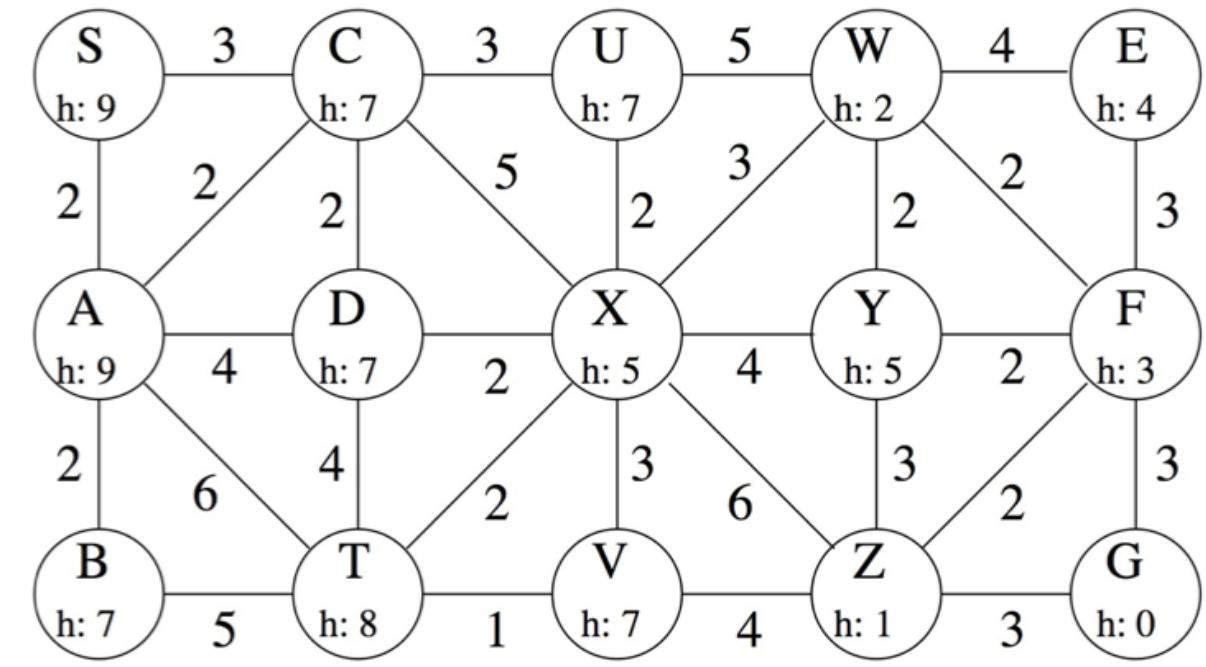
1. (Depth-first) Arad, Sibiu, Fagaras, Bucharest (note that the solution is found on the first branch only because of the rule for ordering the successors alphabetically; this is not usually the case!)
2. (Breadth-first) Arad, Sibiu, Timisoara, Zerind, Fagaras, Bucharest (assuming that the search stops once the goal state is generated and that when expanding a node, previously expanded nodes are checked to ensure that nodes with states already explored are not added to the frontier, e.g. Arad which is generated via the paths Arad → Sibiu → Arad and Arad → Sibiu → Oradea → Zerind → Arad)
3. (Uniform-cost) Arad (0), Zerind (75), Timisoara (118), Sibiu (140), Oradea (146), Rimnicu Vilcea (220), Lugoj (229), Fagaras (239), Mehadia (299), Pitesti (317), Craiova (366), Drobeta (374), Bucharest (418) (assuming a check that nodes with states previously generated are not added to the frontier, except when they have lower path cost than a node with that state already on the frontier, in which case the node with higher path cost is removed, so ignore Oradea (291) reached via Sibiu, and Sibiu (297) reached via Zerind and Oradea)
4. (Iterative deepening) Arad, Arad, Sibiu, Timisoara, Zerind, Arad, Sibiu, Fagaras, Oradea, Rimnicu Vilcea, Timisoara, Lugoj, Zerind, Oradea, Arad, Sibiu, Fagaras, Bucharest (assuming a cycle check on each path, so omit Arad reached via Arad → Sibiu → Arad)
5. (Greedy) Arad (366), Sibiu (253), Fagaras (178), Bucharest (0)
6. (A*) Arad (366), Sibiu (393), Rimnicu Vilcea (413), Pitesti (415), Fagaras (417), Bucharest (418) (unexpanded states on the frontier are Timisoara (447), Zerind (449), Craiova (526), Oradea (671)) – for example, $f(\text{Rimnicu Vilcea}) = g(\text{Rimnicu Vilcea}) + h(\text{Rimnicu Vilcea}) = 140 + 80 + 193 = 413$ (remember to use the total path cost from Arad to Rimnicu Vilcea here)

Question 2

Fagaras (415) is expanded before Pitesti (417)

Question 3 - Path Search Algorithms on a Graph

Consider the task of finding a path from start state S to goal state G, given the distances and heuristic values in this diagram:



For each of the following strategies, list the order in which the states are expanded. Whenever there is a choice of states, you should select the one that comes first in alphabetical order. In each case, you should skip any states that have previously been expanded, and you should continue the search until the goal node is expanded.

(a) Breadth First Search

S, A, C, B, D, T, U, X, V, W, Y, Z, E, F, G

(b) Depth First Search

S, A, B, T, D, C, U, W, E, F, G

(c) Uniform Cost Search [Hint: first compute $g()$ for each state in the graph]

$S(0)$, $A(2)$, $C(3)$, $B(4)$, $D(5)$, $U(6)$, $X(7)$, $T(8)$, $V(9)$, $W(10)$, $Y(11)$, $F(12)$, $Z(13)$, $E(14)$, $G(15)$

(d) Greedy Search, using the heuristic shown S, C, X, Z, G

(e) A*Search, using the heuristic shown

$S(9)$, $C(10)$, $A(11)$, $B(11)$, $D(12)$, $X(12)$, $W(12)$, $U(13)$, $Z(14)$, $F(15)$, $G(15)$

Note that $g(X)$ becomes 8 after C is expanded, but drops to 7 after D is expanded. Similarly, $g(G)$ becomes 16 when Z is expanded, but drops to 15 after F is expanded.

Question 4 - Relationships Between Search Strategies

Prove each of the following statements, or give a counterexample:

- (a) Breadth First Search is a special case of Uniform Cost Search
Uniform Cost Search reduces to Breadth First Search when all edges have the same cost.
- (b) Breadth First Search, Depth First Search and Uniform Cost Search are special cases of best-first search.
- (c) Best-first search reduces to Breadth-First Search when $f(n)$ =number of edges from start node to n , to UCS when $f(n)=g(n)$. It can be reduced to DFS by, for example, setting $f(n)$ =-(number of nodes from start state to n) (thus forcing deep nodes on the current branch to be searched before shallow nodes on other branches). Another way to produce DFS is to set $f(n)=-g(n)$ (but this might produce a particularly bad choice of nodes within the DFS framework - for example, try tracing the order in which nodes are expanded when traveling from Urziceni to Craiova).
- (d) Uniform Cost Search is a special case of A* Search
- (e) A* Search reduces to UCS when the heuristic function is zero everywhere, i.e. $h(n)=0$ for all n . This heuristic is clearly admissible since it always (grossly!) underestimates the distance remaining to reach the goal.

Question 5 - Heuristic Path Algorithm

The *heuristic path algorithm* is a best-first search in which the objective function is

$$f(n) = (2 - w) \cdot g(n) + w \cdot h(n), \text{ where } 0 \leq w \leq 2$$

What kind of search does this perform when $w=0$? when $w=1$? when $w=2$?

This algorithm reduces to Uniform Cost Search when $w = 0$, to A* Search when $w = 1$ and to Greedy Search when $w = 2$.

For what values of w is this algorithm complete? For what values of w is it optimal, assuming $h()$ is admissible?

*

It is guaranteed to be optimal when $0 \leq w \leq 1$, because it is equivalent to A Search using the heuristic:

$$f(n) = (2 - w) \cdot g(n) + w \cdot h(n)$$

When $w > 1$ it is not guaranteed to be optimal (however, it might work very well in practice, for some problems).

COMP3411 Tutorial- Week 4

Constraint Satisfaction

Question 1 - Cryptarithmetic

Cryptarithmetic is a type of mathematical puzzle where the numbers have been replaced with letters, or other symbols.

Solve the famous Cryptarithmetic problem and Provide not just the final answer, but also explain your reasoning along the way.

$$\begin{array}{r}
 \text{S} \quad \text{E} \quad \text{N} \quad \text{D} \\
 + \quad \text{M} \quad \text{O} \quad \text{R} \quad \text{E} \\
 \hline
 \text{M} \quad \text{O} \quad \text{N} \quad \text{E} \quad \text{Y}
 \end{array}$$

Variables:	Constraints:
D E M N O R S Y	$M \neq 0, S \neq 0$ (unary constraints)
Domains:	$Y = D + E$ or $Y = D + E - 10$, etc.
{0,1,2,3,4,5,6,7,8,9}	$D \neq E, D \neq M, D \neq N$, etc.

- a) Can you identify any backtracking heuristics or enhancements that you may have (unknowingly) used when you solved the problem?
- b) Are there any backtracking heuristics or enhancements that you would now use to solve the problem more efficiently?

What heuristics and strategies did you use along the way?

The sum of two 4-digit numbers cannot exceed 1998, so $M=1$.

$10+O = S+1$ or $S+1+1$, i.e. $S = O+9$ or $O+8$, but 1 has already been used, so $O=0$.

Therefore $S=9$, because there is no possibility of a carry from $E+O$.

We then have $E+1 = N$ and $10+E = N+R$ or $N+R+1$, so $R = 8$ or 9 , but 9 has already been assigned, so $R=8$.
(Note how Minimum Remaining Values has been used at each step.)

The puzzle now looks like this:

9END + 108E

10NEY

This gives us the two constraints: $E+1 = N$

$D+E = 10+Y$

The remaining values are 2,3,4,5,6,7.

We have $D+E \leq 6+7 = 13$, so $Y = 2$ or 3 . (Note: MRV again).

But if $Y=3$ (Most Constraining Value) then all three variables D,E,N would need to take values 6 or 7, which is impossible (Constraint Propagation).

Therefore $Y=2$, $E=5$, $N=6$ and $D=7$.

Question 2 - Map Colouring



(Refer to lectures for week 3)

Use Forward Checking to show that the Australia map-colouring problem has no solution when we assign WA=green, V=Red, NT=Red. If we apply Arc Consistency as well, can the inevitable failure be detected further up the tree?

Present your answer to this question and discuss with others in the tutorial group.

Solution:

	WA	NT	Q	NSW	V	SA	T
initial domains	R G B	R G B	R G B	R G B	R G B	R G B	R G B
WA=Green	G	R B	R G B	R G B	R G B	R B	R G B
V = Red	G	R B	R G B	G B	R	B	R G B
NT = Red	G	R	G B	G B	R	B	R G B
SA = Blue	G	R	G	G	R	B	R G B
Q = Green	G	R	G		R	B	R G B

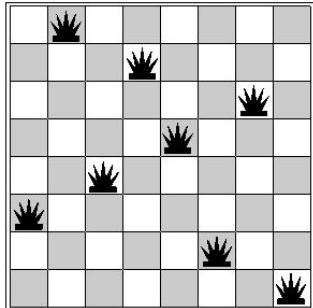
No options remain for NSW, so there is no solution.

If we also apply Arc Consistency, the question can be resolved further up the tree (but with extra computation at each node) as follows:

WA	NT	Q	NSW	V	SA	T	
initial domains	R G B	R G B	R G B	R G B	R G B	R G B	R G B
WA=Green	G	R B	R G B	R G B	R G B	R B	R G B
V = Red	G	R B	R G B	G B	R	B	R G B
NT→SA, Q→SA, NSW→SA	G	R	R G	G	R	B	R G B
Q → NT	G	R	G	G	R	B	R G B
Q → NSW	G	R		G	R	B	R G B

Question 3 - 8-queens problem

Consider the following state for the 8-queens problem:



- a) Is this a solution?

NO

- b) What is the value of h ?

There is only one violation, so $h=1$.

- c) Explain why Hill-climbing with Min Conflicts would get stuck in this state, but simulated annealing may be able to "escape" and eventually find a solution.

For each column, moving the queen on that column (while keeping the other queens in place) would result in an increase to h . Therefore, any such move will be rejected by Hill-climbing. Simulated Annealing, however, can accept such a move with probability $e^{-(h_1-h_0)/T}$, thus bumping the system out of this local optimum and allowing it to continue the search for a global optimum. (Note: when started from a random initial state, Hill-climbing will get stuck 86% of the time on this problem, and will need to be continually re-started from a new random state each time, until it succeeds.)

Question 4 - Logic Puzzle

(Exercise 6.6 from Russell & Norvig.)

Consider the following logic puzzle: In five houses, each with a different colour, live five persons of different nationalities, each of whom prefers a different brand of candy, a different drink, and a different pet. Given the following facts, the questions to answer are "Where does the zebra live, and in which house do they drink water?".

1. The Englishman lives in the red house.
2. The Spaniard owns a dog.
3. The Norwegian lives in the first house on the left.
4. The Green house is immediately to the right of the ivory house.
5. The man who eats Hershey bars lives in the house next to the man with the fox.
6. Kit Kats are eaten in the yellow house.
7. The Norwegian lives next to the blue house.
8. The Smarties eater owns snails.
9. The Snickers eater drinks orange juice.
10. The Ukrainian drinks tea.
11. The Japanese eats Milky Ways.

12. Kit Kats are eaten in a house next to the house where the horse is kept.

13. Coffee is drunk in the green house.

14. Milk is drunk in the middle house.

Discuss different representations of this problem as a CSP. Why might we prefer one representation over another?

There is a straightforward Prolog solution to this:

```
houses(Houses) :-  
    length(Houses, 5),  
    member(house(red, english, _, _, _), Houses),  
    member(house(_, spanish, dog, _, _), Houses),  
    Houses = [house(_, norwegian, _, _, _), _, _, _, _],  
    right_of(house(green, _, _, _, _), house(ivory, _, _, _, _), Houses),  
    next_to(house(_, _, _, _, hershey), house(_, _, fox, _, _), Houses),  
    member(house(yellow, _, _, _, kit_kats), Houses),  
    next_to(house(_, norwegian, _, _, _), house(blue, _, _, _, _), Houses),  
    member(house(_, _, snails, _, smarties), Houses),  
    member(house(_, _, _, orange_juice, snickers), Houses),  
    member(house(_, ukrainian, _, tea, _), Houses),  
    member(house(_, japanese, _, _, milky_ways), Houses),  
    next_to(house(_, _, _, _, kit_kats), house(_, _, horse, _, _), Houses),  
    member(house(green, _, _, coffee, _), Houses),  
    member(house(_, _, zebra, _, _), Houses),  
    Houses = [_, _, house(_, _, _, milk, _), _, _, _],  
    member(house(_, _, _, water, _), Houses),  
    print_houses(Houses).  
  
next_to(A, B, Ls) :- append(_, [A,B|_], Ls).  
next_to(A, B, Ls) :- append(_, [B,A|_], Ls).  
  
right_of(A, B, Ls) :- append(_, [B,A|_], Ls).
```

COMP3411 Tutorial - Week 5

Logic

Question 1 - Propositional Logic

Decide whether each of the following sentences is valid, satisfiable, or unsatisfiable. Verify your decisions using truth tables or logical equivalence and inference rules. For those that are satisfiable, list all the models that satisfy them.

a. $\text{Smoke} \Rightarrow \text{Smoke}$

Valid [implication, excluded middle]

b. $\text{Smoke} \Rightarrow \text{Fire}$ Satisfiable

Smoke	Fire	$\text{Smoke} \Rightarrow \text{Fire}$
T	T	T
T	F	F
F	T	T
F	F	T

Models are: $\{\text{Smoke}, \text{Fire}\}$, $\{\text{Fire}\}$, $\{\}$.

c. $(\text{Smoke} \Rightarrow \text{Fire}) \Rightarrow (\neg \text{Smoke} \Rightarrow \neg \text{Fire})$ Satisfiable

Smoke	Fire	$\text{Smoke} \Rightarrow \text{Fire}$	$\neg \text{Smoke} \Rightarrow \neg \text{Fire}$	KB
T	T	T	T	T
T	F	F	T	T
F	T	T	F	F
F	F	T	T	T

Models are: $\{\text{Smoke}, \text{Fire}\}$, $\{\text{Smoke}\}$, $\{\}$

d. $\text{Smoke} \vee \text{Fire} \vee \neg \text{Fire}$

Valid

e. $((\text{Smoke} \wedge \text{Heat}) \Rightarrow \text{Fire}) \Leftrightarrow ((\text{Smoke} \Rightarrow \text{Fire}) \vee (\text{Heat} \Rightarrow \text{Fire}))$

Valid

$$\begin{aligned} ((S \wedge H) \Rightarrow F) &\Leftrightarrow (F \vee \neg(S \wedge H)) & [\text{implication}] \\ &\Leftrightarrow (F \vee \neg S \vee \neg H) & [\text{de Morgan}] \\ &\Leftrightarrow (F \vee \neg S \vee F \vee \neg H) & [\text{idempotent, commutativity}] \\ &\Leftrightarrow (S \Rightarrow F) \vee (H \Rightarrow F) & [\text{implication}] \end{aligned}$$

f. $(\text{Smoke} \Rightarrow \text{Fire}) \Rightarrow ((\text{Smoke} \wedge \text{Heat}) \Rightarrow \text{Fire})$

Valid

$$\begin{aligned} (S \Rightarrow F) &\Leftrightarrow (F \vee \neg S) & [\text{implication}] \\ &\Rightarrow (F \vee \neg S \vee \neg H) & [\text{generalization}] \\ &\Rightarrow (F \vee \neg(S \wedge H)) & [\text{de Morgan}] \\ &\Rightarrow ((S \wedge H) \Rightarrow F) & [\text{conditional}] \end{aligned}$$

g. $\text{Big} \vee \text{Dumb} \vee (\text{Big} \Rightarrow \text{Dumb})$

Valid

$$\begin{aligned} \text{Big} \vee \text{Dumb} \vee (\text{Big} \Rightarrow \text{Dumb}) &\Leftrightarrow \text{Big} \vee \text{Dumb} \vee \text{Dumb} \vee \neg \text{Big} & [\text{implication}] \\ &\Leftrightarrow \text{Big} \vee \neg \text{Big} \vee \text{Dumb} & [\text{idempotent}] \\ &\Leftrightarrow \text{TRUE} \vee \text{Dumb} & [\text{excluded middle}] \\ &\Leftrightarrow \text{TRUE} \end{aligned}$$

h. $(\text{Big} \wedge \text{Dumb}) \vee \neg \text{Dumb}$

Satisfiable

Big	Dumb	$(\text{Big} \wedge \text{Dumb})$	$(\text{Big} \wedge \text{Dumb}) \vee \neg \text{Dumb}$
T	T	T	T
T	F	F	T
F	T	F	F
F	F	F	T

Models are: {Big, Dumb}, {Big}, {}

Question 2 - Tautologies

Determine whether the following sentences are valid (i.e. tautologies) using truth tables.

- (i) $((P \vee Q) \wedge \neg P) \rightarrow Q$
- (ii) $((P \rightarrow Q) \wedge \neg(P \rightarrow R)) \rightarrow (P \rightarrow Q)$
- (iii) $\neg(\neg P \wedge P) \wedge P$
- (iv) $(P \vee Q) \rightarrow \neg(\neg P \wedge \neg Q)$

	P	Q	$\neg P$	$P \vee Q$	$(P \vee Q) \wedge \neg P$	$((P \vee Q) \wedge \neg P) \rightarrow Q$
(i)	T	T	F	T	F	T
	T	F	F	T	F	T
	F	T	T	T	T	T
	F	F	T	F	F	T

Last column is always true no matter what truth assignment to P and Q . Therefore $((P \vee Q) \wedge \neg P) \rightarrow Q$ is a tautology.

- (ii) $S = ((P \rightarrow Q) \wedge \neg(P \rightarrow R)) \rightarrow (P \rightarrow Q)$

P	Q	R	$P \rightarrow Q$	$P \rightarrow R$	$\neg(P \rightarrow R)$	$(P \rightarrow Q) \wedge \neg(P \rightarrow R)$	S
T	T	T	T	T	F	F	T
T	T	F	T	F	T	T	T
T	F	T	F	T	F	F	T
T	F	F	F	F	T	F	T
F	T	T	T	T	F	F	T
F	T	F	T	T	F	F	T
F	F	T	T	T	F	F	T
F	F	F	T	T	F	F	T

Last column is always true no matter what truth assignment to P , Q and R . Therefore $((P \rightarrow Q) \wedge \neg(P \rightarrow R)) \rightarrow (P \rightarrow Q)$ is a tautology.

	P	$\neg P$	$\neg P \wedge P$	$\neg(\neg P \wedge P)$	$\neg(\neg P \wedge P) \wedge P$
(iii)	T	F	F	T	T
	F	T	F	T	F

Last column is not always true. Therefore $\neg(\neg P \wedge P) \wedge P$ is not a tautology.

- (iv) $(P \vee Q) \rightarrow \neg(\neg P \wedge \neg Q)$

P	Q	$\neg P$	$\neg Q$	$P \vee Q$	$\neg P \wedge \neg Q$	$\neg(\neg P \wedge \neg Q)$	$(P \vee Q) \rightarrow \neg(\neg P \wedge \neg Q)$
T	T	F	F	T	F	T	T
T	F	F	T	T	F	T	T
F	T	T	F	T	F	T	T
F	F	T	T	F	T	F	T

Last column is always true no matter what truth assignment to P and Q . Therefore $(P \vee Q) \rightarrow \neg(\neg P \wedge \neg Q)$ is a tautology.

Question 3 - Entailment

Show using the truth table method that the corresponding inferences are valid.

- (i) $P \rightarrow Q, \neg Q \models \neg P$
- (ii) $P \rightarrow Q \models \neg Q \rightarrow \neg P$
- (iii) $P \rightarrow Q, Q \rightarrow R \models P \rightarrow R$

P	Q	$P \rightarrow Q$	$\neg Q$	$\neg P$
T	T	T	F	F
T	F	F	T	F
F	T	T	F	T
F	F	T	T	T

In all rows where both $P \rightarrow Q$ and $\neg Q$ true, $\neg P$ is true. Therefore, valid inference.

P	Q	$\neg P$	$\neg Q$	$P \rightarrow Q$	$\neg Q \rightarrow \neg P$
T	T	F	F	T	T
T	F	F	T	F	F
F	T	T	F	T	T
F	F	T	T	T	T

In all rows where both $P \rightarrow Q$ true, $\neg Q \rightarrow \neg P$ is true. Therefore, valid inference.

P	Q	R	$P \rightarrow Q$	$Q \rightarrow R$	$P \rightarrow R$
T	T	T	T	T	T
T	T	F	T	F	F
T	F	T	F	T	T
T	F	F	F	T	F
F	T	T	T	T	T
F	T	F	T	F	T
F	F	T	T	T	T
F	F	F	T	T	T

In all rows where both $P \rightarrow Q$ and $Q \rightarrow R$ true, $P \rightarrow R$ is true. Therefore, valid inference.

Question 4 - Inference Rules

(Exercise 7.2 from R & N)

Consider the following Knowledge Base of facts:

If the unicorn is mythical, then it is immortal, but if it is not mythical, then it is mortal and a mammal. If the unicorn is either immortal or a mammal, then it is horned. The unicorn is magical if it is horned.

1. Translate the above statements into Propositional Logic.

$$\begin{aligned} \text{Myth} &\Rightarrow \neg \text{Mortal} \\ \neg \text{Myth} &\Rightarrow (\text{Mortal} \wedge \text{Mammal}) \\ \neg \text{Mortal} \vee \text{Mammal} &\Rightarrow \text{Horned} \\ \text{Horned} &\Rightarrow \text{Magic} \end{aligned}$$

2. Convert this Knowledge Base into Conjunctive Normal Form.

$$(\neg \text{Myth} \vee \neg \text{Mortal}) \wedge (\text{Myth} \vee \text{Mortal}) \wedge (\text{Myth} \vee \text{Mammal}) \wedge (\text{Mortal} \vee \text{Horned}) \wedge (\neg \text{Mammal} \vee \neg \text{Horned}) \wedge (\neg \text{Horned} \vee \text{Magic})$$

3. Use a series of resolutions to prove that the unicorn is Horned.

Using Proof by Contradiction, we add to the database the negative of what we are trying to prove:

$$\neg \text{Horned}$$

We then try to derive the "empty clause" by a series of Resolutions:

$$\frac{\neg \text{Horned} \wedge (\text{Mortal} \vee \text{Horned})}{\text{Mortal}}$$

$$\frac{\neg \text{Horned} \wedge (\neg \text{Mammal} \vee \text{Horned})}{\neg \text{Mammal}}$$

$$\frac{\text{Mortal} \wedge (\neg \text{Myth} \vee \neg \text{Mortal})}{\neg \text{Myth}}$$

$$\frac{\neg \text{Myth} \wedge (\text{Myth} \vee \text{Mammal})}{\text{Mammal}}$$

$$\frac{\text{Mammal} \wedge \neg \text{Mammal}}{\text{ }}$$

Having derived the empty clause, the proof (of Horned) is complete.

4. Give all models that satisfy the Knowledge Base. Can you prove that the unicorn is Mythical? How about Magical?

Because of the rule ($\text{Horned} \Rightarrow \text{Magic}$), Magic must also be True. We can construct a truth table for the remaining three variables:

Myth	Mortal	Mammal	$\text{Myth} \Rightarrow \neg \text{Mortal}$	$\neg \text{Myth} \Rightarrow (\text{Mortal} \wedge \text{Mammal})$	KB
T	T	T	F	T	F
T	T	F	F	T	F
T	F	T	T	T	T
T	F	F	T	T	T
F	T	T	T	T	T
F	T	F	T	F	F
F	F	T	T	F	F
F	F	F	T	F	F

There are three models which satisfy the entire Knowledge Base:
 $\{\text{Horned}, \text{Magic}, \text{Myth}, \text{Mammal}\}$, $\{\text{Horned}, \text{Magic}, \text{Myth}\}$, $\{\text{Horned}, \text{Magic}, \text{Mortal}, \text{Mammal}\}$.
 We cannot prove that the unicorn is Mythical, because of the third model where Mythical is False.

Question 5 - First Order Logic

Represent the following sentences in first-order logic, using a consistent vocabulary.

- a. Some students studied French in 2016.

$$\exists x \text{ Student}(x) \wedge \text{Study}(x, \text{French}, 2016)$$

- b. Only one student studied Greek in 2015.

$$\exists x \text{ Study}(x, \text{Greek}, 2015) \wedge \forall y (\text{Study}(y, \text{Greek}, 2015) \Rightarrow y = x)$$

sometimes written as

$$\exists!x \text{ Study}(x, \text{Greek}, 2015)$$

- c. The highest score in Greek is always higher than the highest score in French.

$$\forall t \exists x \forall y \text{Score}(x, \text{Greek}, t) > \text{Score}(y, \text{French}, t)$$

- d. Every person who buys a policy is smart.

$$\forall x, p \text{ Person}(x) \wedge \text{Policy}(p) \wedge \text{Buy}(x, p) \Rightarrow \text{Smart}(x)$$

- e. No person buys an expensive policy.

$$\neg \exists x, p \text{ Person}(x) \wedge \text{Policy}(p) \wedge \text{Expensive}(p) \wedge \text{Buy}(x, p)$$

- f. There is a barber who shaves all men in town who do not shave themselves.

$$\exists b \text{ Barber}(b) \wedge \forall m (\text{Man}(m) \wedge \text{InTown}(m) \wedge \neg \text{Shave}(m, m) \Rightarrow \text{Shave}(b, m))$$

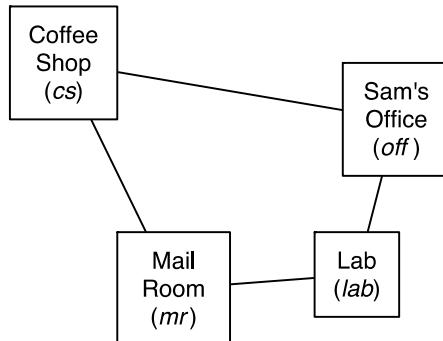
- g. Politicians can fool some of the people all of the time, and they can fool all of the people some of the time, but they can't fool all of the people all of the time.

$$\forall p (\text{Politician}(p) \Rightarrow ((\exists x \forall t \text{Fool}(p, x, t)) \wedge (\exists t \forall x \text{Fool}(p, x, t)) \wedge (\neg \forall x \forall t \text{Fool}(p, x, t))))$$

COMP3411 Tutorial - Week 7

Planning

Question 1 (Exercise 6.1 from Poole & Macworth)



Features:

RLoc – Rob's location
RHC – Rob has coffee
SWC – Sam wants coffee
MW – Mail is waiting
RHM – Rob has mail

Actions:

mc – move clockwise
mcc – move counterclockwise
puc – pickup coffee
dc – deliver coffee
pum – pickup mail
dm – deliver mail

Consider the planning problem from the lectures.

- Give the STRIPS representations for the pick up mail (*pum*) and deliver mail (*dm*) actions.
- Give the feature-based representation of the *MW* and *RHM* features.

Solution

- The pickup mail action (*pum*) is defined using STRIPS by:

Preconditions: $RLoc = mr \wedge mw$

Effects: $[\neg mw, rhm]$

The deliver mail action (*dm*) is defined using STRIPS by:

Preconditions: $RLoc = off \wedge rhm$

Effects: $[\neg rhm]$

- The *MW* feature can be axiomatised by defining when $MW = true$ (written as *mw*):

$mw' \leftarrow mw \wedge Action \neq pum$

The *RHM* feature can be axiomatised by defining when $RHM = true$ (written as *rhm*):

$rhm' \leftarrow Action = pum$

$rhm' \leftarrow rhm \wedge Action \neq dm$

Question 2

Formulate the blocks world using STRIPS planning operators. The actions are stack (move one block to the top of another) and unstack (move one block to the table). The robot can hold only one block at a time.

To simplify the world, assume the only objects are the blocks and the table, and that the only relations are the on relation between (table and) blocks and the clear predicate on table and blocks. Also assume that it is not possible for more than one block to directly support another block (and vice versa).

Solution

stack(A, B):

Preconditions: clear(A) \wedge clear(B)

Effects: on(A, B) \wedge \neg clear(B)

unstack(A):

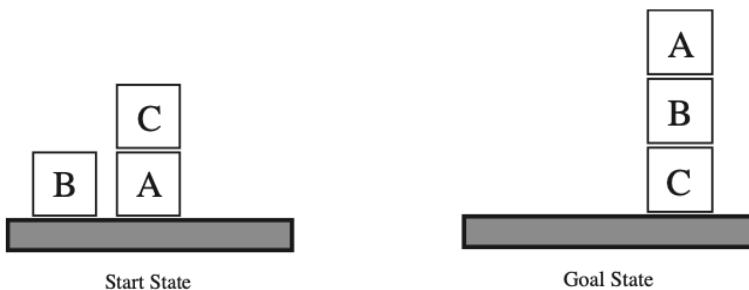
Preconditions: clear(A) \wedge on(A, B)

Effects: on(A, Table) \wedge clear(B) \wedge \neg on(A, B)

Note: The **Effects** could also be written as an addles contain only the positive literals and the delete list, containing only the negative literals.

Question 3

The Sussman anomaly, shown below, is a simple planning problem that could not be solved by the early linear planners. Show how a partial order planner would solve this problem with the blocks world operators defined above.



Solution

- The nonlinear planner introduces the two actions stack(B, C) and stack(A, B).
- The clear(A) precondition of stack(A,B) does not hold in the initial state, so unstack(C) is added to the plan.
- Because stack(A,B) deletes clear(B), which is a precondition of stack(B,C), stack(B,C) must be before stack(A,B).
- For the same reason, unstack(C) must be before move(B, C).
- The plan is therefore unstack(C), stack(B, C), stack(A, B).

COMP3411/9814 Artificial Intelligence 20T1, 2020

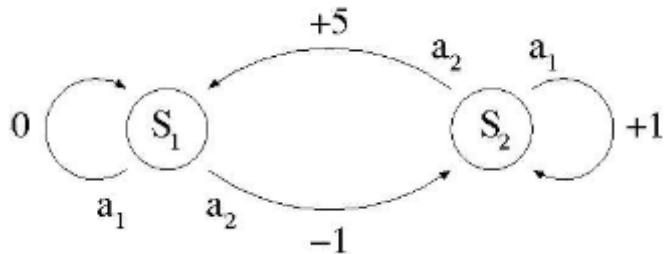
Tutorial Solutions - Week 8

Question 1

Consider a world with two states $S = \{S_1, S_2\}$ and two actions $A = \{a_1, a_2\}$, where the transitions δ and reward r for each state and action are as follows:

$$\begin{array}{ll} \delta(S_1, a_1) = S_1 & r(S_1, a_1) = 0 \\ \delta(S_1, a_2) = S_2 & r(S_1, a_2) = -1 \\ \delta(S_2, a_1) = S_2 & r(S_2, a_1) = +1 \\ \delta(S_2, a_2) = S_1 & r(S_2, a_2) = +5 \end{array}$$

- (i) Draw a picture of this world, using circles for the states and arrows for the transitions.



- (ii) Assuming a discount factor of $\gamma = 0.9$, determine:

- (a) The optimal policy is:

$$\begin{aligned} \pi^*(S_1) &= a_2 \\ \pi^*(S_2) &= a_2 \end{aligned}$$

- (b) The optimal value function V^* is calculated as follows.

$$\begin{aligned} V^*(S_1) &= -1 + \gamma V^*(S_2) \\ V^*(S_2) &= 5 + \gamma V^*(S_1) \end{aligned}$$

$$\begin{aligned} \text{So } V^*(S_1) &= -1 + 5\gamma + \gamma^2 V^*(S_1) \\ \text{i.e. } V^*(S_1) &= (-1 + 5\gamma)/(1 - \gamma^2) = 3.5/0.19 = 18.42 \end{aligned}$$

$$V^*(S_2) = 5 + \gamma V^*(S_1) = 5 + 0.9 * 3.5/0.19 = 21.58$$

- (c) The Q function for the optimal policy is calculated as follows.

$$\begin{aligned} Q(S_1, a_1) &= \gamma V^*(S_1) = 16.58 \\ Q(S_1, a_2) &= V^*(S_1) = 18.42 \\ Q(S_2, a_1) &= 1 + \gamma V^*(S_2) = 20.42 \\ Q(S_2, a_2) &= V^*(S_2) = 21.58 \end{aligned}$$

(iii) Write the Q values in a table.

Q	a_1	a_2
S_1	16.58	18.42
S_2	20.42	21.58

(iv) Trace through the first few steps of the Q-learning algorithm, with all Q values initially set to zero. Explain why it is necessary to force exploration through probabilistic choice of actions in order to ensure convergence to the true Q values.

current state	chosen action	new Q value
S_1	a_1	$0 + \gamma * 0 = 0$
S_1	a_2	$-1 + \gamma * 0 = -1$
S_2	a_1	$1 + \gamma * 0 = 1$

At this point, the table looks like this:

Q	a_1	a_2
S_1	0	-1
S_2	1	0

If the agent always chooses the current best action, it can have a policy where it always prefers a suboptimal action, e.g. a_1 in state S_2 , so will never sufficiently explore action a_2 . This means that $Q(S_2, a_2)$ will remain zero forever, instead of converging to the true value of 21.58. With exploration, the next few steps might look like this:

current state	chosen action	new Q value
S_2	a_2	$5 + \gamma * 0 = 5$
S_1	a_1	$0 + \gamma * 0 = 0$
S_1	a_2	$-1 + \gamma * 5 = 3.5$
S_2	a_1	$1 + \gamma * 5 = 5.5$
S_2	a_2	$5 + \gamma * 3.5 = 8.15$

Now we have this table:

Q	a_1	a_2
S_1	0	3.5
S_2	5.5	8.15

From this point on, the agent will prefer action a_2 both in state S_1 and in state S_2 . Further steps refine the Q value estimates, and, in the limit, they will converge to their true values.

current state	chosen action	new Q value
S_1	a_1	$0 + \gamma * 3.5 = 3.15$
S_1	a_2	$-1 + \gamma * 8.15 = 6.335$
S_2	a_1	$1 + \gamma * 8.15 = 8.335$
S_2	a_2	$5 + \gamma * 6.34 = 10.70$
...

Question 2

Consider the task of predicting whether children are likely to be hired to play members of the Von Trapp Family in a production of The Sound of Music, based on these data:

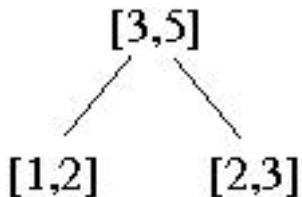
height	hair	eyes	hired
short	blond	blue	+
tall	red	blue	+
tall	blond	blue	+
tall	blond	brown	-
short	dark	blue	-
tall	dark	blue	-
tall	dark	brown	-
short	blond	brown	-

- a. Compute the information (entropy) gain for each of the three attributes (height, hair, eyes) in terms of classifying objects as belonging to the class, + or - .

There are 3 objects in class '+' and 5 in '-', so the entropy is:

$$\text{Entropy(parent)} = \sum_i P_i \log_2 P_i = -(3/8)\log(3/8) - (5/8)\log(5/8) = 0.954$$

Suppose we split on height:



Of the 3 'short' items, 1 is '+' and 2 are '-', so $\text{Entropy(short)} = -(1/3)\log(1/3) - (2/3)\log(2/3) = 0.918$

Of the 5 'tall' items, 2 are '+' and 3 are '-', so $\text{Entropy(tall)} = -(2/5)\log(2/5) - (3/5)\log(3/5) = 0.971$

The average entropy after splitting on 'height' is $\text{Entropy(height)} = (3/8)(0.918) + (5/8)(0.971) = 0.951$

The information gained by testing this attribute is: $0.954 - 0.951 = 0.003$ (i.e. very little)

If we try splitting on 'hair' we find that the branch for 'dark' has 3 items, all '-' and the branch for 'red' has 1 item, in '+'. Thus, these branches require no further information to make a decision. The branch for 'blond' has 2 '+' and 2 '-' items and so requires 1 bit. That is,

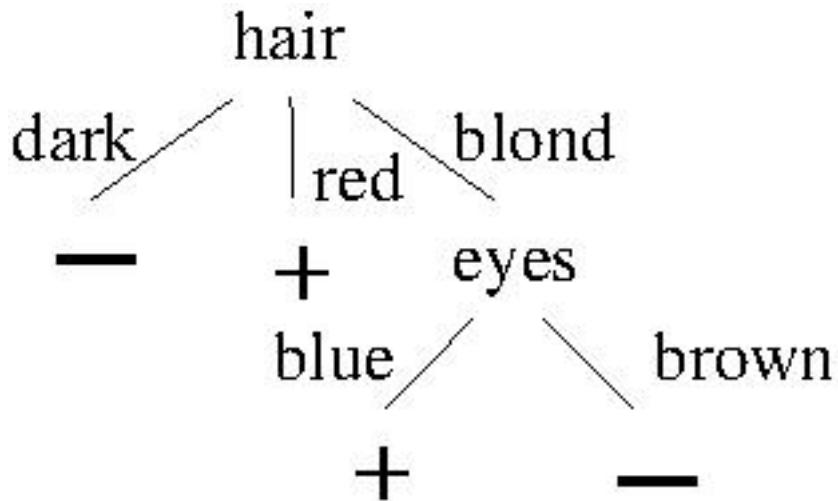
$$\text{Entropy(hair)} = (3/8)(0) + (1/8)(0) + (4/8)(1) = 0.5$$

and the information gained by testing hair is $0.954 - 0.5 = 0.454$ bits.

By a similar calculation, the entropy for testing 'eyes' is $(5/8)(0.971) + (3/8)(0) = 0.607$, so the information gained is $0.954 - 0.607 = 0.347$ bits.

Thus 'hair' gives us the maximum information gain.

- b. Construct a decision tree based on the minimum entropy principle. Since the 'blond' branch for hair still contains a mixed population, we need to apply the procedure recursively to these four items. Note that we now only need to test 'height' and 'eyes' since the 'hair' attribute has already been used. If we split on 'height', the branch for 'tall' and 'short' will each contain one '+' and one '-', so the entropy gain is zero. If we split on 'eyes', the 'blue' branch contains two +'s and the 'brown' branch two '-'s, so the tree is complete:

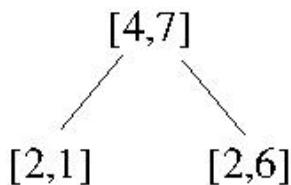


Question 3

The Laplace error estimate for pruning a node in a Decision Tree is given by:

$$E = 1 - \frac{n + 1}{N + k}$$

where N is the total number of items, n is the number of items in the majority class and k is the number of classes. Given the following subtree, should the children be pruned or not? Show your calculations.



$$\text{Error(Parent)} = 1 - (7+1)/(11+2) = 1 - 8/13 = 5/13 = 0.385$$

$$\text{Error(Left)} = 1 - (2+1)/(3+2) = 1 - 3/5 = 2/5 = 0.4$$

$$\text{Error(Right)} = 1 - (6+1)/(8+2) = 1 - 7/10 = 3/10 = 0.3$$

$$\text{Backed Up Error} = (3/11)*(0.4) + (8/11)*(0.3) = 0.327 < 0.385$$

Since Error of Parent is larger than Backed Up Error \Rightarrow Don't Prune

Question 4

Construct a Decision Tree for the following set of examples.

Day	Outlook	Temperature	Humidity	Wind	PlayTennis
D1	Sunny	Hot	High	Weak	No
D2	Sunny	Hot	High	Strong	No
D3	Overcast	Hot	High	Weak	Yes
D4	Rain	Mild	High	Weak	Yes
D5	Rain	Cool	Normal	Weak	Yes
D6	Rain	Cool	Normal	Strong	No
D7	Overcast	Cool	Normal	Strong	Yes
D8	Sunny	Mild	High	Weak	No
D9	Sunny	Cool	Normal	Weak	Yes
D10	Rain	Mild	Normal	Weak	Yes
D11	Sunny	Mild	Normal	Strong	Yes
D12	Overcast	Mild	High	Strong	Yes
D13	Overcast	Hot	Normal	Weak	Yes
D14	Rain	Mild	High	Strong	No

What class is assigned to the instance {D15, Sunny, Hot, High, Weak}?

1. (i) $Values(Outlook) = \{sunny, overcast, rain\}$

$$S = [9+, 5-]$$

$$S_{sunny} \leftarrow [2+, 3-]$$

$$S_{overcast} \leftarrow [4+, 0-]$$

$$S_{rain} \leftarrow [3+, 2-]$$

$$\begin{aligned} Gain(S, Outlook) &= Entropy(S) - \sum_{v=\{sunny, overcast, rain\}} \frac{|S_v|}{|S|} Entropy(S_v) \\ &= Entropy(S) - \frac{5}{14} Entropy(S_{sunny}) - \frac{4}{14} Entropy(S_{overcast}) - \frac{5}{14} Entropy(S_{rain}) \\ &= 0.940 - \frac{5}{14} \times 0.971 - \frac{4}{14} \times 0 - \frac{5}{14} \times 0.971 \\ &= 0.247 \end{aligned}$$

$$\begin{aligned} Entropy(S) &= Entropy([9+, 5-]) = -\frac{9}{14} \log_2 \frac{9}{14} - \frac{5}{14} \log_2 \frac{5}{14} \\ &= 0.940 \end{aligned}$$

$$Entropy(S_{sunny}) = Entropy([2+, 3-]) = -\frac{2}{5} \log_2 \frac{2}{5} - \frac{3}{5} \log_2 \frac{3}{5} = 0.971$$

$$Entropy(S_{overcast}) = Entropy([4+, 0-]) = -\frac{4}{4} \log_2 \frac{4}{4} - \frac{0}{4} \log_2 \frac{0}{4} = 0$$

$$Entropy(S_{rain}) = Entropy([3+, 2-]) = -\frac{3}{5} \log_2 \frac{3}{5} - \frac{2}{5} \log_2 \frac{2}{5} = 0.971$$

- (ii) $Values(Temperature) = \{hot, mild, cool\}$

$$S = [9+, 5-]$$

$$S_{hot} \leftarrow [2+, 2-]$$

$$S_{mild} \leftarrow [4+, 2-]$$

$$S_{cool} \leftarrow [3+, 1-]$$

$$\begin{aligned} Gain(S, Temperature) &= Entropy(S) - \sum_{v=\{hot, mild, cool\}} \frac{|S_v|}{|S|} Entropy(S_v) \\ &= Entropy(S) - \frac{4}{14} Entropy(S_{hot}) - \frac{6}{14} Entropy(S_{mild}) - \frac{4}{14} Entropy(S_{cool}) \\ &= 0.940 - \frac{4}{14} \times 1.00 - \frac{6}{14} \times 0.918 - \frac{4}{14} \times 0.811 \\ &= 0.029 \end{aligned}$$

$$Entropy(S_{hot}) = Entropy([2+, 2-]) = -\frac{2}{4} \log_2 \frac{2}{4} - \frac{2}{4} \log_2 \frac{2}{4} = 1.00$$

$$Entropy(S_{mild}) = Entropy([4+, 2-]) = -\frac{4}{6} \log_2 \frac{4}{6} - \frac{2}{6} \log_2 \frac{2}{6} = 0.918$$

$$Entropy(S_{cool}) = Entropy([3+, 1-]) = -\frac{3}{4} \log_2 \frac{3}{4} - \frac{1}{4} \log_2 \frac{1}{4} = 0.811$$

(iii) $Values(Humidity) = \{high, normal\}$

$$S = [9+, 5-]$$

$$S_{high} \leftarrow [3+, 4-]$$

$$S_{normal} \leftarrow [6+, 1-]$$

$$\begin{aligned} Gain(S, Humidity) &= Entropy(S) - \sum_{v=\{high, normal\}} \frac{|S_v|}{|S|} Entropy(S_v) \\ &= Entropy(S) - \frac{7}{14} Entropy(S_{high}) - \frac{7}{14} Entropy(S_{normal}) \\ &= 0.940 - \frac{7}{14} \times 0.985 - \frac{7}{14} \times 0.592 \\ &= 0.152 \end{aligned}$$

$$Entropy(S_{high}) = Entropy([3+, 4-]) = -\frac{3}{7} \log_2 \frac{3}{7} - \frac{4}{7} \log_2 \frac{4}{7} = 0.985$$

$$Entropy(S_{normal}) = Entropy([6+, 1-]) = -\frac{6}{7} \log_2 \frac{6}{7} - \frac{1}{7} \log_2 \frac{1}{7} = 0.592$$

(iv) $Values(Wind) = \{weak, strong\}$

$$S = [9+, 5-]$$

$$S_{weak} \leftarrow [6+, 2-]$$

$$S_{strong} \leftarrow [3+, 3-]$$

$$\begin{aligned} Gain(S, Wind) &= Entropy(S) - \sum_{v=\{weak, strong\}} \frac{|S_v|}{|S|} Entropy(S_v) \\ &= Entropy(S) - \frac{8}{14} Entropy(S_{weak}) - \frac{6}{14} Entropy(S_{strong}) \\ &= 0.940 - \frac{8}{14} \times 0.811 - \frac{6}{14} \times 1.00 \\ &= 0.048 \end{aligned}$$

$$Entropy(S_{weak}) = Entropy([6+, 2-]) = -\frac{6}{8} \log_2 \frac{6}{8} - \frac{2}{8} \log_2 \frac{2}{8} = 0.811$$

$$Entropy(S_{strong}) = Entropy([3+, 3-]) = -\frac{3}{6} \log_2 \frac{3}{6} - \frac{3}{6} \log_2 \frac{3}{6} = 1.00$$

2. (i) $Values(Temperature) = \{hot, mild, cool\}$

$$S_{sunny} = [2+, 3-]$$

$$S_{sunny, hot} \leftarrow [0+, 2-]$$

$$S_{sunny, mild} \leftarrow [1+, 1-]$$

$$S_{sunny, cool} \leftarrow [1+, 0-]$$

$$\begin{aligned} Gain(S_{sunny}, Temperature) &= Entropy(S_{sunny}) - \sum_{v=\{hot, mild, cool\}} \frac{|S_{sunny,v}|}{|S_{sunny}|} Entropy(S_{sunny,v}) \\ &= Entropy(S) - \frac{2}{5} Entropy(S_{sunny, hot}) - \frac{2}{5} Entropy(S_{sunny, mild}) - \frac{1}{5} Entropy(S_{sunny, cool}) \\ &= 0.971 - \frac{2}{5} \times 0.00 - \frac{2}{5} \times 1.00 - \frac{1}{5} \times 0.00 \\ &= 0.571 \end{aligned}$$

$$Entropy(S_{sunny}) = Entropy([2+, 3-]) = -\frac{2}{5} \log_2 \frac{2}{5} - \frac{3}{5} \log_2 \frac{3}{5} = 0.971$$

$$Entropy(S_{sunny, hot}) = Entropy([0+, 2-]) = -\frac{0}{2} \log_2 \frac{0}{2} - \frac{2}{2} \log_2 \frac{2}{2} = 0.00$$

$$Entropy(S_{sunny, mild}) = Entropy([1+, 1-]) = -\frac{1}{2} \log_2 \frac{1}{2} - \frac{1}{2} \log_2 \frac{1}{2} = 1.00$$

$$Entropy(S_{sunny, cool}) = Entropy([1+, 0-]) = -\frac{1}{1} \log_2 \frac{1}{1} - \frac{0}{1} \log_2 \frac{0}{1} = 0.00$$

(ii) $Values(Humidity) = \{high, normal\}$

$$S_{sunny} = [2+, 3-]$$

$$S_{sunny, high} \leftarrow [0+, 3-]$$

$$S_{sunny, normal} \leftarrow [2+, 0-]$$

$$\begin{aligned} Gain(S, Humidity) &= Entropy(S_{sunny}) - \sum_{v=\{high, normal\}} \frac{|S_{sunny,v}|}{|S_{sunny}|} Entropy(S_{sunny,v}) \\ &= Entropy(S_{sunny}) - \frac{3}{5} Entropy(S_{sunny, high}) - \frac{2}{5} Entropy(S_{sunny, normal}) \\ &= 0.971 - \frac{3}{5} \times 0.00 - \frac{2}{5} \times 0.00 \\ &= 0.971 \end{aligned}$$

$$Entropy(S_{sunny, high}) = Entropy([0+, 3-]) = -\frac{0}{3} \log_2 \frac{0}{3} - \frac{3}{3} \log_2 \frac{3}{3} = 0.00$$

$$Entropy(S_{sunny, normal}) = Entropy([2+, 0-]) = -\frac{2}{2} \log_2 \frac{2}{2} - \frac{0}{2} \log_2 \frac{0}{2} = 0.00$$

3. (i) $Values(Temperature) = \{hot, mild, cool\}$

$$S_{rain} = [2+, 3-]$$

$$S_{rain,hot} \leftarrow [0+, 0-]$$

$$S_{rain,mild} \leftarrow [2+, 1-]$$

$$S_{rain,cool} \leftarrow [1+, 1-]$$

$$\begin{aligned} Gain(S_{rain}, Temperature) &= Entropy(S_{rain}) - \sum_{v=\{hot, mild, cool\}} \frac{|S_{rain,v}|}{|S_{rain}|} Entropy(S_{rain,v}) \\ &= Entropy(S) - \frac{0}{5} Entropy(S_{rain,hot}) - \frac{3}{5} Entropy(S_{rain,mild}) - \frac{2}{5} Entropy(S_{rain,cool}) \\ &= 0.971 - \frac{0}{5} \times 0.00 - \frac{3}{5} \times 0.918 - \frac{2}{5} \times 1.00 \\ &= 0.020 \end{aligned}$$

$$Entropy(S_{rain}) = Entropy([3+, 2-]) = -\frac{3}{5} \log_2 \frac{3}{5} - \frac{2}{5} \log_2 \frac{2}{5} = 0.971$$

$$Entropy(S_{rain,hot}) = Entropy([0+, 0-]) = -\frac{0}{0} \log_2 \frac{0}{0} - \frac{0}{0} \log_2 \frac{0}{0} = 0.00$$

$$Entropy(S_{rain,mild}) = Entropy([2+, 1-]) = -\frac{2}{3} \log_2 \frac{2}{3} - \frac{1}{3} \log_2 \frac{1}{3} = 0.918$$

$$Entropy(S_{rain,cool}) = Entropy([1+, 1-]) = -\frac{1}{2} \log_2 \frac{1}{2} - \frac{1}{2} \log_2 \frac{1}{2} = 1.00$$

(ii) $Values(Humidity) = \{high, normal\}$

$$S_{rain} = [3+, 2-]$$

$$S_{rain,high} \leftarrow [1+, 1-]$$

$$S_{rain,normal} \leftarrow [1+, 1-]$$

$$\begin{aligned} Gain(S, Humidity) &= Entropy(S_{rain}) - \sum_{v=\{high, normal\}} \frac{|S_{rain,v}|}{|S_{rain}|} Entropy(S_{rain,v}) \\ &= Entropy(S_{rain}) - \frac{2}{5} Entropy(S_{rain,high}) - \frac{3}{5} Entropy(S_{rain,normal}) \\ &= 0.971 - \frac{2}{5} \times 1.00 - \frac{3}{5} \times 0.551 \\ &= 0.020 \end{aligned}$$

$$Entropy(S_{rain,high}) = Entropy([1+, 1-]) = -\frac{1}{2} \log_2 \frac{1}{2} - \frac{1}{2} \log_2 \frac{1}{2} = 1.00$$

$$Entropy(S_{rain,mild}) = Entropy([2+, 1-]) = -\frac{2}{3} \log_2 \frac{2}{3} - \frac{1}{3} \log_2 \frac{1}{3} = 0.551$$

$$S_{rain,high} \leftarrow [1+, 1-]$$

$$S_{rain,normal} \leftarrow [1+, 1-]$$

$$\begin{aligned} Gain(S, Humidity) &= Entropy(S_{rain}) - \sum_{v=\{high, normal\}} \frac{|S_{rain,v}|}{|S_{rain}|} Entropy(S_{rain,v}) \\ &= Entropy(S_{rain}) - \frac{2}{5} Entropy(S_{rain,high}) - \frac{3}{5} Entropy(S_{rain,normal}) \\ &= 0.971 - \frac{2}{5} \times 1.00 - \frac{3}{5} \times 0.551 \\ &= 0.020 \end{aligned}$$

$$Entropy(S_{rain,high}) = Entropy([1+, 1-]) = -\frac{1}{2} \log_2 \frac{1}{2} - \frac{1}{2} \log_2 \frac{1}{2} = 1.00$$

$$Entropy(S_{rain,mild}) = Entropy([2+, 1-]) = -\frac{2}{3} \log_2 \frac{2}{3} - \frac{1}{3} \log_2 \frac{1}{3} = 0.551$$

(iii) $Values(Wind) = \{weak, strong\}$

$$S_{rain} = [3+, 2-]$$

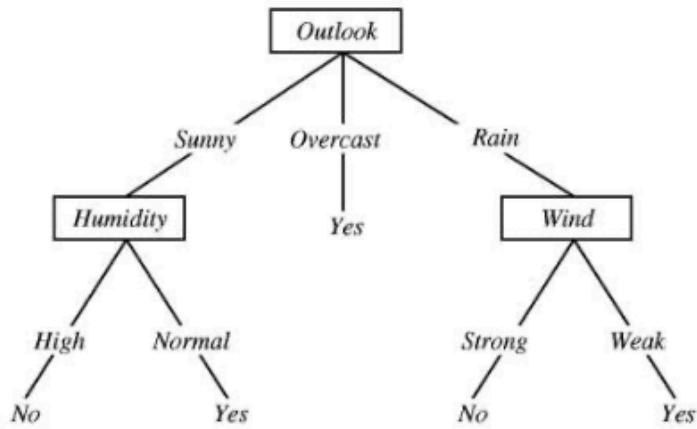
$$S_{weak} \leftarrow [3+, 0-]$$

$$S_{strong} \leftarrow [0+, 2-]$$

$$\begin{aligned} Gain(S, Wind) &= Entropy(S_{rain}) - \sum_{v=\{weak, strong\}} \frac{|S_{rain,v}|}{|S_{rain}|} Entropy(S_{rain,v}) \\ &= Entropy(S) - \frac{3}{5} Entropy(S_{rain,weak}) - \frac{2}{5} Entropy(S_{strong}) \\ &= 0.971 - \frac{3}{5} \times 0.00 - \frac{2}{5} \times 0.00 \\ &= 0.971 \end{aligned}$$

$$Entropy(S_{weak}) = Entropy([3+, 0-]) = -\frac{3}{3} \log_2 \frac{3}{3} - \frac{0}{3} \log_2 \frac{0}{3} = 0.00$$

$$Entropy(S_{strong}) = Entropy([0+, 2-]) = -\frac{0}{2} \log_2 \frac{0}{2} - \frac{2}{2} \log_2 \frac{2}{2} = 0.00$$



So the example is assigned the *No* class.

COMP3411 Tutorial - Week 9

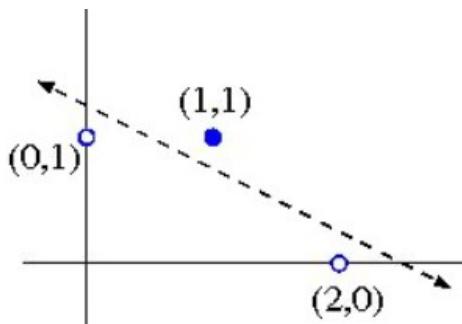
Neural Nets and ILP

Question 1

- a) Construct by hand a Perceptron which correctly classifies the following data; use your knowledge of plane geometry to choose appropriate values for the weights w_0 , w_1 and w_2 .

Training Example	x_1	x_2	Class
a.	0	1	-1
b.	2	0	-1
c.	1	1	+1

The first step is to plot the data on a 2-D graph, and draw a line which separates the positive from the negative data points:



Based on the two intersection points, we can derive the following line slope:

$$Slope = m = \frac{(y_2 - y_1)}{(x_2 - x_1)} = \frac{(0 - 1)}{(2 - 0)} = -0.5$$

With the slope, the corresponding line equation can be partially calculated, utilising an identified point on the line, which will be $(0.5, 1)$ for this example (the point between $(0,1)$ and $(1,1)$):

$$\begin{aligned} y &= mx + b \\ &\cong 1 = -0.5 * 0.5 + b \\ b &= 5/4 \equiv 1.25 \\ y &= -0.5x + 1.25 \end{aligned}$$

Since x_2 is equivalent to y based on our dataset, we can deduce a quadratic equation using our line equation which will provide the weightings:

$$\begin{aligned} x_2 &= -0.5x_1 + 1.25 \\ \equiv 0 &= 0.5x_1 + x_2 - 1.25 \\ \equiv 0 &= 2x_1 + 4x_2 - 5 \end{aligned}$$

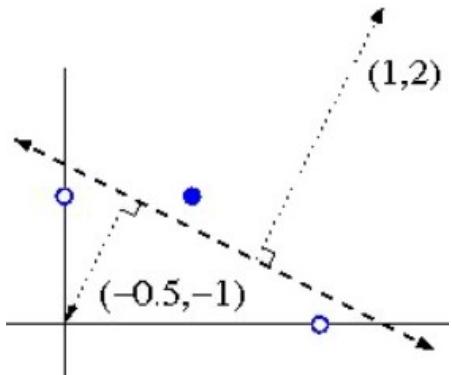
This results in the following weights:

$$w_0 = -5$$

$$w_1 = 2$$

$$w_2 = 4$$

Alternatively, we can derive weights $w_1=1$ and $w_2=2$ by drawing a vector normal to the separating line, in the direction pointing towards the positive data points:



The bias weight w_0 can then be found by computing the dot product of the normal vector with a perpendicular vector from the separating line to the origin. In this case $w_0 = 1(-0.5) + 2(-1) = -2.5$

Note: these weights differ from the previous ones by a normalising constant, which is fine for a perceptron.

- b) Demonstrate the Perceptron Learning Algorithm on the above data, using a learning rate of 1.0 and the following initial weight values:

$$w_0 = -0.5$$

$$w_1 = 0$$

$$w_2 = 1$$

Be sure to round the predicted result to the closest class value available (-1 or 1 in this case).

Iteration	w ₀	w ₁	w ₂	Example	x ₁	x ₂	Class	Prediction	Rounded	Action
1	-2.5	0.0	-1.0	a	0.0	1.0	-1	0.5	1.0	Subtract
2	-2.5	0.0	-1.0	b	2.0	0.0	-1	-2.5	-1	None
3	-0.5	2.0	1.0	c	1.0	1.0	1	-3.5	-1	Add
4	-2.5	2.0	-1.0	a	0.0	1.0	-1	0.5	1	Subtract
5	-4.5	-2.0	-1.0	b	2.0	0.0	-1	1.5	1	Subtract
6	-2.5	0.0	1.0	c	1.0	1.0	1	-7.5	-1	Add
7	-2.5	0.0	1.0	a	0.0	1.0	-1	-1.5	-1	None
8	-2.5	0.0	1.0	b	2.0	0.0	-1	-2.5	-1	None
9	-0.5	2.0	3.0	c	1.0	1.0	1	-1.5	-1	Add
10	-2.5	2.0	1.0	a	0.0	1.0	-1	2.5	1.0	Subtract
11	-4.5	-2.0	1.0	b	2.0	0.0	-1	1.5	1.0	Subtract
12	-2.5	0.0	3.0	c	1.0	1.0	1	-5.5	-1	Add
13	-4.5	0.0	1.0	a	0.0	1.0	-1	0.5	1	Subtract
14	-4.5	0.0	1.0	b	2.0	0.0	-1	-4.5	-1	None
15	-2.5	2.0	3.0	c	1.0	1.0	1	-3.5	-1	Add
16	-4.5	2.0	1.0	a	0.0	1.0	-1	0.5	1.0	Subtract
17	-4.5	2.0	1.0	b	2.0	0.0	-1	-0.5	-1.0	None
18	-2.5	4.0	3.0	c	1.0	1.0	1	-1.5	-1	Add
19	-4.5	4.0	1.0	a	0.0	1.0	-1	0.5	1.0	Subtract
20	-6.5	0.0	1.0	b	2.0	0.0	-1	3.5	1	Subtract
21	-4.5	2.0	3.0	c	1.0	1.0	1	-5.5	-1	Add
22	-4.5	2.0	3.0	a	0.0	1.0	-1	-1.5	-1	None
23	-4.5	2.0	3.0	b	2.0	0.0	-1	-0.5	-1	None
24	-4.5	2.0	3.0	c	1.0	1.0	1	0.5	1	None

Note that we only have three training examples (a, b, c), but we keep iterating over them until no corrections are required for the network to produce the correct output.

Question 2

Explain how each of the following could be constructed:

- a) Perceptron to compute the OR function of m inputs

Set the bias weight to $-\frac{1}{2}$, all other weights to 1.

The OR function is almost always True. The only way it can be False is if all inputs are 0. Therefore, we set the bias to be slightly less than zero for this input.

- b) Perceptron to compute the AND function of n inputs

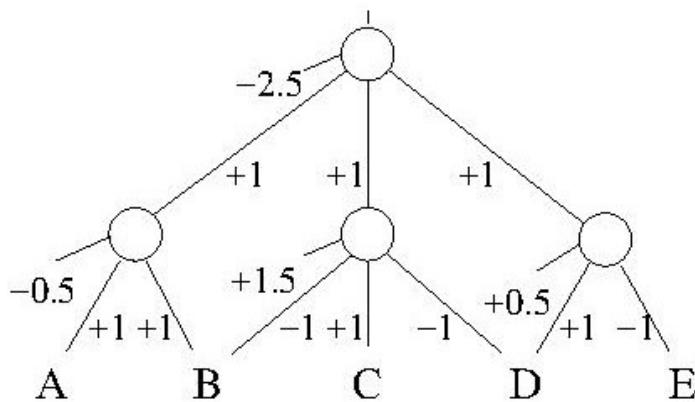
Set the bias weight to $\left(\frac{1}{2} - n\right)$, all other weights to 1.

AND function is almost always False. The only way it can be True is if all inputs are 1. Therefore, we set the bias so that, when all inputs are 1, the combined sum is slightly greater than 0.

- c) 2-Layer Neural Network to compute any (given) logical expression, assuming it is written in Conjunctive Normal Form.

Each hidden node should compute one disjunctive term in the expression. The weights should be -1 for items that are negated, +1 for others. The bias should be $k - \frac{1}{2}$ where k is the number of items that are negated. The output node then computes the conjunction of all hidden nodes.

For example, here the network computes $(A \vee B) \wedge (\neg B \vee C \vee \neg D) \wedge (D \vee \neg E)$



Question 3

- a) Find the least general generalisation of the following terms:

```
f(g(a, b), [1, 2, [3, 4], 5], 1 + 2 * 6)
f(g(a, h(x, y)), [1, 2, [3, 4, 5]], 1 + 6)
```

To find the LGG, look for where the terms differ and replace with a variable. Reuse the same variable if the same terms appear in corresponding positions.

```
lgg = f(g(a, X), [1, 2, [3, 4 | Y] | Y], 1 + Z)
subst = [X/{b, h(x, y)}, Y/{[], [5]}, Z/{2*6, 6}]
```

The list is tricky because the inverse substitution $Y/{}[], [5]\}$ applies also as $Y/{}[5], []\}$.

- b) Find the least general generalisation of the following clauses:

```
q(f(a)) :- p(a, b), r(b, c), r(b, e).
q(f(x)) :- p(x, y), r(y, z), r(w, z).
```

To find the LGG of two clauses, match the head, then find all combinations of possible LGGs between literals that have consistent inverse substitutions.

```
q(f(X)) :- p(X, Y), r(Y, Z), r(W, Z), R(Y, E), r(W, E)
subst = [X/{a, x}, Y/{b, y}, Z/{c, z}, W/{b, w}, E/{e, z}]
```