**CSC 520-001/601**

# Introduction to Artificial Intelligence

**Fall 2015**

[Back to Moodle] [Back to Course at a Glance]

[Lecture Notes] [Section Index] [Next]

# Outline

1. Introduction to Artificial Intelligence
2. Connections to Other Fields
3. Facets of AI

## Introduction to Artificial Intelligence

Most definitions of Artificial Intelligence (AI), including the one in Russell & Norvig, avoid trying to define intelligence itself. Instead, the goals of AI can be grouped into categories, depending on whether researchers are interested mostly in modeling human intelligence or in constructing useful tools and systems; and on whether the effort is concentrated on processing ("thinking") or on exhibiting intelligent behavior.

| Psychology | Engineering |
|---|---|
| **Think like humans**<br>Cognitive science, cognitive modeling | **Think rationally**<br>Law of Thought<br>Aristotle and other Greeks, logic<br>Problem: representing knowledge, complexity |
| **Act like humans**<br>The Turing Test: requires<br>• natural language processing,<br>• knowledge representation,<br>• automated reasoning,<br>• learning,<br>• vision,<br>• robotics, etc. | **Act rationally**<br>More general than the Laws of Thought<br>Needs all the Turing Test requirements, plus Agents, etc.<br>compare to reflexes (not intelligent) |

### Another Distinction

- **Top-down:** explicit programming of desired behavior; facts, rules, encoding of human expertise,

etc. Expert systems, rule-based systems, most other approaches.
- **Bottom-up:** coding of "micro-behaviors", massively parallel collections of simple elements, intelligence as an **emergent** property; insect robotics, genetic algorithms, ACO, many other optimization methods.
- Most learning systems show aspects of both types.

## Connections to Other Fields

AI at its edges overlaps with:

- philosophy (esp. philosophy of knowledge, etc.)
- math (esp. logic, probability, statistics) -- algorithms, Godel undecidability, intractability and incompleteness
- economics (esp utility theory, decision theory, game theory
- neuroscience
- psychology (esp behaviorism, cognitive psychology, cognitive science
- computer engineering
- control theory and cybernetics
- linguistics

## Facets of AI

Among the subspecialties:

- planning and scheduling
- game playing
- control
- diagnosis
- logistics
- robotics
- natural language processing

---

[Lecture Notes] [Section Index] [Next]

---

Top of Page

---


Dr. Dennis Bahler /

**CSC 520-001/601**

# Introduction to Artificial Intelligence

**Fall 2015**

[Back to Moodle] [Back to Course at a Glance]

[Lecture Notes] [Section Index] [Previous]

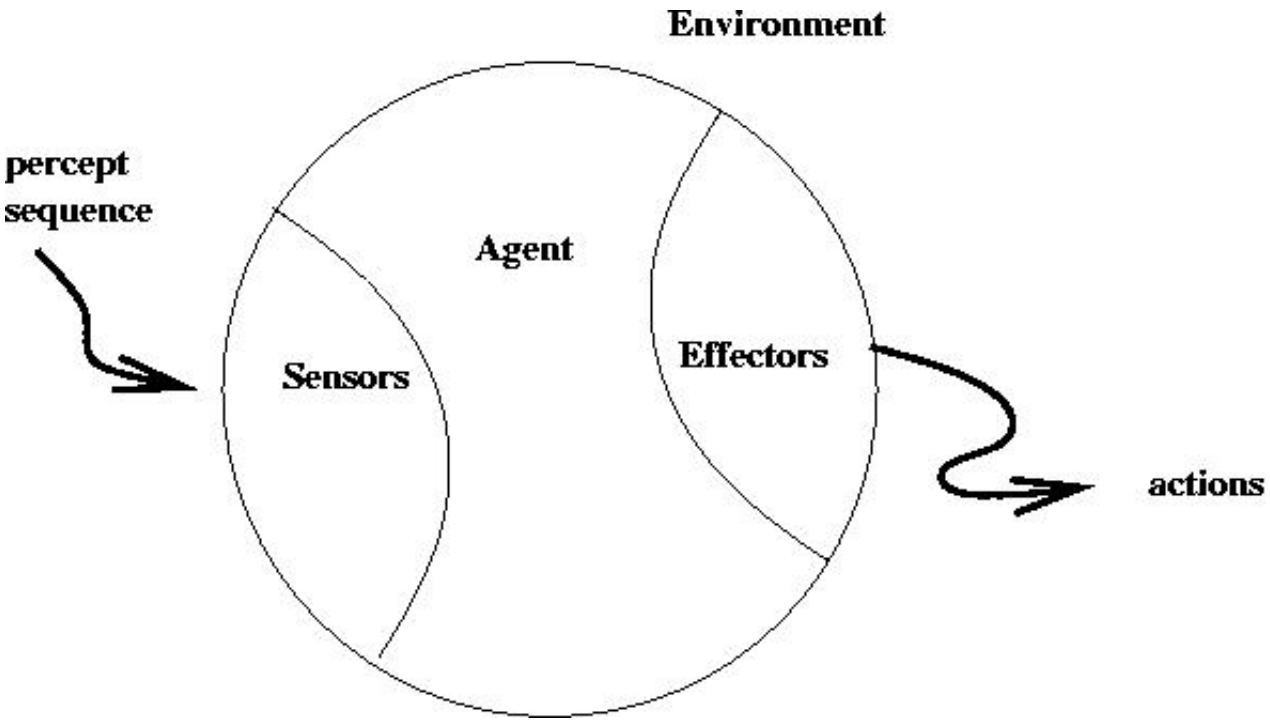
## Outline



## Intelligent Agents

An **Agent** is a general term for an entity situated in an **environment**, which can act to change the environment, such as by moving, manipulating objects with the environment, etc.

### High Level Architecture




An agent usually operates on the basis of an **agent function**, an abstraction that can be viewed as a

function mapping a time-ordered sequence of one or more percepts to an action: $f_1$: $P_1$, ..., $P_n$ --> A. An **agent program** is a particular concrete implementation of an agent function.

The actions are externally visible and detected by the environment (and any observer). The internals of the agent are visible only to the agent itself. The percepts may or may not be externally visible.

## How to Evaluate Agent Performance

Intelligent agent behavior depends on **rationality**, which in turn requires:

1. a **performance measure** P;
2. an **environment** E (see below for more on the space of possible environments), and nearly always at least some **prior knowledge** of E;
3. a set of **actuators** A that can perform actions, or alternatively, the set of **actions** the agent can perform via the actuators;
4. a set of **sensors** S, or alternatively, the set of **percepts** or percept sequences that the agent can detect.

The agent task, then, is to do the following forever:

For every percept sequence detected via S, select an action and perform it via A to maximize its performance measure P, given the percept sequence and the agent's knowledge (prior or otherwise) of E.

P, E, A, and S (above) together define a **task environment**, a form of **problem specification**.

Rationality does not require omniscience. Omniscience means knowing everything about everything. This is unattainable and unnecessary; humans don't try for omniscience either.

There may be actions whose effect is to acquire other percepts: "Look both ways before crossing the street...."

An agent that can **learn** has the ability to augment its prior knowledge.

An agent that is **autonomous** relies more and more on its percepts and not on its prior knowledge as times goes on. In other words, it records what it needs to, in order to improve its performance.

## Varieties of Environments

Environments can be classified according to where they fall on a number of scales:

- **Fully vs. partially observable**. Can all relevant properties of the environment be observed by the agent, or are some of them sometimes or always hidden?
- **Deterministic vs. stochastic**. Do all the actions have dependable consequences, or is there a random component?
- **Episodic vs. sequential**. In an episodic environment, things are organized by single percept-action pairs. The next episode does not depend on previous actions in an episodic environment. In a sequential environment, it might.
- **Static vs. dynamic**. Do aspects of the environment change independent of the agent actions, or not? Can things change "by themselves"?
- **Discrete vs. continuous**. Does the environment consist of discrete points in space, or is space continuous? The same can be asked about time, percepts, and actions.

- **Single agent vs. multiagent**. Is there more than one agent situated in the environment at one time? Other agents can be competitive, cooperative, or some of each depending on circumstances.

The most complex environments are:

- partially observable,
- stochastic,
- sequential,
- dynamic,
- continuous, and
- multiagent.

## Agent Structure

The idea of agent is so general that agents can vary widely in complexity.

**Agent functions** map a percept or a series of percepts to an action. **Agent programs** implement an agent function.

A **Table-Driven Agent** uses a simple lookup table to find a corresponding action for each possible percept. The (BIG) problem is, the table size combinatorically explodes (i.e., grows faster than any polynomial function of the number of percepts, and there are a LOT of percepts in most environments).

A **Simple Reflex Agent** uses a function to map a single percept to a single action, not an explicit table.

A **Model-Based Reflex Agent** maintains an explicit internal representation of the state of the world, and can use that representation in deciding on actions. The world model can capture the agent's view of how the world evolves independently of the agent, and how the agent's actions affect the world. A model-based agent can handle a partially observable environment, at least at the elementary level. It internal state depends on its percept history (or at least part of it).

A **Goal-Based Agent** maintains explicit goals, and thus can rank world states according to desirability. It can do planning and search, and can distinguish between alternative actions with the same consequences, by ranking them according to cost, measured somehow.

A **Utility-Based Agent** assigns a numerical utility value to each state description (U: state --> real_numbers). High-utility world states represent goals.

**Learning Agents** contain a **learning element** that can make improvements based on experience to some aspect of what the agent does. Often this involves a **performance element** to select action, a **critic** to give feedback on the choices, and a **problem generator** to suggest new actions.

---

[Lecture Notes] [Section Index] [Previous]

---

Top of Page

---

**CSC 520-001/601**

# Introduction to Artificial Intelligence

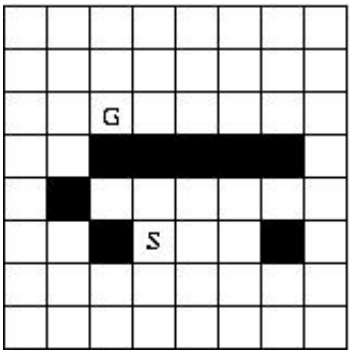
**Fall 2015**

[Back to Moodle] [Back to Course at a Glance]

[Section Index] [Next]

## A Problem in Agent Navigation

Consider the problem of finding a path from position S to position G in the 8-by-8 grid environment shown below:




The agent can move on the grid either horizontally or vertically, one square at a time. There are no diagonal moves, and no step may be taken into a forbidden shaded area or off the grid.

[Section Index] [Next]

Top of Page

**CSC 520-001/601**

# Introduction to Artificial Intelligence

**Fall 2015**

---

[Back to Moodle] [Back to Course at a Glance]

[Section Index] [Previous] [Next]

---

## Outline

1. What is Search?
2. State Spaces and Search Graphs
3. Search in State Spaces
4. Uninformed vs. Informed Algorithms

---

## What is Search?

A major ultimate goal of AI is to construct **agents** that can behave intelligently in their **environment**. **Goal-based** agents can succeed in this by considering future actions and the desirability of their outcomes.

One type of goal-based agent is the **problem-solving** agent. Problem-solving agents choose what to do by finding **solutions** to problems, i.e., sequences of actions that lead to desirable states of the environment (and of the agent itself).

One concrete means of defining "problem" and "solution" is to represent the world as a collection of **states** that the environment+agent can be in, and to represent each action as a **transition** between a pair of states. This yields a **graph** with states labeling the **nodes** and actions labeling the **edges**. A goal, then, is a state, or more realistically, a set of states. The idea is to find a sequence of actions that will bring about a **goal state** starting from **current state**, the state which describes the current state of the environment+agent. Such a sequence corresponds to a **path** in the graph.

The process of looking for such a sequence is called **search**. Search is the systematic enumeration of potential partial solutions to a problem so that they can be checked to see if they truly are solutions, or could lead to solutions.

---

## State Spaces and Search Graphs

The set of all possible world states is called the **state space**. The entire state space can be represented explicitly as a graph if it is small enough. Most realistic state spaces, however, are too large for this. For this reason only as much of the graph as necessary is ever explicitly represented. A good search algorithm should look at as small a subgraph of the state space as it can. This subgraph, which depends on the search algorithm used, is called the **search graph**.

The state space depends on the problem and how states are represented; the search graph depends on the state space and the algorithm used to search the state space.

---

## Search in State Spaces

Regardless of the details, all search algorithms must contain these components:

1. **Data Structure:** Some data structure to contain candidate solution paths. Paths here are ordered sequences of states, where states are connected by edges labeled with moves. Depending on the algorithm, this data structure may be as simple as a stack or queue, or more complex.
2. **Path expansion:** Path expansion is a three-step process:
    a. Removing a path from a data structure (queue, stack, priority queue, hash table, whatever), possibly testing for goal-match (**goal match**) along the way,
    b. Generating its successor paths (**move generation**), and
    c. Inserting the successor paths back onto the data structure.

    Path expansion will in turn include **move generation** and **goal match**.
3. **Move generation:** Move generation applies to the terminal (i.e., last) state on a path. For each state, we need a means of determining,
    a. All the actions that are enabled in that state, and
    b. For each of those actions, the state that would result if the action were taken.

    The portion of an implementation which performs this task is often called a **move generator**. Move generators may be as simple as a table lookup, or as complex as arbitrarily large segments of code.
4. **Goal match:** The component which determines if a state matches a goal specification. If we are keeping track of paths on the data structure, we should match the goal only with the terminal state on a path. The complexity of this step depends on the representation of a state. It is essential that this step occur at the appropriate place in the algorithm. Specifically, in all the algorithms, if we are truly interested in an optimal cost path, we cannot return immediately after some successor matches a goal, because, although we have found a path to a goal-matching state, we may not have found the **shortest** path. The only exceptions to this are the simplest algorithms, such as depth-first and breadth-first, which pay no attention to path cost.

Algorithms may be compared against each other by comparing:

a. Number of node expansions,
b. Quality of solution path returned,
c. Some combination of these.

---

## Path-Oriented Algorithms vs. Node-Oriented Algorithms

Most often we are interested in solution paths, not just yes-no information on whether a solution path exists. Unfortunately for us, most discussions refer to **node-oriented** algorithms. These node-oriented algorithms (items on the data structure are nodes, nodes are expanded, etc.) usually require extra work to recover the actual solution path when they terminate. This work may involve parent pointers, back-pointers, or similar extra information.

The section just above describes a path-oriented algorithm, not the more common node-oriented algorithm.

If we construct a **path-oriented** algorithm (items on the data structure are paths, paths are expanded, etc.) then the solution path is simple to obtain, because we have been dealing with paths throughout the operation of the algorithm.

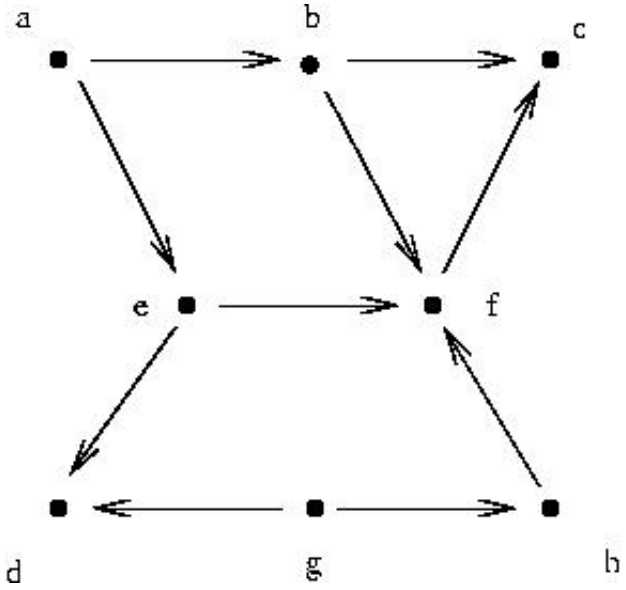
All algorithms presented in psuedo-code form below are path oriented.

---

## Uninformed vs. Informed Algorithms

Search algorithms fall into two categories:

- **Uninformed:** An uninformed search algorithm has no more information about states that what's included in the problem definition, i.e., the state space nodes and edges themselves; Uninformed search algorithms are sometimes called **blind search** algorithms.




- **Informed:** An informed search algorithm can measure whether one non-goal state is "more promising" than another. To do this, the algorithm applies a **heuristic** function to the state. A heuristic is a rule-of-thumb; it may not always be exactly accurate, but it's good enough as an approximate measure, enough of the time to be useful.

  A heuristic function in simplest form simply takes a state description (and whatever other information is appropriate) as input, and returns a real number. Most informed search algorithms apply their heuristic to quantitatively estimate the distance in the underlying state space between a current state and some goal state, and therefore take two state descriptions as input and output a real number.

  Informed search algorithms are sometimes called **heuristic search** algorithms.

---

[Section Index] [Previous] [Next]

Top of Page

Comments or suggestions are always welcome.
Dr. Dennis Bahler /

# CSC 520-001/601 Introduction to Artificial Intelligence Fall 2015

[Back to Moodle] [Back to Course at a Glance]

[Section Index] [Previous] [Next]

## Outline

Uninformed Search

1. Depth-First
2. Breadth-First
3. Uniform Cost: Dynamic Programming
4. Depth-Bounded
    1. Depth-First Iterative Deepening

Notice we use a stack (or queue, etc.) **of paths** so the algorithm returns the solution path, not just the last node on it. Otherwise, returning path information will involve keeping track of it at expand time, such as with parent pointers, etc.

Remember, in an algorithm which keeps track of paths rather than single nodes, goal-match means pattern-matching the goal to the last node on a path.

Finally, to review from the previous page, removing a node from a data structure (queue, stack, priority queue, hash table, whatever) and generating its successors, possibly testing for goal-match along the way, is called **expanding** that node.

---

## Depth-First (DFS)

Time: $O(b^d)$, Space: O(d), Problem: nontermination

```
Initialize a stack of paths with the one-node path consisting of the initial state
While (stack not empty)
    Pop top path
    If last node on path matches goal, return path
    Else extend the path by one node in all possible ways,
        by generating successors of the last node on the path
    Push successor paths on top of stack
Return FAIL
```

---

## Breadth-First (BFS)

Time: $O(b^d)$, Space: $O(b^d)$

```
Initialize a queue of paths with the one-node path consisting of the initial state
While (queue not empty)
    Remove path on front of queue
    If last node on path matches goal, return path
    Else extend the path by one node in all possible ways,
        by generating successors of the last node on the path
    Insert successor paths on rear of queue
Return FAIL
```

---

## Uniform Cost: Dynamic Programming

Time: $O(b^d)$, Space: $O(b^d)$

This algorithm, and all those remaining, requires paths to be considered in order of how promising they appear to be. Instead of maintaining a linear data structure which must be sorted very iteration, a more efficient implementation avoids lots of redundant sorting by using a PRIORITY QUEUE.

```
Initialize a priority queue of paths with the one-node path consisting of the initial state
While (queue not empty)
    Remove path at root (which will be of min cost)
    If last node on path matches goal, return path
    Else extend the path by one node in all possible ways, by generating successors of the last node on the path
    Foreach successor path succ
        Update path cost to succ
        Insert succ on queue and re-heapify
            using PATH COST FROM START NODE as the priority
    If two or more paths reach the same node, delete all paths except
        the one of min cost
Return FAIL
```

---

## Depth-First Iterative Deepening (DFID)

Time: $O(b^d)$, Space: $O(d)$

NOTE: The inner loop of DFID is doing ***depth-first*** search.

In the simplest case (but not often in practice), `initialCutoff = 1` and `cutoffIncrement = 1`.

This version assumes a solution path exists.

```
cutoff = initialCutoff
goalNotFound = true
While (goalNotFound)
    initialize a stack of paths with the one-node path consisting of the initial state
    depth(start) = 0
    While (stack not empty)
    |   Pop top path
    |   If last node on path matches goal, return path
    |   Else extend the path by one node in all possible ways,
    |       by generating successors of the last node on the path
    |   Foreach successor path succ
    |       depth(succ) = depth(parent(succ)) + 1
    |       If depth(succ) <= cutoff
    |           Push succ on stack
    cutoff = cutoff + cutoffIncrement
```

---

[Section Index] [Previous] [Next]

Top of Page

---


Dr. Dennis Bahler /

**CSC 520-001/601** # Introduction to Artificial Intelligence **Fall 2015**

[Back to Moodle] [Back to Course at a Glance]

[Section Index] [Previous] [Next]

## Outline

Informed Search

1. A General Paradigm: Best-First
2. Greedy Best-First
3. Greedy 2: Hillclimbing
4. A*
5. An Application: Route-Finding in Romania
6. IDA*

---

This class of algorithms are said to be **informed**, in that they use a problem-class- (or even problem-instance-) specific heuristic estimate of the distance between two states, and search those nodes first that are most promising using this estimate.

---

## A General Paradigm: Best-First

Time: $O(b^d)$, Space: $O(b^d)$

```
Initialize a priority queue of paths with the one-node path consisting of the initial state
While (queue not empty)
    Remove path at root (which will be of min cost)
    If last node on path matches goal, return path
    Else extend the path by one node in all possible ways,
        by generating successors of the last node on the path
    Foreach successor path succ
        Compute cost, MEASURED SOMEHOW, of succ
        Insert succ on queue and re-heapify using COST as the priority
Return FAIL
```

Do not confuse the search graph of states with the priority queue of states. These are different structures. For example:

1. The root of the search graph is the initial state; this doesn't change. The search graph contains all states of the state space that we have discovered so far.
2. Remember, we keep track of paths, not nodes one at a time, because we want the algorithms all to return a solution path. We don't directly represent the search graph anywhere.
3. The root of the priority queue is the path of min cost; this changes in general each iteration. The priority queue contains all paths whose last node has not yet been expanded. Priority queues are often implemented as complete binary min trees, hence the use of the (overloaded) term "root".

NOTE: The notion of "best-first" can encompass both uninformed and informed algorithms, depending on how "best" is measured.

---

## Greedy Best-First

Time: heuristically $O(b^d)$ or better, depending on how good the estimate h-hat is, Space: $O(b^d)$

```
Initialize a priority queue of paths with the one-node path consisting of the initial state
While (queue not empty)
    Remove path at root (which will be of min cost)
    If last node on path matches goal, return path
    Else extend the path by one node in all possible ways,
        by generating successors of the last node on the path
```

```
    Foreach successor path succ
        Heuristically estimate remaining distance (h-hat) to goal from last node on succ
        Insert succ on queue and re-heapify
            using ESTIMATED REMAINING DISTANCE TO GOAL as the priority
Return FAIL
```

---

## Greedy 2: Hillclimbing

```
Initialize a stack of paths with the one-node path consisting of the initial state
While (stack not empty)
    Pop top path
    If last node on path matches goal, return path
    Else extend the path by one node in all possible ways,
        by generating successors of the last node on the path
    Foreach successor path succ
        Heuristically estimate remaining distance (h-hat) to goal from last node on succ
    Sort successor paths by increasing h-hat
    Push successors on stack in sorted order so that min cost path is on top
Return FAIL
```

---

## A*

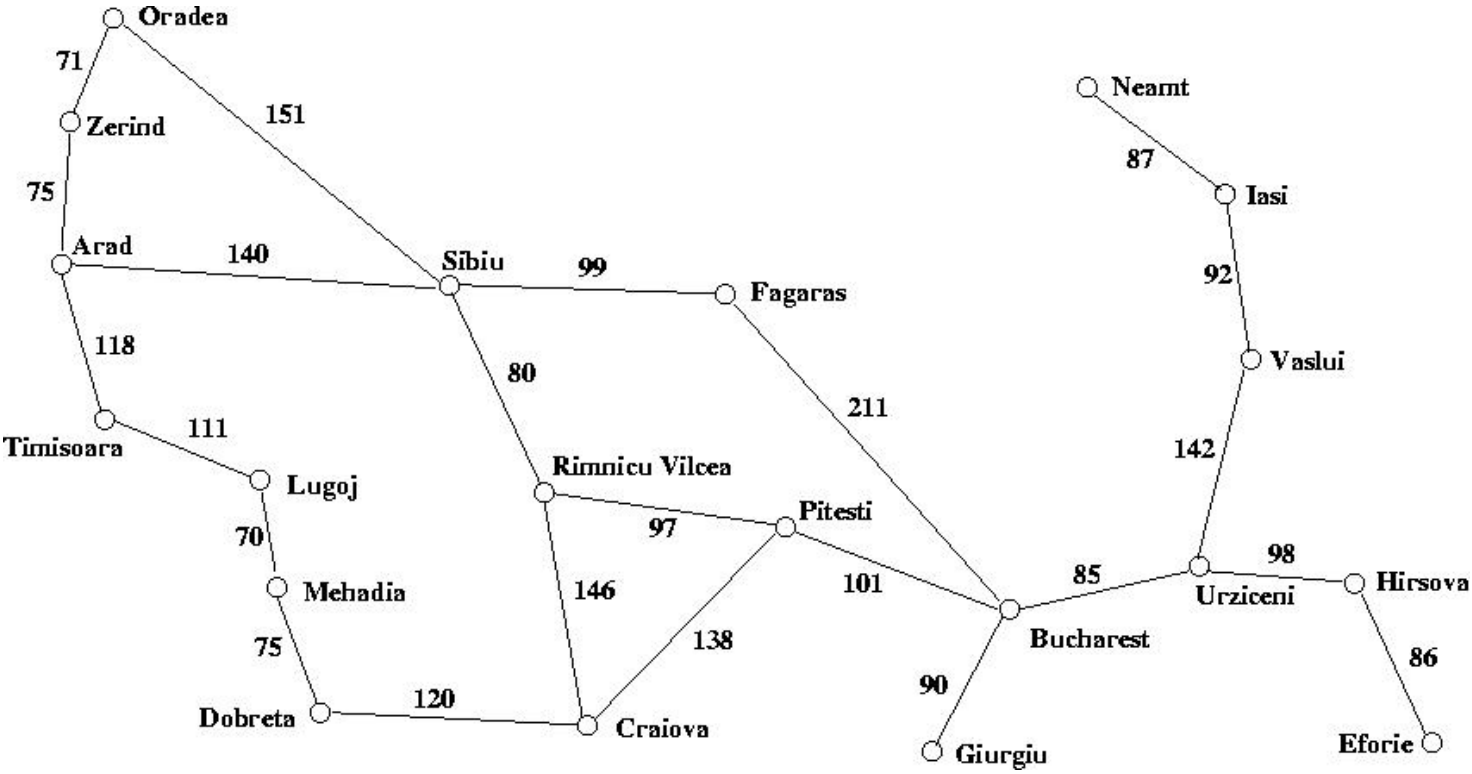
Time: heuristically $O(b^d)$ or better, depending on how good the estimate h-hat is, Space: $O(b^d)$

In all informed algorithms, with a perfect estimate, time complexity is linear in path length. In general, time complexity is a function of the ERROR in the estimate of remaining distance.

```
Initialize a priority queue of paths with the one-node path consisting of the initial state
While (queue not empty)
    Remove path at root (which will be of min cost)
    If last node on path matches goal, return path
    Else extend the path by one node in all possible ways,
        by generating successors of the last node on the path
    Foreach successor path succ
        Update path cost (g) to succ
        Heuristically estimate remaining distance (h-hat) to goal from last node on succ
        Insert succ on queue and re-heapify
            using SUM OF PATH COST FROM ROOT (g) AND
                ESTIMATED REMAINING DISTANCE TO GOAL (h-hat) as the priority
    If two or more paths reach the same node, delete all paths except
        the one of min cost
Return FAIL
```

---

## An Application: Route-Finding in Romania




These cities actually exist, mostly: [Map] [roads.pl].

## Iterative Deepening A* (IDA*)

Time: heuristically $O(b^d)$ or better, depending on how good h-hat is, Space: O(d)

IDA* combines the ideas of DFID and the heuristic estimate of A*. Notice that it's stack-based and is doing DFS in the inner loop, hence the space-complexity. IDA* amounts to DFID with a heuristically variable cutoff, so we can search more promising paths deeper, earlier.

```
cutoff = f(root) ( = h-hat(root) because g(root) = 0)
While (goal not found)
    Initialize a stack of paths with the one-node path consisting of the initial state
    Construct list L initially empty
    While (stack not empty)
    |   Pop top path
    |   If last node on path matches goal, return path
    |   Else extend the path by one node in all possible ways,
    |       by generating successors of the last node on the path
    |   Foreach successor path succ
    |       Update path cost (g) to succ
    |       Heuristically estimate remaining distance (h-hat) to goal from last node on succ
    |       f(succ) = g(succ) + h-hat(succ)
    |       If f(succ) <= cutoff
    |           Push succ on stack
    |       Else append f() to list L
    cutoff = minimum f() on list L such that f() > cutoff
```

List L could easily itself be a min heap, so the min element would be trivial to find. This heap will be subject to multiple inserts (and re-heapify steps) and one retrieval per iteration.

[Section Index] [Previous] [Next]

Top of Page

Dr. Dennis Bahler /

**CSC 520-001/601**

# Introduction to Artificial Intelligence

**Fall 2015**

---

[Back to Moodle] [Back to Course at a Glance]

[Section Index] [Previous]

---

## Outline

1. Game Tree Search
2. Min-Max
3. An Example Game Tree
4. Alpha-Beta Pruning

---

## Game Tree Search

It may be the case that an agent's task is to play (and win) a game against another agent, or even against a human opponent. This page discusses the most common way such an agent goes about determining which move to make, at each stage of a particular type of game. Computer game playing of this kind involves a different kind of search from the uninformed or informed algorithms presented earlier.

Assumptions:

- Two players: A and B
- Zero-sum: A wins and B loses, or vice-versa
- Perfect knowledge (no random component, no dice throwing, etc.)
- Search to a bounded depth (called the **search horizon**)
- Examples: chess, checkers, go, othello, etc.

To keep things straight, all discussion will be from the perspective of Player A.

Player A wants to choose the best move. That is, A wants to choose any one of the moves that cause all of B's possible replies to lead to a win for A.

By extension, A wants to choose any one of the moves that cause all of B's possible replies to lead to a second move for A that causes all of B's possible replies to lead to a win for A.

Or ultimately, A wants to choose any one of the moves that cause all of B's possible replies to lead to a second move for A that causes all of B's possible replies to lead to a third move for A that causes all of B's possible replies to lead to
...
an $n^{th}$ move for A that causes all of B's possible replies to lead to a win for A.

This process can be captured by a tree, where

- each node is labeled by a game position,
- edges are labeled by moves,
- the number of successors of a node is the number of moves possible in that position,
- and the successor node is labeled by the position resulting from that move.

The label on the root of the tree is the initial (or more generally, current) position of the game.

Obviously, only for trivially simple games can we produce a full tree all the way from the beginning of the game to the end. In most realistic cases, we can only generate a few moves ahead. How far ahead depends on:

1. the branching factor of the tree (average number of possible moves from a game position),
2. how much time we have before we have to decide on a move, and
3. how efficiently we can search the tree for the best possible move in the current position.

The number of moves ahead we can search is called the **search horizon**. Since the game is almost always not finished at the search horizon, we usually don't know for sure which player will eventually win. The best we can do is to apply an **Evaluation Function** to the game state, which estimates the goodness of a position (from A's point of view) as a numeric value. The evaluation function works like the heuristic estimate H in the $A^*$ algorithm.

The resulting trees are called **Game Trees** and the search problem is called **Game Tree Search**.

## Min-Max

Since Player A wants the best move, A is said to be **maximizing** the outcome (from A's perspective). Since Player B wants the best move from B's perspective, which will be the worst move from A's perspective (in a zero-sum game), B is said to be **minimizing** the outcome (from A's perspective). Since play alternates between A and B, the resulting game tree (when it's A's turn to move) has a maximizing root, whose immediate successors are minimizing nodes, the successors of which are maximizing, etc. As a result, such a game tree is often called a **Min-Max Tree**.

Pseudocode, where d is edge depth (number of edges from root to node n):

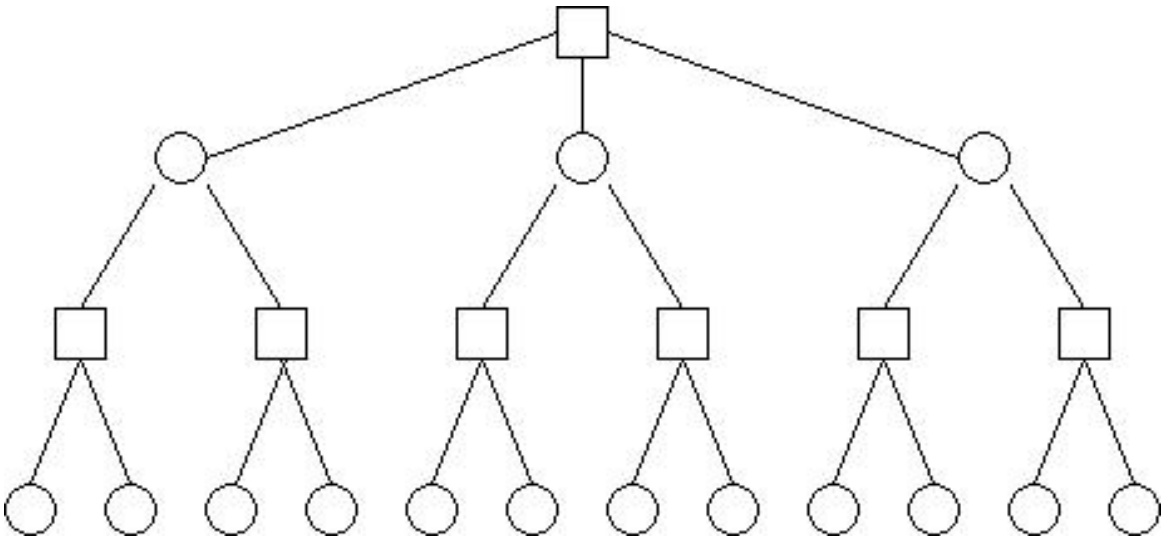
```
function value(node n, depth d) {
    if (isLeaf(n) {
        return[(-1)^d * evalFunction(n)]
    }
    return[max_over_successors_of_n(-value(n, d+1))]
}
```

## An Example Game Tree

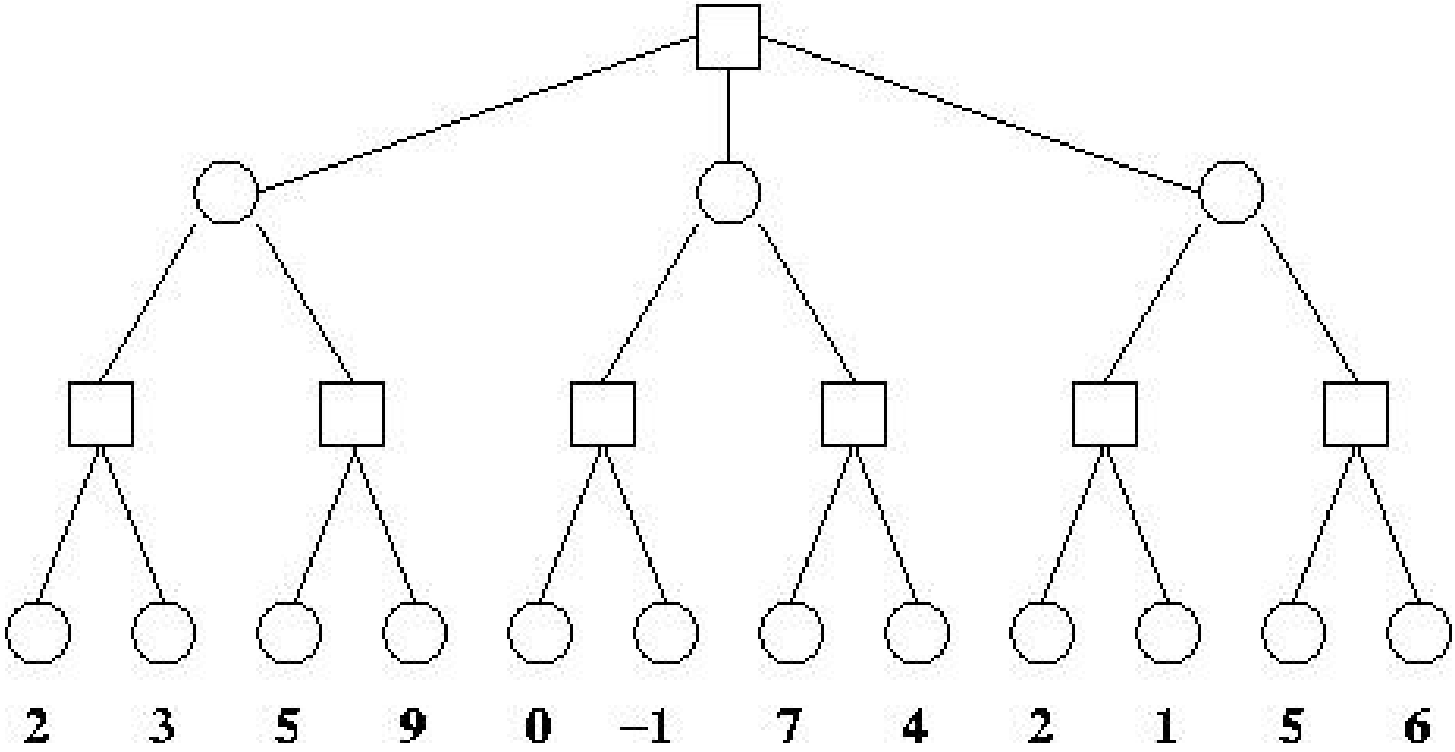
In this game tree:

- Nodes would be labeled by game states (not shown).
- Edges are labeled by the various moves that A and B can make (also not shown).
- It's A's turn to move.
- Square nodes (A's turn) maximize and circular nodes (B's turn) minimize.

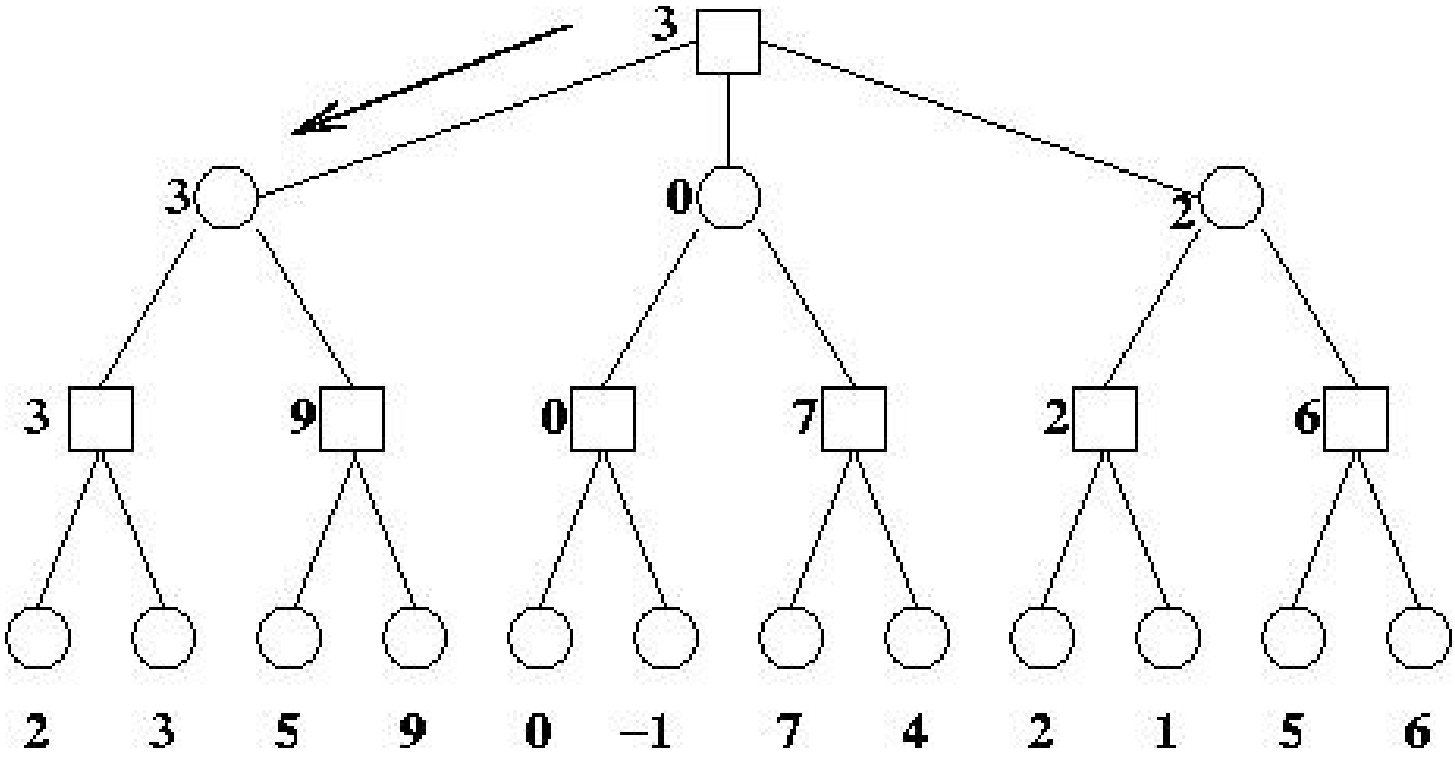
---



Applying an (arbitrary, in this case) evaluation function to the leaves:




The scores at sibling leaves are maximized by their parent, and the resulting maxima are minimized by *their* parent, and so on bottom-up, until the root receives a score. The edge from which this score came corresponds to the move that A should make.




## Alpha-Beta Pruning

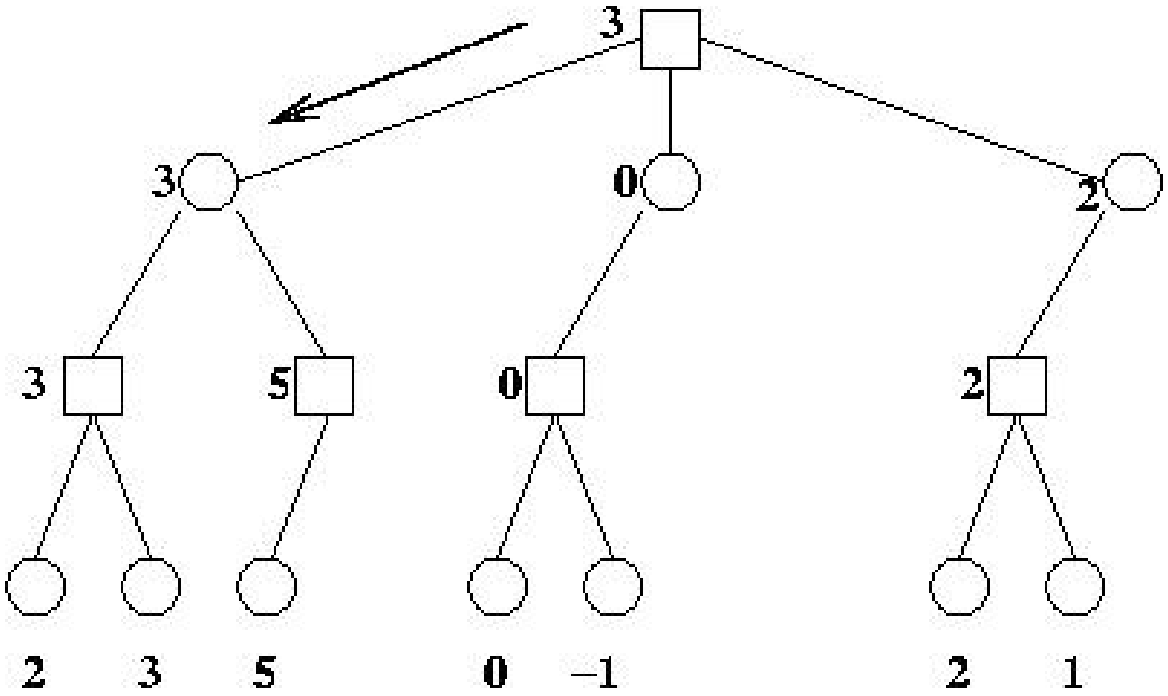
In order to search as far ahead as possible, it is essential to be as efficient as possible in the search of the game tree. A major saving comes about by observing that game tree search is depth-first, and the search can be short-circuited if no game state on the current path can possibly be as good as a state on a previously examined path.

This process can be implemented by associating an interval with each node, initialized to (-inf, +inf) and shortened from one end or the other during search until it is a single point, the min-max value of the node.

[Search Sequence]

**Alpha Cutoff**: eliminating the generation of successors to a MINIMIZING node $n_1$ because the value of that node must be WORSE than the value of a sibling minimizing node $n_2$, because $n_1$ will never be chosen by the maximizing parent of $n_1$ and $n_2$.

**Beta Cutoff**: eliminating the generation of successors to a MAXIMIZING node $n_1$ because the value of that node must be BETTER than the value of a sibling maximizing node $n_2$, because $n_1$ will never be chosen by the minimizing parent of $n_1$ and $n_2$.




```
function value(node n, float alpha, float beta) {
    if (isLeaf(n)) return evalFunction(n);
    else {
        m = alpha;
        foreach successor succ {
            t = -value(succ, -beta, -m);
            if (t > m) m = t;
            if (m >= beta) return m;
        }
        return m;
    }
}
```

To invoke:

```
value(root, -MAXINT, MAXINT);
```

For more general termination than at leaf, modify `value()` to include a parameter for depth:

```
function value(node n, float alpha, float beta, int depth) {
    if (done(n, depth)) return evalFunction(n, depth);
    else {
        m = alpha;
        foreach successor succ {
            t = -value(succ, -beta, -m, depth+1);
            if (t > m) m = t;
            if (m >= beta) return m;
        }
        return m;
    }
}
```

---

[Section Index] [Previous]

---

Top of Page

---

Comments or suggestions are always welcome.
Dr. Dennis Bahler /

**CSC 520-001/601**

# Introduction to Artificial Intelligence

**Fall 2015**

[Back to Moodle] [Back to Course at a Glance]

[Section Index] [Next]

## Outline

1. Representation and Reasoning Systems
2. Knowledge-Based Agents
    1. Knowledge Base
    2. Inference Engine
3. Comparison Metrics
    1. For Representations
    2. For Reasoning Methods
4. Types of Reasoning

# Representation and Reasoning Systems

AI systems, regardless of the details, can be viewed as consisting of two major components:

1. **Knowledge Representation**
2. **Reasoning**

**Knowledge Representation** (KR) refers to how relevant knowledge, facts, and relationships describing the world can be represented so they can be used by an automated agent.

There are numerous popular forms of KR:

- Logic: propositional, predicate, temporal, fuzzy, probabilistic, etc. etc.
- Networks of all kinds, including so-called Semantic Networks
- Frames
- Semantic Web: RDF, OWL, XML, etc.
- and for humans, English, Swahili, Urdu, etc.

**Reasoning** refers to the formal manipulation of symbols representing a set of believed concepts to produce representations of new concepts.

Examples of Reasoning:

- Logical inference
- Temporal reasoning

- Graph/subgraph matching
- Analogical reasoning
- Probabilistic inference
- etc.

Representation and reasoning are highly interdependent. Choices of representation will usually place major restrictions on the type of reasoning that can be employed, and vice versa.

Some people use the term **Representation and Reasoning System** (RRS) as a synonym for intelligent agent, because all intelligent agents must employ some form of both representation and reasoning.

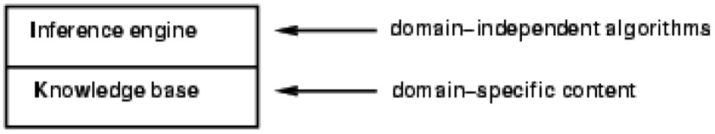
There are strong analogies in other fields, including mainstream computer science:

1. Data Structures
2. Algorithms

---

# Knowledge-Based Agents

A **knowledge-based agent** is an agent that is explicitly structured into a representation component and a reasoning component. There are synonyms for both representation and reasoning:




## Knowledge Base

In such agents, the knowledge base is the representational component. The knowledge base is often further subdivided into **facts** and **rules**. Facts are atomic units of representation, while rules (usually of the form IF...THEN...) can be used during reasoning to construct new facts beyond those originally in the knowledge base.

The knowledge base is a set of sentences that describe the world and its behavior in some language. Knowledge bases are typically highly domain-specific, although large knowledge corpuses exist as general knowledge resources (one example being Cyc).

There are two common operations on a knowledge base:

- Add new statements to the KB
- Query the KB

These operations are performed by the inference engine.

## Inference Engine

In knowledge-based agents, the inference engine is the reasoning component. The inference engine is that part of the agent that performs tasks such as pattern-matching to determine which rules have their IF part satisfied; organizing, reorganizing, and indexing the knowledge base as appropriate; choosing one or more rules to execute; and executing rules, incorporating the resulting new facts into the knowledge base.

In most such agents, the inference engine is domain-independent, although there are exceptions.

Inference engine operations generally require search, sometimes a lot of it.

---

# Comparison Metrics

## For Representations

1. **Representational Adequacy:** the ability to represent sufficient kinds of knowledge in the domain
2. **Acquisitional Efficiency:** the ability to easily and quickly represent new knowledge in the formalism

## For Reasoning Methods

1. **Inferential Adequacy:** the ability to manipulate representational structures to derive sufficient new ones
2. **Inferential Efficiency:** the ability to derive new representational structures quickly

---

# Types of Reasoning

| | Input | Input | Output | Examples |
|---|---|---|---|---|
| Deduction | FORALL X drunk(X) => weave(X) | drunk(john) | weave(john) | Resolution, Knowledge-based Systems, Expert Systems,... |

There are two other major types of reasoning besides deduction. We will get to both of them later.

---

[Section Index] [Next]

Top of Page

---

**CSC 520-001/601**

# Introduction to Artificial Intelligence

**Fall 2015**

---

[Back to Moodle] [Back to Course at a Glance]

[Section Index] [Previous] [Next]

---

## Outline



---

# Representation I: Propositional Logic (PL)

## Syntax

Propositional Logic language:

1. Set P of propositional symbols: p, q, r, ...
2. Two truth values: t, f
3. Set of logical symbols: $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$, $\neg$, $\square$
4. Various punctuation: ( ), [ ], { } etc.

Def: A **Proposition** is:

1. A propositional symbol, p
2. The negation of a proposition, $\neg$p
3. The disjunction of two propositions, $p_1 \vee p_2$
4. The conjunction of two propositions, $p_1 \wedge p_2$
5. An implication between two propositions, $p_1 \Rightarrow p_2$
6. A logical equivalence between two propositions, $p_1 \Leftrightarrow p_2$
7. ( p ) etc.
8. t
9. f

Note that this is a recursive definition, which refers to a proposition in 2 through 7.

---

## Semantics

Def: An **Interpretation** for a proposition using symbol set P is a mapping I:P → {t,f} (i.e., t or f for every symbol p in P)

In PL, a more common term for interpretation is Truth Value Assignment. For Propositional Logic, this is just a single row in a truth table.

Notation: For prop symbol p and interpretation I, I ⊨ p (read "I entails p") means p is assigned "true" in interpretation I, or equivalently I(p) = t. I ¬ ⊨ p means I(p) = f.

Def: **Semantic Entailment ( ⊨ )** (p, q are propositions)

For any interpretation I:

1. I ⊨ t
2. I ¬ ⊨ f
3. I ⊨ (p ∧ q) iff I ⊨ p and I ⊨ q
4. I ⊨ (p ∨ q) iff I ⊨ p or I ⊨ q
5. I ⊨ (p ⇒ q) iff I ¬ ⊨ p or I ⊨ q
6. I ⊨ (p ⇔ q) iff ( I ⊨ p and I ⊨ q ) or ( I ¬ ⊨ p and I ¬ ⊨ q )
7. I ⊨ ¬p iff I ¬ ⊨ p
8. I ⊨ ( p ) etc. iff I ⊨ p

Def. A **model** for a proposition is an interpretation in which the proposition is true.

Def. A proposition p is **valid** if for any interpretation I, I ⊨ p, i.e., every interpretation is a model.

Def. A proposition p is **satisfiable** if there exists an interpretation I such that I ⊨ p, i.e., at least one interpretation is a model.

Def. A proposition p is **unsatisfiable** if p is not satisfiable, i.e., no interpretation is a model.

Thm. A proposition p is valid iff ¬p is unsatisfiable.

---

Exercise: Use a truth table to show $(p_1 \Rightarrow p_2) \Leftrightarrow (\neg p_1 \vee p_2)$

| p1 | ⇒ | p2 | ⇔ | ¬p1 | ∨ | p2 |
|---|---|---|---|---|---|---|
| t | t | t | t | f | t | t |
| t | f | f | t | f | f | f |

| | | | | | | |
|---|---|---|---|---|---|---|
| f | t | t | t | t | t | t |
| f | t | f | t | t | t | f |

Column under "⇔" is true regardless of interpretation (i.e., valid). QED.

---

## English to PL and PL to English

Translating from natural language such as English into logical statements can be tricky because natural language is filled with ambiguities, and the same concept or relationship can be expressed in many ways.

For example, the simple implication $p \Rightarrow q$ can be expressed as

- p implies q
- if p, then q
- if p, q
- q if p
- p only if q
- q when p
- q whenever p
- q follows from p
- q unless not p
- q is necessary for p
- a necessary condition for p is q
- p is sufficient for q
- a sufficient condition for q is p

and so on.

---

## Handy Properties and Definitions

1. Commutativity of $\wedge$ and $\vee$
   - $(p_1 \wedge p_2) \Leftrightarrow (p_2 \wedge p_1)$
   - $(p_1 \vee p_2) \Leftrightarrow (p_2 \vee p_1)$
2. Associativity of $\wedge$ and $\vee$
   - $((p_1 \wedge p_2) \wedge p_3) \Leftrightarrow (p_1 \wedge (p_2 \wedge p_3))$
   - $((p_1 \vee p_2) \vee p_3) \Leftrightarrow (p_1 \vee (p_2 \vee p_3))$
3. Distributivity of $\wedge$ over $\vee$, and $\vee$ over $\wedge$
   - $(p_1 \wedge (p_2 \vee p_3)) \Leftrightarrow ((p_1 \wedge p_2) \vee (p_1 \wedge p_3))$
   - $(p_1 \vee (p_2 \wedge p_3)) \Leftrightarrow ((p_1 \vee p_2) \wedge (p_1 \vee p_3))$
4. DeMorgan
   - $\neg(p_1 \wedge p_2) \Leftrightarrow (\neg p_1 \vee \neg p_2)$
   - $\neg(p_1 \vee p_2) \Leftrightarrow (\neg p_1 \wedge \neg p_2)$

5. Complementation
    - $\neg(\neg p) \Leftrightarrow p$

Def. A propositional symbol p or its negation ¬p is called a **literal**.

Def. A pair of literals p and ¬p are called **complementary literals**.

Def. A **clause** is a proposition of the form $L_1 \vee L_2 \vee ... \vee L_n$, where the $L_i$ are literals. If n = 0 we have the **empty clause**, written as an empty box: □
The empty clause is always false.

Def. A **Horn Clause** is a clause with at most one positive literal. There are 3 types of Horn clauses:

1. **Unit Clause**: 1 positive literal, no negative literals
2. **Nonunit Clause**: 1 positive literal, remainder negative literals
3. **Negative Clause**: no positive literals, all negative literals

A Horn Clause that is not a Negative Clause is called a **Definite Clause**. A Definite Clause is a Horn Clause with exactly one positive literal.

It will turn out that, among Horn clauses, unit clauses correspond to facts, and nonunit clauses correspond to rules.

Solely to accomodate the inference procedure that we will use, it turns out to be necessary to simplify the syntactic representation of logical statements into what is called **normal form**.

Def. A proposition is in **Conjunctive Normal Form** (CNF) if it is of the form $C_1 \wedge C_2 \wedge ... \wedge C_m$, where each $C_i$ is a clause.

Thm. Any proposition can be put into CNF, i.e., for every proposition p there exists a proposition $p_2$ such that

1. $p_2$ is in CNF, and
2. $I \models p$ iff $I \models p_2$ for any interpretation I

If a proposition is not in CNF, it must be rewritten so that it is in CNF, without changing the semantics. This involves an algorithm that must get rid of all symbols except ¬ and ∨ and move negation symbols next to the atom(s) they're negating, by using the Semantic Entailment rules and the Handy Properties and Definitions in the right combination and order. Details for the more complex case of First-Order Predicate Logic are in R&N Ch. 9.

---

# Reasoning I: Propositional Resolution

Those who have formalized human logical reasoning, starting from the ancient Greeks, have identified numerous **rules of inference**.

Notation:

```
    *what we know*
--------------------
*what we can conclude*
```

A short list of human inference rules might include

- Addition,
- Simplification,
- Conjunction,
- Modus Ponens,
- Modus Tollens,
- Hypothetical Syllogism,
- Disjunctive Syllogism,
- Universal Instantiation,
- Universal Generalization,
- Existential Instantiation, and
- Existential Generalization.

That's 11 rules of inference.

In doing **automated** logical inference, by contrast, we are lucky; it turns out we only need one rule of inference.

---

## Rules of Inference

1. Modus Ponens

   $p$
   $p \Rightarrow q$
   ------
   $q$

2. Modus Tollens

   $p \Rightarrow q$
   $\neg q$
   ------
   $\neg p$

These are the only two we need (unlike, say, CSC 226). In fact, instead of using them directly, we can combine them, producing a single rule of inference known as **Resolution**.

Def. Suppose clause $C_1$ contains literal p, and suppose clause $C_2$ contains literal $\neg p$. Then a clause

$C = ((C_1 - \{p\}) \cup (C_2 - \{\neg p\}))$

can be obtained by **Resolution** of $C_1$ and $C_2$, and $(C_1 \wedge C_2) \Rightarrow (C_1 \wedge C_2 \wedge C)$ -- i.e., $C_1$ and $C_2$ are still around and still true.

C is called a **resolvent** of $C_1$ and $C_2$. In general, there may be more than one way to resolve $C_1$ and $C_2$.

Resolution can be viewed as a cancellation process incorporating Modus Ponens and Modus Tollens, with input in CNF.

p
¬p ∨ q
---
q

Here, the p and ¬p cancel, leaving the rest of each clause below the line as the conclusion.

¬p ∨ q
¬q
---
¬p

Here, the q and ¬q cancel, leaving the rest of each clause below the line as the conclusion.

¬a ∨ b ∨ ¬c ∨ d
a ∨ e ∨ ¬f
---
b ∨ ¬c ∨ d ∨ e ∨ ¬f

Here, the a and ¬a cancel, leaving the rest of each clause (OR'ed together) below the line as the conclusion.

---

## Automated Deduction Using Resolution

Resolution-based reasoning takes as input

1. A set of propositions in CNF (called a "theory", a set of clauses implicitly ANDed together)
2. The **negation** of a conclusion, also in CNF.

The inference mechanism resolves clauses pairwise, **using as resolvents clauses containing literals that are complementary,** until the empty clause is derived.

This is proof by contradiction: we show the conclusion C follows from the theory T (i.e., the implication T ⇒ C is valid) by deriving a contradiction (the empty clause, which is always false) from the union of T and the negated conclusion ¬C. (This simple version works only if the theory itself is internally consistent, i.e., if we cannot derive false from the theory alone.)

Symbolically, T ⇒ C is valid iff ¬(T ⇒ C) is unsatisfiable, and ¬(T ⇒ C) is equivalent to ¬(¬T ∨ C), which is equivalent to T ∧ ¬C.

---

## Handy Properties of Resolution

Def. **Derivability** (symbol: |- the "single turnstile")
A clause C is said to be **derivable** from a theory T (written $T \vdash C$) iff C can be derived from T using one or more rules of inference.

Def. A rule of inference is said to be **sound** if $T \vdash C \Rightarrow T \models C$ (anything derivable is semantically entailed)

(Notation abuse: T is assumed valid, so $T \models C$ is taken to mean any interpretation of T entails C.)

Def. A rule of inference is said to be **complete** if $T \models C \Rightarrow T \vdash C$ (anything semantically entailed is derivable)

Handy property of resolution: If T is a theory, C a clause, and resolution is the sole rule of inference, then $T \vdash C$ iff $T \models C$ (i.e., resolution is sound and complete). However, if $T \nvdash C$, resolution will never produce the empty clause, so termination must be determined heuristically. Some have called this property **semi-completeness.**

---

## The 4-Part Heuristic

Each iteration of resolution in general requires four choices:

1. Which clause $C_1$?
2. Which literal $L_1$ within $C_1$?
3. Which clause $C_2$?
4. Which literal $L_2$ within $C_2$?

Purely to make the needed choices more systematically, we will adopt heuristic guidelines for the four choices. Together these can be called **The 4-Part Heuristic**. For the most part we will use this heuristic in example proofs, even when it produces a proof that is longer than optimal.

1. Which clause $C_1$?
   The most recently generated clause that it is possible to use.
2. Which literal $L_1$ within $C_1$?
   The leftmost literal that it is possible to use.
3. Which clause $C_2$?
   The least recently generated clause that it is possible to use.
4. Which literal $L_2$ within $C_2$?
   The leftmost literal that it is possible to use, although if we restrict our representation to Horn Clauses, this choice is unnecessary, as there is only one such literal (WHY?).

It's worth pointing out, however, that before resolution can work, the knowledge base must be in Conjunctive Normal Form, and logical conjunction is commutative, so what is "most recent", "least recent", "leftmost", etc. is under nearly complete user control.

---

[Section Index] [Previous Index] [Next]

Top of Page

---


Comments or suggestions are always welcome.
Dr. Dennis Bahler /

**CSC 520-001/601**

# Introduction to Artificial Intelligence

**Fall 2015**

[Back to Moodle] [Back to Course at a Glance]

[Section Index] [Previous] [Next]

## Outline

1. PL Examples P1-P2
2. Problems with Propositional Logic

# P1: Example Propositional Resolution Proof (Derivation)

```
English:
Assume that if it's raining, then it's wet outside.
Assume that if it's Friday, then it's raining.
Assume that it's Friday.
Prove that it's wet outside.

Lexicon (meanings of symbols):
p == It's raining
q == It's wet outside
r == It's Friday
```

PL:

1. $p \Rightarrow q$
2. $r \Rightarrow p$
3. $r$

CNF:

| | |
|---|---|
| 1. $\neg p \vee q$ | Clause 1 of theory |
| 2. $\neg r \vee p$ | Clause 2 of theory |
| 3. $r$ | Clause 3 of theory |
| --- | |
| 4. $\neg q$ | Negated conclusion |
| --- | |
| 5. $\neg p$ | By resolution of 1 and 4 -- Step 1 of proof |
| 6. $\neg r$ | By resolution of 2 and 5 -- Step 2 of proof |

7. □ By resolution of 3 and 6 -- Step 3 of proof

---

# P2: A Simple Chemistry Example

```
Chemical Reactions:
1. MgO + H2 -> Mg + H2O
2. C + O2 -> CO2
3. CO2 + H2O -> H2CO3
Assuming sufficient quantities of MgO, H2, C, and O2,
prove that we can synthesize H2CO3.

Lexicon:
MgO, H2, Mg, H2O, C, O2, CO2, H2CO3 -- chemical compounds
```

PL:

1. MgO ∧ H2 ⇒ Mg ∧ H2O
2. C ∧ O2 ⇒ CO2
3. CO2 ∧ H2O ⇒ H2CO3
4. MgO
5. H2
6. C
7. O2

---

Conclusion

8. H2CO3

---

CNF:

1. ¬MgO ∨ ¬H2 ∨ Mg
2. ¬MgO ∨ ¬H2 ∨ H2O
3. ¬C ∨ ¬O2 ∨ CO2
4. ¬CO2 ∨ ¬H2O ∨ H2CO3
5. MgO
6. H2
7. C
8. O2

---

| | |
|---|---|
| 9. ¬H2CO3 | Negated conclusion |

---

| | |
|---|---|
| 10. ¬CO2 ∨ ¬H2O | 4 + 9 |
| 11. ¬C ∨ ¬O2 ∨ ¬H2O | 3 + 10 |

| | |
|---|---|
| 12. ¬O2 ∨ ¬H2O | 7 + 11 |
| 13. ¬H2O | 8 + 12 |
| 14. ¬MgO ∨ ¬H2 | 2 + 13 |
| 15. ¬H2 | 5 + 14 |
| 16. □ | 6 + 15 |

---

But resolution in general is a (hard) search problem. Much more about this when we talk about First Order Predicate Logic and Logic Programming.

---

## Problems with Propositional Logic

- Propositional Logic does not allow us to formulate general rules or represent relations
  - No "domain" variables
  - No quantification
- Solution:
  - Introduce variables and quantification: **First Order Predicate Logic**

---

[Section Index] [Previous] [Next]

Top of Page

---