Module 1: Artificial Intelligence Introduction and Overview

Module 1 Study Guide and Deliverables

Module Theme:	Introduction and Agents; Searching for Solutions; Heuristics; The A* Algorithm
Readings:	• Module 1 online content • Russell & Norvig Chapter 1 (Introduction), concentrate on Section 1.1 • Russell & Norvig Chapter 3 (Solving Problems) by Searching, concentrate on Section 3.1 • Russell & Norvig Chapter 4 (Search in Complex Environments), concentrate on Section 4.1
Assignments:	• Self-Introduction due Thursday, September 9, at 11:59 PM ET (not graded; access at “Class Discussion” on the left-hand course menu) • Lab 1 due Sunday, September 12, at 6:00 AM ET • Assignment 1, due Wednesday, September 15, at 6:00 AM ET
Live Classrooms:	• Wednesday, September 8, from 8:00 PM to 9:00 PM ET • Thursday, September 9, from 8:00 PM to 9:00 PM ET • Live Office: Wednesday and Thursday after Live Classroom, for as long as there are questions

Introduction and Agents

Learning Objectives

The word “Artificial Intelligence (AI)” has become ubiquitous. But what, exactly, does it mean? This part answers that question. A good portion of AI approaches problems from the perspective of ***agents***—autonomous objects like you and me. We may react to circumstances but what we do is up to each of us.

After successfully completing this part of the module, you will be able to do the following:

1. Describe objectives of AI.
2. Apply *agents*.

Introduction to AI

In this section, we will explain the history of two great currents in AI: logic and learning from data.

Brief History

Early History

The early history of AI was driven by attempts to capture and apply "intelligence." Much effort was expended on identifying what intelligence actually was. It was thought to be largely our ability to reason.

An example is the way we prove theorems. When you try to automate this, you find yourself processing symbols, such as *the system knows* a = b *and it also knows* b = c, *so it should be able to infer* a = c. Symbolic processing is still very much with us; in fact, there has been a blossoming of "SAT solvers" which are quite capable of reasoning.

In the case of specialized fields such as the maintenance of sophisticated radars, there is a kind of shortcut to symbolic reasoning. It consists of transferring the specialized knowledge of experts to computers. These are called ***expert systems***.

An approach championed by John McCarthy and others was to identify basic principles of intelligence—hopefully not a large number of them. An example is the ***closed world assumption***: that an assertion not provable by what's known to the system can be assumed false. This is not a practical approach as of this writing.

Early AI (i.e., pre-Internet) systems suffered from lack of contact with the world outside themselves, and this influenced their design.

Early Example: Blocks World

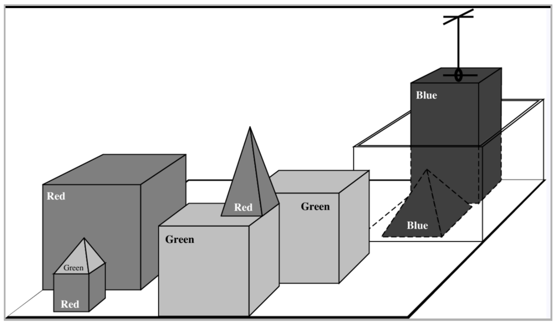
The *Blocks World*, consisting of real or simulated blocks of various shapes, was often used as a demonstration environment for early reasoning systems.

Loading [Contrib]/a11y/accessibility-menu.js

For example: *you can't place a block on one with a pointed top*.

Blocks World is a useful playground in which to develop planning algorithms.

Blocks World

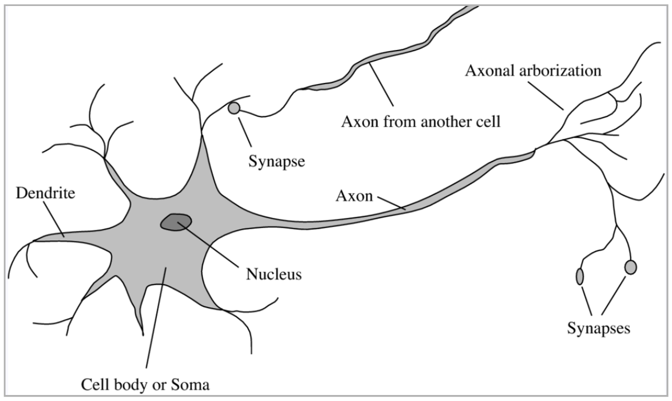


Source: Russel & Norvig

Midway On: Neuron

Many animals have neurons, the main constituents of human brains. These are cells that accept electro-chemical input from other neurons (via dendrites), process these inputs (in the soma), and send output to other neurons. The gaps between connected neurons are called synapses, and their strengths reflects the state of the brain. Artificial neural nets simulate this process.

Neuron



Source: Russel & Norvig

The field of neural nets has its origin with McCulloch and Pitts in 1943 but was not considered part of AI until the 21st century. Neural nets use the idea of brain neurons to fit functions to data sets. When Hinton et al showed in

Loading [Contrib]/a11y/accessibility-menu.js

2012 that deep learning could be effective, the field expanded rapidly into areas—such as natural language—than had been considered the purview of classical AI.

Contemporary

There is no "last word" on AI: discoveries are being made which could turn the tables once more. Approaches that seemed to have passed their heyday, such as fuzzy logic, could well surpass the best known current approaches. For example, humans do not seem to learn via massive amounts of data, as neural nets do. We appear to need only a few examples. Does this matter? We don't know. The fact that airplanes do not flap their wings as birds do has not prevented their success.

AI Topics

In this section, we will review the main topics of the course. Besides providing a setting, it will help in pointing you to a term project topic.

From the earliest days, AI has often been thought of a searching for solutions. We will cover those basics, including the A* algorithm. Another classical way to arrive at solutions is to do this in the context of ***constraints***—limits imposed on solutions. First order logic refers to the simplest kind of reasoning, which we will certainly discuss.

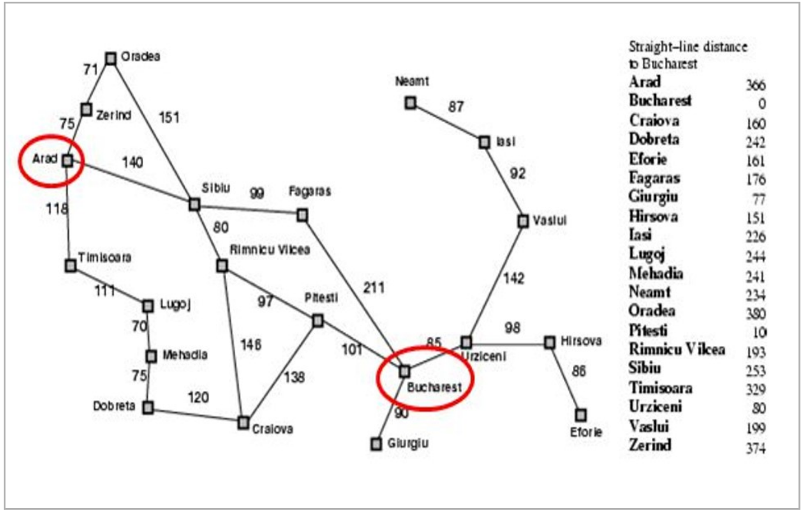
Searching for Solutions

- Define problem
- Perform (efficient) search
- Heuristics

A common example of searching for a solution is to find a route from one city to another. A central idea in AI is ***heuristics***—a way of proceeding that may not be bound to known mathematics but which arises from experience and from similarity with other problems.

Find a Route from One City to Another

Loading [Contrib]/a11y/accessibility-menu.js



Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

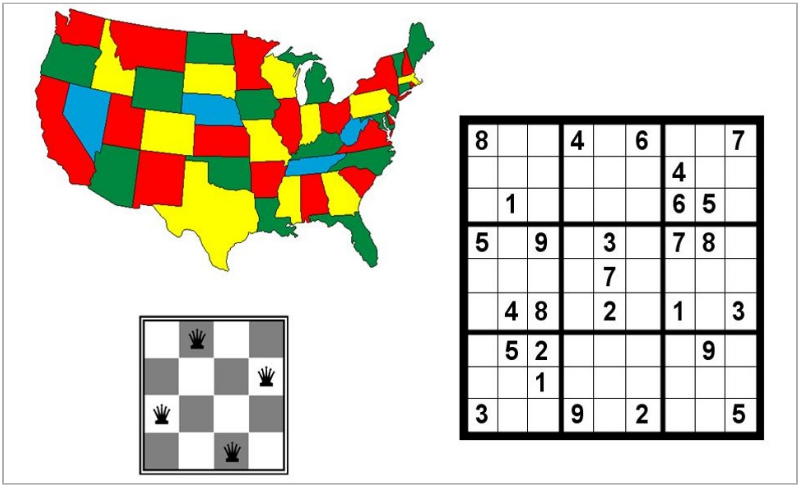
Source: A * search example

Constraint Satisfaction

Solve problem by successively “boxing in”.

Every problem context has constraints—conditions that the solution has to conform to. These limit what can and can not be solutions. For example, if we want to find values for \(x\) and \(y\), we can consider a required relationship as a constraint: for example, \(x + y = 3\). When there are many strong constraints, the solution may be entirely determined, such as \(y – x = 1\) (in addition). Constraints do not necessarily determine a unique solution but many AI problems do not have a (single) unique solution. The second constraint could have been \(y – x \le 1\), for example.

Sudoku, and chess piece placement, can be thought of as constraint satisfaction problems.



8			4		6			7
						4		
	1					6	5	
5		9		3		7	8	
				7				
	4	8		2		1		3
	5	2					9	
		1						
3			9		2			5

Source: Constraint satisfaction problems examples

Loading [Contrib]/a11y/accessibility-menu.js

Reasoning in First-Order Logic

"Intelligence" involves reasoning, which applies logic.

Propostional Logic – one last proof

- Show that \([p\land{(p\to{q})}]\to{q}\) is a tutology.
- We use \(\equiv\) to show that \([p\land{(p\to{q})}]\to{q}\equiv{T}\).

\(\begin{align} [p\land{(p\to{q})}]&\to{q}\\ \longrightarrow&\equiv{[p\land{(\lnot{p}\lor{q})}]}\to{q} &\text{substitution for} \rightarrow\\ \longrightarrow&\equiv{[(p\land{\lnot{p}})\lor{(p\land{q})}]}\to{q} & \text{distributive}\\ \longrightarrow&\equiv{[F\lor{(p\land{q})}]}\to{q} & \text{complement}\\ \longrightarrow&\equiv{(p\land{q})}\to{q} & \text{identity}\\ \longrightarrow&\equiv{\lnot{(p\land{q})}}\lor{q} & \text{substitution for} \rightarrow\\ \longrightarrow&\equiv{(\lnot{p}\lor{\lnot{q}})}\lor{q} & \text{DeMorgan's} \\ \longrightarrow&\equiv{\lnot{p}\lor{(\lnot{q}\lor{q})}} & \text{associative} \\ \longrightarrow&\equiv{\lnot{p}\lor{T}} & \text{complement} \\ \longrightarrow&\equiv{T} \end{align}\)

Source: https://image.slidesharecdn.com/c10logic1-121109070410-phpapp02/95/propositional-and-firstorder-logic-10-638.jpg?cb=1352445053

It is natural to think of logical reasoning as an important element of intelligence. In this example, we show ("prove" is a better term) that

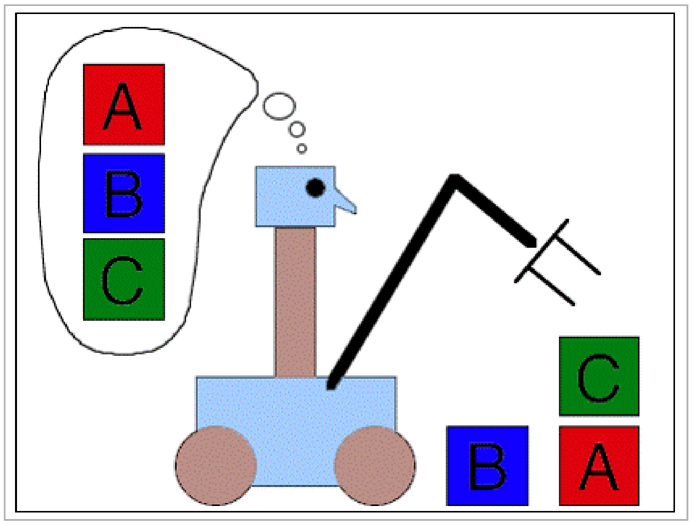
\([p\land{(p\to{q})}]\to{q}\) implies \(q\qquad\qquad(1)\)

regardless of what predicates p and q stand for. (For example "p" may stand for "there is a rabbit on Fifth Avenue.") We can see that if we know a fact ("p"), and we know that it always implies (results in) another fact q, then this is enough to conclude that q is true. We'd call that obvious but an AI system may not know this up front, whereas, it may know various more fundamental rules, and so it would have to conclude (1) from those rules. The illustration indicates the steps that the AI system would need to go through to establish it. We will return to first order logic later.

Planning

Generate steps to fulfill a goal (execution separate).

Planning is another classical AI area. We will explore it in depth during the course. The figure “Block Configuration” illustrates the problem of devising a plan to get to the block configuration in the lower right from the initial configuration at upper left. Planning is a lot like programming.



Source:
http://www.cs.colostate.edu/~cs540/spr2018/res_images/cs540logo.gif

Knowledge Representation

Practical coding (“data structures”)

The first issue that most AI applications have to deal with is how to represent the relevant artifacts. In the planning example, we need to define the data structure that species the beginning and ending block configurations, as well as those in between.

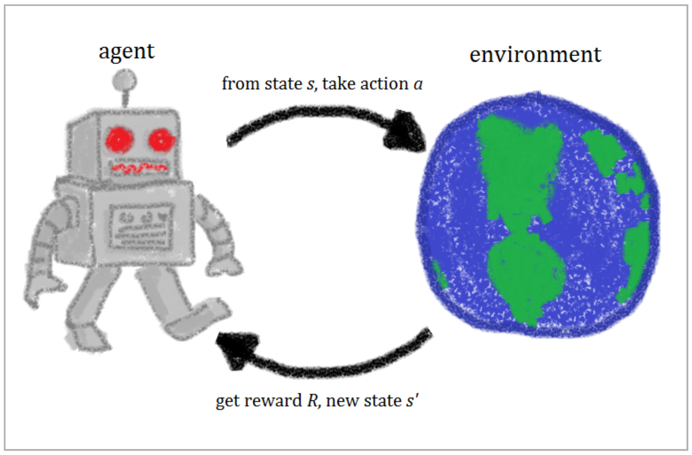
Machine Learning

Will review selected aspects (otherwise CS 767).

Machine learning has always been part of AI: in fact, a debate has raged for years as to whether any system can be considered “intelligent” is it does **not** learn. Neural nets have become the dominant way to do this. CS 767 is devoted entirely to machine learning. In this course, we will discuss the basics but we will also discuss techniques not covered in CS 767.

Loading [Contrib]/a11y/accessibility-menu.js ***forcement learning***, in which an agent interacts with the world around it, and learns from this by **reinforcing** beneficial actions and suppressing destructive ones. We will discuss agents in this part of Module 1.

Something beneficial found ⇒ code tendency.


agent
environment
from state s, take action a
get reward R, new state s'

Source: https://simple.wikipedia.org/wiki/Reinforcement_learning

Natural language processing (NLP) has also been a part of AI for many years, and has progressed steadily. It was though to be distinct from neural net methods but in recent years, neural net approaches have accelerated the effectiveness of NLP. We will discuss NLP in a separate unit.

Agents

"Intelligence" is often thought of as something that a being possesses. Beings behave within their environment in a kind of autonomous fashion—as opposed to being externally controlled. That's the idea of an ***agent***.

Agents: Definition and Example

Agents: A software entity with autonomy that causes actions.

Examples:

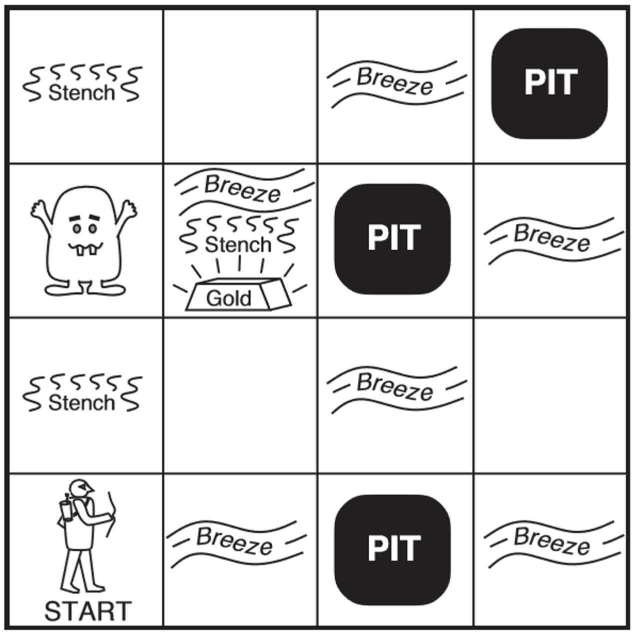
- Automated adversary in a video game
- Intelligent tutor

Loading [Contrib]/a11y/accessibility-menu.js

Wumpus World (R&N)

Russell and Norvig developed a simple environment in which to illustrate AI principles based on agents. They call this ***Wumpus World*** because it contains a troublesome "wumpus," of interest to the agent (shown with bow). Wumpus World enables demonstrations of reasoning—here, in the presence of stenches near the wumpus, and breezes near pits that trap the hunter agent, in a search for gold. Since a breeze delivers a stench, the agent is able to reason about what to do before experiencing the consequences of a decision.

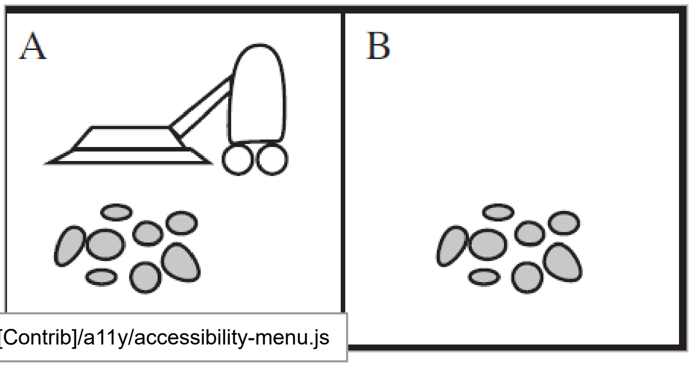
Wumpus World



Simplest Agent Example (R&N): Vacuum World

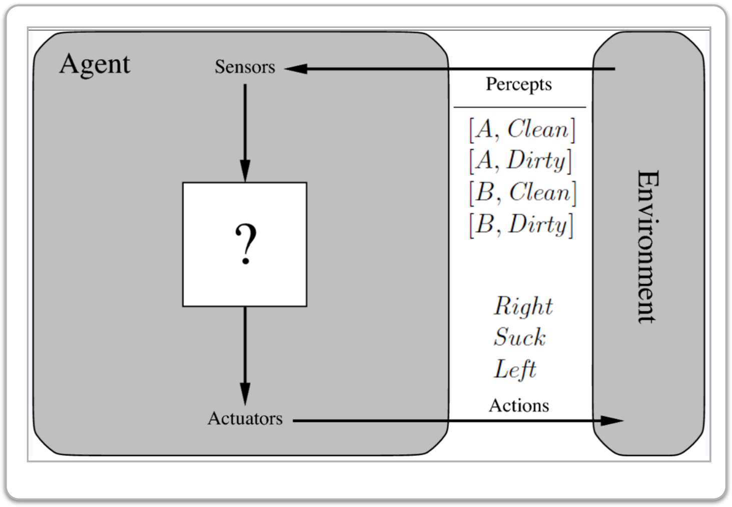
An even simpler example is ***Vacuum World***. The agent is an intelligent vacuum that lives in spaces A and B. This is its **environment**. In general, an agent interacts with an environment that's external to it.

Vacuum World



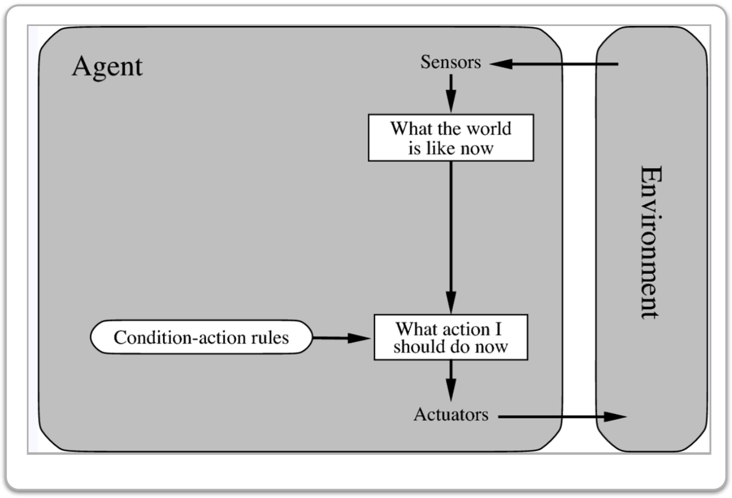
Loading [Contrib]/a11y/accessibility-menu.js

Agents: Architecture



A ***percept*** is a fact about the environment that an agent is capable of detecting. In the case of Vacuum World, there are four percepts ([A, Clean][A, Dirty][B, Clean][B, Dirty]), as shown. Agents are capable of detecting some, but not necessarily all percepts, and that's what makes this interesting.

Agents are capable of ***actions*** that may affect the environment. There are three (Right, Such, Left), as shown, in the case of Vacuum World.



A basic loop for an agent trying to fulfill a goal is the following:

1. Make an observation (a percept).
2. Add this percept to my existing sequence of percepts.
3. Select an action from the condition-action rules.
4. Perform the action.

Loading [Contrib]/a11y/accessibility-menu.js

Example of Select an Action: Burglary Simulation

The algorithm for selecting an action is thus at the heart of every agent design. An example, still at a high level, is shown in the figure “Burglar Behaviour: PECS”.

PECS:

- Physical Conditions
- Emotional States
- Cognitive Capabilities
- Social Status

In this case, action selection depends on the degree of the potential burglar’s need, emotion, and act of will at the time.

Source: http://www.geog.leeds.ac.uk/courses/other/crime/abm/example/index.html

Example: Percept to Action for Vacuum World

In the case of Vacuum World, a table below suffices. In principle, even this is infinite: this illustrates the need in AI applications to impose reasonable limits.

Percept Sequence	**Action**
[A, Clean]	Right
[A, Dirty]	Suck
[B, Clean]	Left
[B, Dirty]	Suck
[A, Clean], [A, Clean]	Right
[A, Clean], [A, Dirty]	Suck
…	…
[A, Clean], [A, Clean], [A, Clean]	Right
[A, Clean], [A, Clean], [A, Dirty]	Suck
…	…

Source: Russel & Norvig

Loading [Contrib]/a11y/accessibility-menu.js

ation: Modeling Burglary Occurrences

Bounded rationality: rationality is limited, when individuals make decisions, by the following factors:

- tractability of the decision problem
- cognitive limitations of the mind
- time available to make the decision

Source: https://en.wikipedia.org/wiki/Bounded_rationality

The following reference shows the resulting activities of several burglar agents: Agent Based Modelling Example.

Summary

The summary below lists the topics mentioned in this introduction. We will also discuss approaches to uncertainty, especially fuzzy logic.

- **Search** for Solutions
- Satisfy **Constraints**
- Use **reasoning** (typically First-Order Logic)
- **Plan**
- Represent Knowledge
- Handle **uncertainty**
- Have **Machines Learn** (e.g., Reinforcement Learning)
- Handle **Natural Language**
- **Agent** concept suits many situations

Search

Learning Objectives

A key perspective for AI an application is to interpret it as searching for a solution. For example, answering the question "How should I furnish my living room?"

A systematic search algorithm that's certain to yield all solutions is called brute force. Such algorithms do not, in general, account for efficiency. So they may be not only inefficient, but also unending. AI approaches (an many

Loading [Contrib]/a11y/accessibility-menu.js tter) are created to avoid suc disadvantages.

Many problems of “intelligence” can be thought of as searching for a solution to a give problem. These are common steps to carry this out. We will elaborate on the terms used.

After successfully completing this part of the module, you will be able to do the following:

1. Identify the role of searching in AI applications.
2. Achieve one or more of the following:
    - Apply heuristics and greedy methods.
    - Solve problems via constraints.
    - Apply A* algorithms.
    - Explain adversarial search.

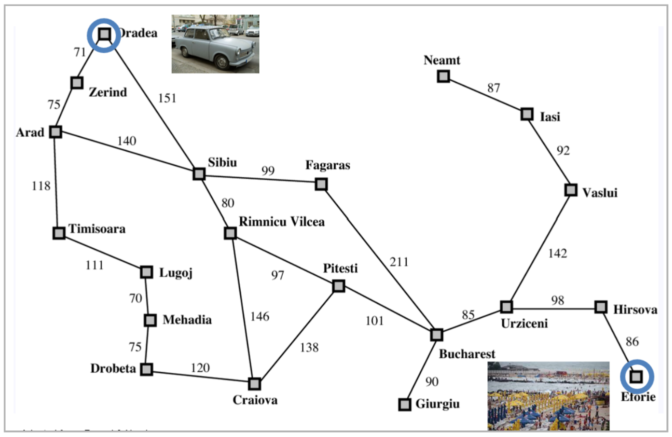
Agent Searching

In this section, we will view the search process from the perspective of an agent.

An Agent Searching

As an example, suppose that we want a route to the town of Eforie (Romania). We can view this as a car agent that’s “finding” a destination.

An Agent Searching



Source: Russel & Norvig

Example of Search: AI Math Tutor (Matt)

Another example of search is an intelligent Math tutor. This can be reduced to searching for the next screen to show the student.

Loading [Contrib]/a11y/accessibility-menu.js

What screen should Matt show next?

A common answer to the question “what (or who) is searching?” is … an agent.

The code below shows an outline of a search process. The assumptions are that the agent (*this* object) is in state *theState*, and the parameter *aPrecept* is not empty. The postcondition—the desired outcome—can’t be more specific than shown, in general.

The algorithm outline is in the form of accumulating code goals (specific objectives G1, G2, …). “Accumulating” means that, in fulfilling a code (implementation) goal, we maintain all those already attained.

The first is that *theState* reflects the true current state. The second code goal is that *actionSeq* = the sequence of actions that accomplish *agentGoal*, the agent’s next goal. This involves performing a search, given the problem formed from the current state and the agent’s goal. The second is to fulfill the postcondition by carrying out the actions in the sequence.

```
simple-problem-solving-agent (Perception aPercept){
/*
 * Preconditions: (1) this (agent) has state theState (2) aPercept!=null
 * Postcondition: Appropriate actions have been taken
*/
//G1: theState known
theState = updateState(theState, aPercept);

//G2: actionSeq is the sequence of actions that accomplish agentGoal
if(actionSeq empty){
             agentGoal = formGoal(goalState);
             problem = formProb(theState, agentGoal); // as in "obstacle"
         actionSeqG = search(problem);}  // ignore failure for now

//G3: Postcondition
actionSeq = removeFirst(actionSeq);
execute( …(actionSeq));
```

Loading [Contrib]/a11y/accessibility-menu.js

Classification Example 1/2

In this code outline, we explore the preceding code in the case that the percept is the appearance of a file.

```
aPerception: file encountered (or given)
Postcondition: Appropriate actions executed

theState == …

actionSeq == {…}

agentGoal == …
```

Classification Example 2/2

This code outline shows possible values. These could be simple constants, depending on the depth of the application.

```
aPerception: file encountered (or given)
Postcondition: Appropriate actions executed

theState == EXAMINING  (or CLASSIFYING, EXTRACTING)

actionSeq == {CLASSIFY X, EXAMINE X, EXTRACT X}

agentGoal == A COMPLETE CLASSIFICATION
```

Greedy

In this section, we will explore greedy search. You can think of these are local, opportunistic steps with an overall objective in mind.

Definition of Greedy Algorithms

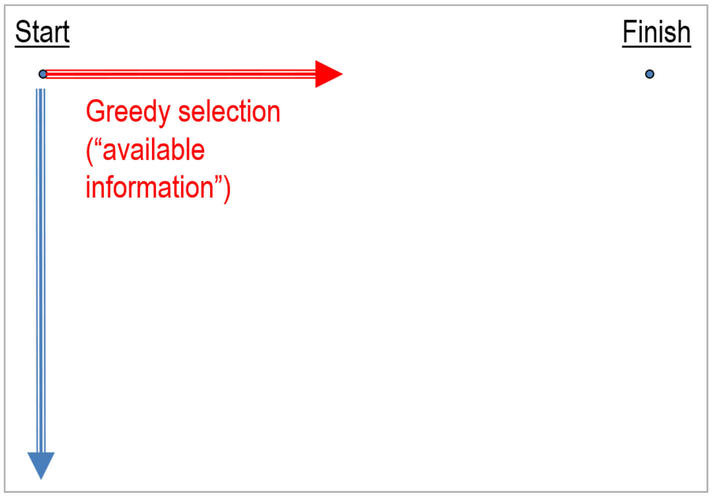
Greedy algorithms strive for **optimizing** (tending to optimization) rather than **optimized** (perfectly efficient) steps. Greedy algorithms do not attempt to retain past information to decide on the next step.

Loading [Contrib]/a11y/accessibility-menu.js

- Goal: “**optimizing**” solution—not necessarily true optimization (AI)
- **Use available (state) information**—(e.g., do not rely on recorded sub-problem solutions as in dynamic programming)

Example: Where Greed …

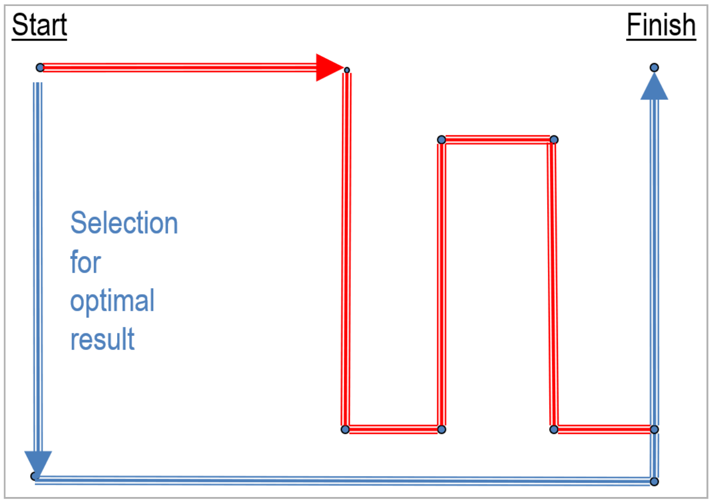
Greedy algorithms are local, and like all locally-made decisions, they can go seriously wrong. This is illustrated here by a car, that takes to fork pointing best to the destination.



Example: Where Greed Does Not Produce Optimality

The red route is not, after all, the best.

However, despite this danger, greedy search can be surprisingly successful in many cases.



Loading [Contrib]/a11y/accessibility-menu.js

Greedy Algorithms

The following code shows the structure of greedy algorithms. The first goal (*a*, which is maintained throughout) expresses that a solution is being built. The second goal (*b*) says that all changes to this solution-under-construction have been additive (i.e., there have been no subtractions or alterations to what's present). Goal (*c*) expresses completion.

```
SolutionType getOptimizingSolution(…)

   GOAL a (Parts): returnS is part of a solution

   [GOAL b] (Greed used): All changes to returnS
   have been additive and optimizing
   relative to the available information

   GOAL c (Complement): returnS is complete
```

Example: Requested Meetings

A classic example of a greedy search is to maximize the number of meetings in a room given a set of requests like those shown in the figure.

Loading [Contrib]/a11y/accessibility-menu.js

```
                Available time

0--[====)----------------------[==============)

0-----[============requested============)------

0-----[=======)-------------------[=========)--

0-----------[====)------[=========-)-----------

            Maximize number of meetings
```

Optimizing Solutions

Pick meeting that ends earliest—optimizes *number of meetings*

Or heuristic (AI):

“most of the important meetings”
…“most” “important?” ← optimizing

The question is *what is “greed” in picking the next meeting*? Is it “pick the one that starts the soonest from now” or something else? In short, there may be more than one greedy algorithm for a given problem. For this problem, it turns out that *picking the meeting that ends earliest* actually optimizes the solution. In general, greedy search may not be optimizing.

Constraint Satisfaction

Constraint satisfaction is a practical way to view search.

Using Constraints to Search

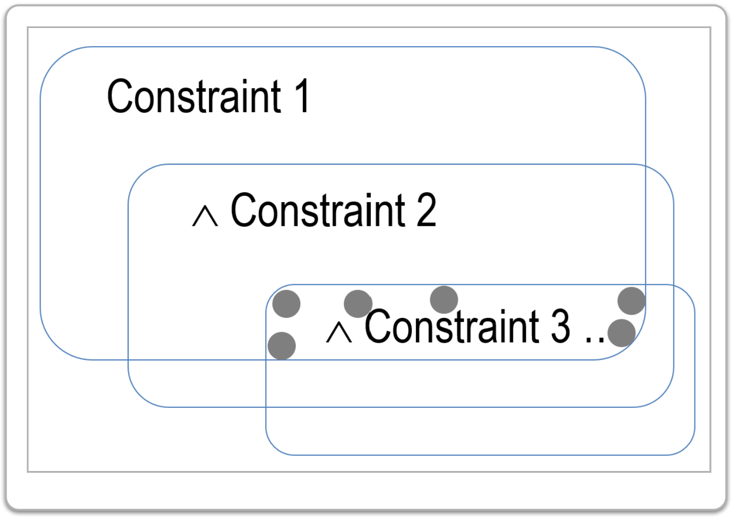
The idea of ***constraint satisfaction*** is to recognize and leverage the limits (constraints) of the needed search in the hope that they narrow the process down to a manageable number of alternatives. It’s the AI version of “we have no choice but to …”

Loading [Contrib]/a11y/accessibility-menu.js ngly bound the problem.

Satisfying Constraints ≡ Sub- … -sub states

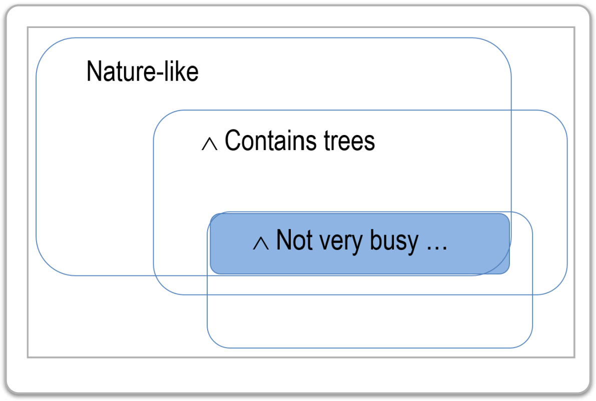
One can visualize a constraint by *the set of states (situations) that satisfy it*. And then implementing constraint satisfaction as an intersection of all the constraints, as in the figure.

A ***constraint*** = a set of relationships between variables.



Example Constraints

An example is setting an AI application to insert a splash screen based on user preferences such as *Nature-like* AND Contains trees AND *Note very busy* (i.e., with details). Imposing these successively focuses the search.



Loading [Contrib]/a11y/accessibility-menu.js

Example: Constraint Satisfaction Applied to Matt

Recall the search example discussed earlier: AI Math Tutor (Matt)—and the problem *what screen should Matt show next?*

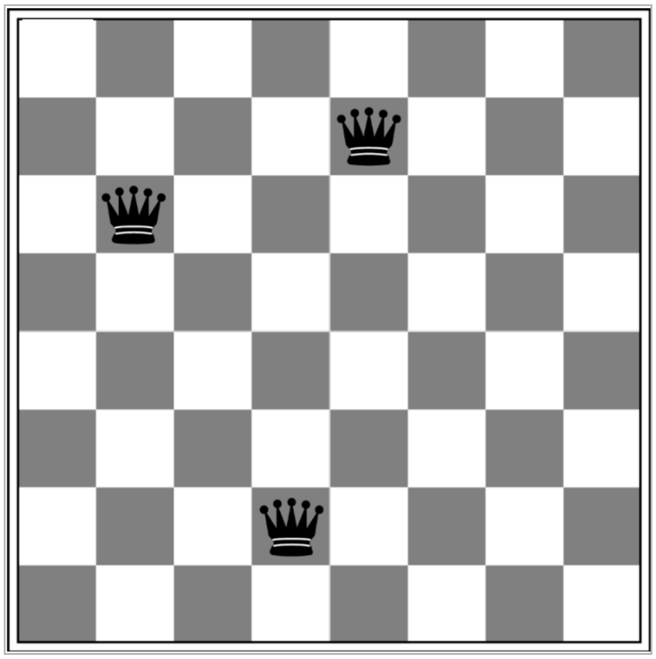
> Constraint 1: Student knows linear equations
> Constraint 2: Student knows exponents
> Constraint 3: Student does not know quadratic equations
> …
> What screen(s) satisfy these?

Matt identifies all screens that satisfy these constraints.

Example: 8-Queens Example

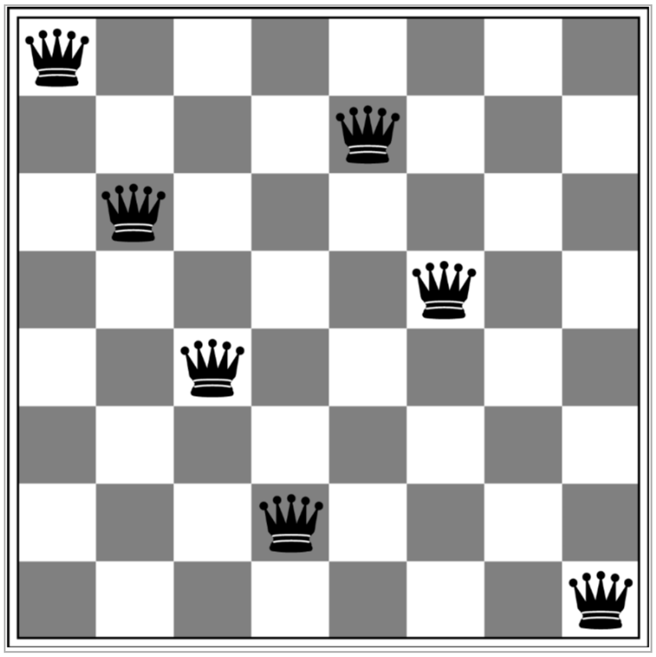
A classic toy example of constraint-based search is to find a stable deployment of 8 queens on a chessboard. “Stable” means that no queen threatens any other.

8-Queens Example 1



The figure shows only three such queens.

8-Queens Example 2



This figure shows 7 such queens. It is natural to think of searching for “the next queen for a given partial deployment.” However, this is not necessarily the most appropriate framing of the problem.

A good framing of the search is to search for all stable configurations on a limited board. This has the added advantage of recognizing that there is more than one stable configuration.

This is an example of **how framing the problem** is itself a key element of AI. Frame it one way, and the problem seem intractable; frame it another, and the problem seems solvable.

8-Queens Example 3: Classical Solution

Goal 1 (On partial board):
stable_n = all stable *n*-queen configurations on *n*-by-8 board.

Goal 2 (Complement): *n* = 8

To fulfill Goal 1 (easily), use *n*=1

These goals will be successful if a solution exists because …

if *s* is a stable configuration of *k* queens on an *n*-by-8 chessboard, there is a stable (*k*-1)-sized subset of *s* on an (*n-1*)-by-8 chess board.

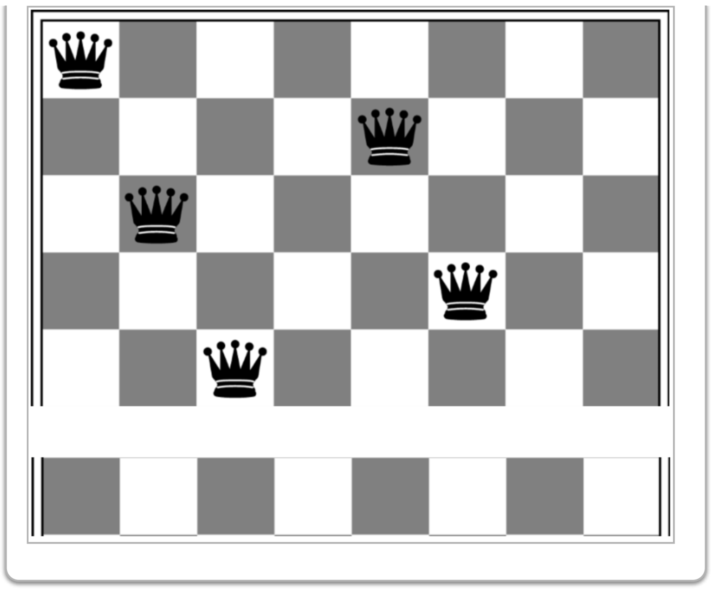
We thus make the first code goal to possess all stable *n*-queen configurations on *n*-by-8 board. By selecting 1 for *n*, this is easy to accomplish. A goal can be thought of as a constraint—as in **the process is constrained by having to have all stable *n*-queen configurations on *n*-by-8 board**.

The second goal, *n=8*, is accomplished by incrementing n and exploring every configuration in *stable_n* and every possible placement of a queen in the new row. The next figure shows this for *n=5*, one of the known stable configurations on 5-by-8, and the new row.

8-Queens Example 4: *n*x8 Board

n=5

Loading [Contrib]/a11y/accessibility-menu.js



A*

A* is a common approach to many searches.

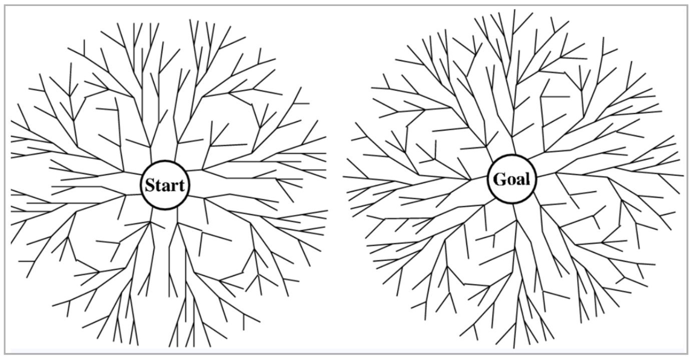
Classic Tree Search (non-AI)

We view the search space as a tree. Data structures already provide various brute-force searches such as breadth-first and depth first.

Bidirectional

If a goal state can be radiated backwards, then it is possible to generate paths from both directions. Unless we apply intelligent methods to this, however, they remain brute force. The best they can do is to reduce solution time by a half.

Loading [Contrib]/a11y/accessibility-menu.js



Tree Search

In theory, tree searches are “perfect.” However, they assume a well-defined objective, which may not be the case in the real world—Problem may not be well-defined, so tree search may be unacceptably slow.

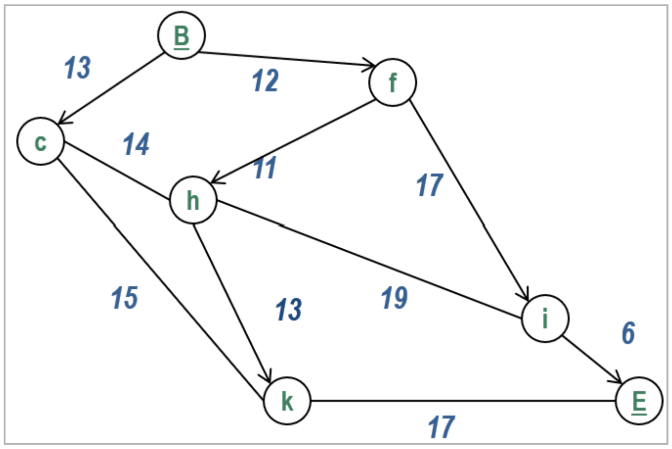
Example Objective

Consider the problem of finding the shortest path from one node to another in a weighted, directed graph.

Consider the problem of finding the shortest path from node \(B\) to node \(E\) in a weighted, directed graph.

This formulation expresses many problems, not just roads or flights. Nodes can, for example, be mental states for a recovering addict, and the cost of an edge the average number of months to get from one to the other.

Shortest Path in a Weighted, Directed Graph

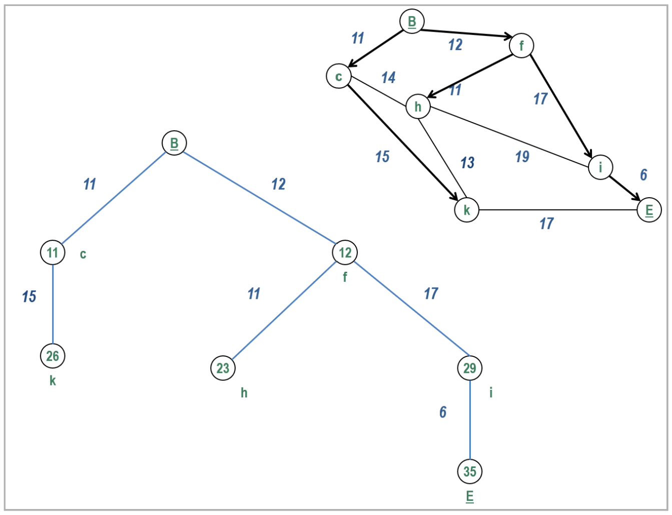


Loading [Contrib]/a11y/accessibility-menu.js

A* is a type of search in that selects among paths—more precisely, among path families defined by the root of a subtree.

The figure shows a search from a beginning node \(B\), intended to end at node \(E\). You can think of this as selecting from all root-to-leaf paths in the tree (which is conceptual—you don't actually build it).

Shortest Path through A* Search

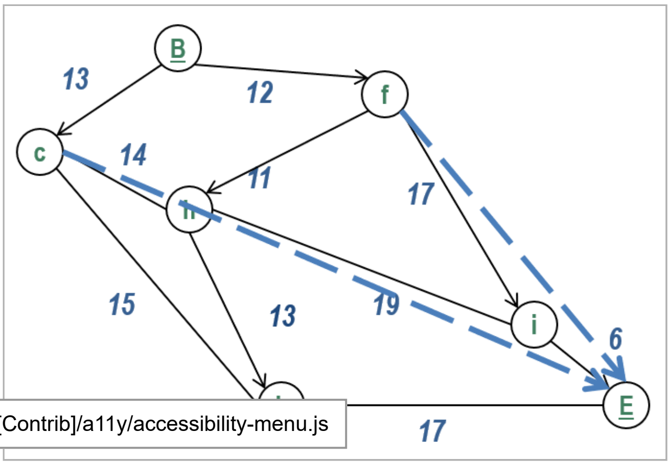


The Idea of A*

As is common in AI, a heuristic is used. But this kind of heuristic is somewhat unusual in that we can prove something about its effectiveness—and that's what makes A* noteworthy.

The heuristic used is *an estimate of the remaining cost of getting to the end node*. A graphic example of this, when the nodes are physical points in two dimensions, is the crow-flies distance to the destination. Two examples are shown in the figure (one example is from f to E and another example is from c to E). Observe that crow-flies distances are underestimates.

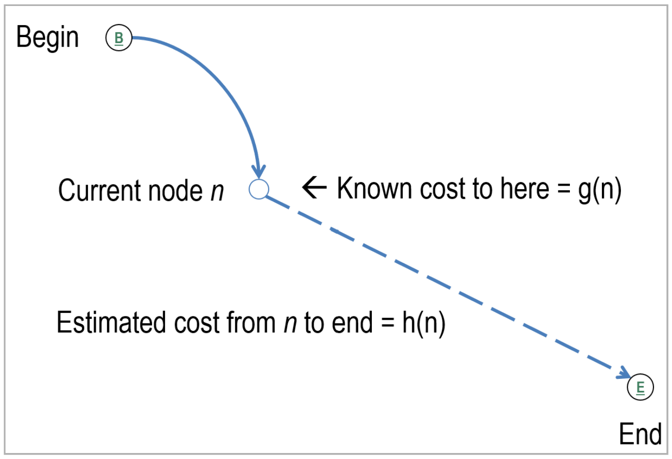
Utilize Heuristic of A*



Loading [Contrib]/a11y/accessibility-menu.js

We define notations for the cost, \(g(n)\) to get from node \(B\) to node \(n\), and the (heuristic) cost estimate \(h(n)\) of getting from \(n\) to \(E\). These are shown in the figure.

Utilize Heuristic of A*


Begin
B
Current node *n*
← Known cost to here = g(n)
Estimated cost from *n* to end = h(n)
E
End

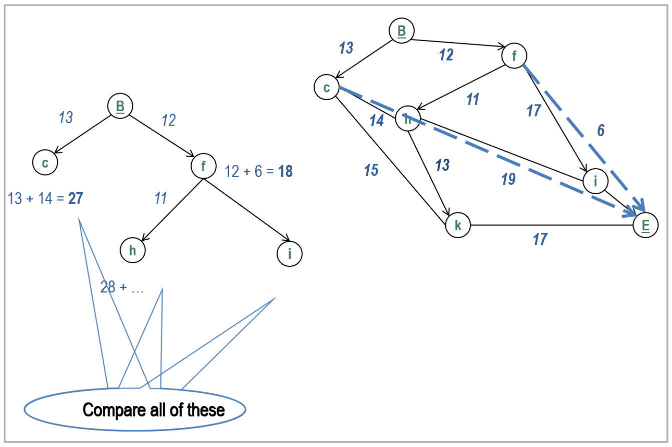
These are used to formulate \(f(n)\), *the estimated cheapest cost from Begin to End via node \(n\):*

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

The A* Algorithm

Upon each expansion, the node is expanded with the lowest value of \(f()\).

Utilize Heuristic of A*



See Russel & Norvig (2020) p. 87 for complete example.

The A* Theorem

The tree-search version of A* states that if \(h()\) underestimates the true cost, then expanding the node with the lowest value of \(f()\) produces an **optimal solution**.

A* produces optimal solution if \(h()\)

- never overestsimate—for tree search.
- satisfies triangle inequality—for graph search.

A* Algorithm Pre- and Postconditions

Here is the signature of aStar.

```
Tree aStar(Graph aGraph, Vertex aBeg, Vertex anEnd)
/*
Precondition 1: aGraph has only non-negative (edge) weights
Pre 2: There is an admissible heuristic h()
Pre 3: Triangle condition

Post: Tree returnT reflects the cheapest paths from aBeg
*/
```

A* Algorithm Outline

If we implement the following two goals, we'll be done..

```
//----Goal 1 (Globally Optimized)
// returnT is a tree of distances from aBegin AND
// The distances in returnT are optimal relative to aGraph

//----Goal 2 (Complete)
// returnT contains anEnd
```

Implementing the first goal is straightforward. (It involved altering *returnT*.)

```
//----Goal 1 (Globally Optimized)
// returnT is a tree of distances from aBegin AND
// The distances in returnT are optimal relative to aGraph

returnT = {cheapest f(n), n connecting to aBeg}
```

↑ This statement fulfills Goal 1.

Fulfilling Goal 2 while maintaining goal 1 can be accomplished as shown, but this requires proof that Goal 1 is indeed kept valid once the operations are complete. This follows.

```
//----Goal 2 (Complete)
// returnT contains anEnd


while anEnd not in returnT
        (t,v) = edge with t in returnT, v not, and f(v) minimal
        add (t,v) to returnT
```

Proof for A*

If there were a cheaper path, there would be an un-expanded node \(n\) on a different optimal path. But this means there is a “cheaper” node to be expanded at every step compared with the *under*estimate of any path through \(n\).

Although “cheaper” is based on estimates, it becomes an actual cost near the end of the process.

Adversarial Search

Adversarial search is the kind that takes place in the presence of an agent that’s trying to thwart your objectives. This is the case in games like tic-tac-toe.

Adversarial Search: MinMax Algorithm

Loading [Contrib]/a11y/accessibility-menu.js

Tic-Tac-Toe: “My” Turn and “My” most advantageous move(s)

Source: Russell, S & Norvig, P. (2020).

Tic-Tac-Toe: “Opponent’s” Most Advantageous Move(s)

We assume that the adversary will do the best we can imagine.

Source: Russell, S & Norvig, P. (2020).

Tic-Tac-Toe: From “My” Perspective

This is from “my” perspective. This is the essence of the MinMax algorithm (minimum penalty for me at my turn; maximum at my adversary’s turn).

Source: Russell, S & Norvig, P. (2020).

MinMax *Without* Pruning

MinMax Without Pruning

A two-ply game tree. The △ nodes are “MAX nodes,” in which it is MAX's turn to move, and the ▽ modes are “MIN nodes.” The terminal nodes show the utility values fro MAX; the other nodes are labeled with their minmax values. MAX's best move at the root is \(a_1\), because it leads to the state with the highest minmax value, and MIN's best reply is \(b_1\), because it leads to the state with the lowest minmax value.

Loading [Contrib]/a11y/accessibility-menu.js

Source: Russell, S & Norvig, P. (2020).

Alpha-Beta Pruning

Problem with minmax: exponential in depth of tree.

Can reduce by half: Prune to eliminate large parts of tree from consideration—branches that can't influence final decision.

When you apply this to a real-world problem—or even a nontrivial board game such as chess—the adversarial possibilities snowball. Pruning is a necessity, certainly for nodes that will ultimately have no influence.

Alpha-Beta pruning Illustration

In alpha-beta pruning, we bracket each potential move with *the minimum I can get* from the move and the maximum.

Notation: [*min I can get, max I can get*]

We then stop considering moves (and their descendants) where the bracket indicates no advantage. For example, there is no point in considering a node with [-\(\infty\), 2] (unlimited downside + max of 2) when I already have a [3, 3] course of action.

The following figure shows the pruning process completed.

Source: Russell, S & Norvig, P. (2020).

Search Summary

- AI: **scientific/empirical duality**
- **Greedy searching** is "powered by" **local heuristics**.

Loading [Contrib]/a11y/accessibility-menu.js

- **tion** "**boxes** solutions in".

- **A*** tree/graph search relies on optimistic heuristic.
- **Adversarial** searches can possibly be **pruned**.

Boston University Metropolitan College

Loading [Contrib]/a11y/accessibility-menu.js