

Artificial Intelligence

History

Could AI Stop This?



What is Artificial Intelligence?

A.I. is the study of how to make computers do things at which, at the moment, people are better.

Or, Stepping Back Even Farther, Can We Build Artificial People?

- Historical attempts
- The modern quest for robots and intelligent agents
- Us vs. Them

Historical Attempts - Frankenstein

The original story, published by Mary Shelley, in 1818, describes the attempt of a true scientist, Victor Frankenstein, to create life.



Frankenstein creates the fiend - illustration by Bernie Wrightson (© 1977)

Historical Attempts – The Turk



**Maelzel's
EXHIBITION,
MASONIC HALL.**

PERFORMANCE EVERY EVENING.

ON SATURDAY, MAY 17th 1834

There will be two exhibitions, one commencing at 4 o'clock in the usual time.—Doors open half an hour previous.

Doors open at half-past 7 o'clock. Performance to commence at 8 o'clock.

**PART FIRST.
THE ORIGINAL AND CELEBRATED
AUTOMATON
CHESS PLAYER.**

Invented by DE KEMPELIN, Improved by J. MAELZEL.

The Chess Player has withstood the first players of Europe and America, and excites universal admiration. He moves his head, eyes, lips, and hands, with the greatest facility, and distinctly pronounces the word "Solec," (the French word signifying "Check") when necessary. If a miss-move is made, he perceives and rectifies it.

**THE
Automaton Trumpeter.**

The Trumpeter is of a full size, and dressed in the uniform of the French Lancers. The pieces executed by this Automaton are performed with a distinctness and precision unattainable by the best living performer; the measurement of time being, from the nature of the mechanism, absolutely perfect. In double-tonguing, his superiority is particularly manifested, not only in the clearness of the tones, but also in the number of the notes which are sounded. All the sounds are actually produced in the Trumpet, there being no pipes whatever within the figure. The pieces he plays were written expressly for him by the first composers. He will perform on each evening, two favorite pieces. 1st, the French or Austrian Cavalry Manoeuvres. 2d, A March, with an accompaniment.

**THE
MECHANICAL THEATRE,**

Purposely introduced for the gratification of Juvenile Visitors.

IT CONSISTS OF THE FOLLOWING FIGURES:

- 1 The Amusing Little Bass Fiddler.
- 2 The French Oyster Women—who bows to the Company, and performs the duties of her station, by opening and presenting her Oysters to the audience.
- 3 The Old French Gentleman, of the ancient Regime, who drinks the health of the Company with great glee.
- 4 The Chinese Dancer, accompanying the Music with his Tambourine.
- 5 The Little Troubadour, playing on several instruments.
- 6 Punchinello will go through his comical attitudes in imitation of the celebrated Marquier.

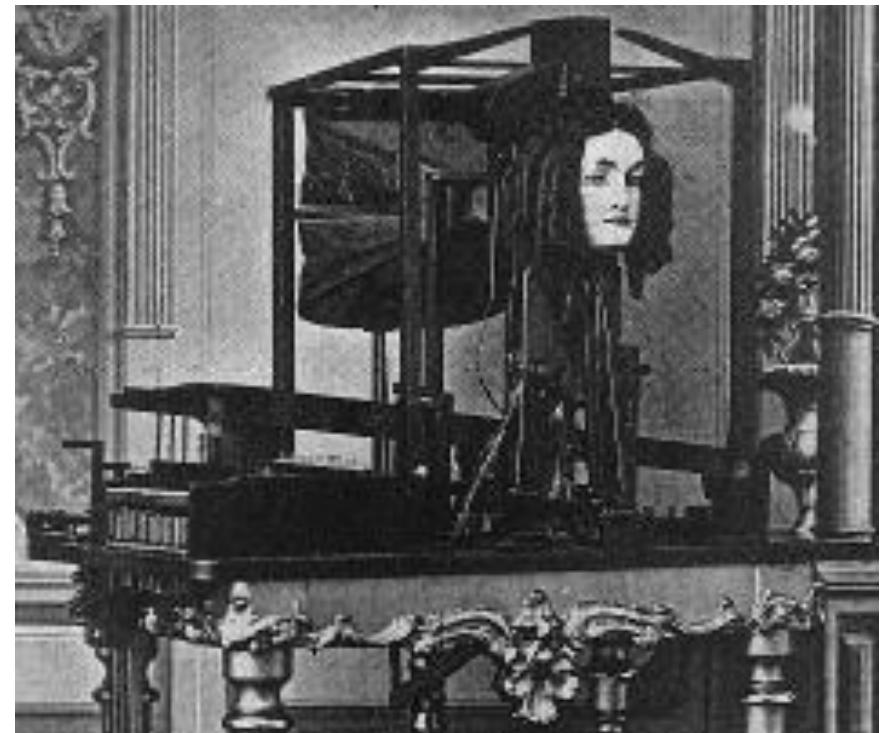


Historical Attempts - Euphonia

Joseph Faber's Amazing Talking Machine (1830-40's). The Euphonia and other early talking devices are described in detail in a paper by David Lindsay called "Talking Head", *Invention & Technology*, Summer 1997, 57-63.

About this device, Lindsay writes:

It is "... a speech synthesizer variously known as the Euphonia and the Amazing Talking Machine. By pumping air with the bellows ... and manipulating a series of plates, chambers, and other apparatus (including an artificial tongue ...), the operator could make it speak any European language. A German immigrant named Joseph Faber spent seventeen years perfecting the Euphonia, only to find when he was finished that few people cared."



From
<http://www.haskins.yale.edu/haskins/HEADS/SIMULACRA/euphonia.html>

Historical Attempts - RUR

In 1921, the Czech author Karel Capek produced the play *R.U.R.* (*Rossum's Universal Robots*).

"CHEAP LABOR. ROSSUM'S ROBOTS."

"ROBOTS FOR THE TROPICS. 150 DOLLARS EACH."

"EVERYONE SHOULD BUY HIS OWN ROBOT."

**"DO YOU WANT TO CHEAPEN YOUR OUTPUT?
ORDER ROSSUM'S ROBOTS"**

Some references state that term "robot" was derived from the Czech word *robo*ta, meaning "work", while others propose that *robo*ta actually means "forced workers" or "slaves." This latter view would certainly fit the point that Capek was trying to make, because his robots eventually rebelled against their creators, ran amok, and tried to wipe out the human race. However, as is usually the case with words, the truth of the matter is a little more convoluted. In the days when Czechoslovakia was a feudal society, "*robo*ta" referred to the two or three days of the week that peasants were obliged to leave their own fields to work without remuneration on the lands of noblemen. For a long time after the feudal system had passed away, *robo*ta continued to be used to describe work that one wasn't exactly doing voluntarily or for fun, while today's younger Czechs and Slovaks tend to use *robo*ta to refer to work that's boring or uninteresting.

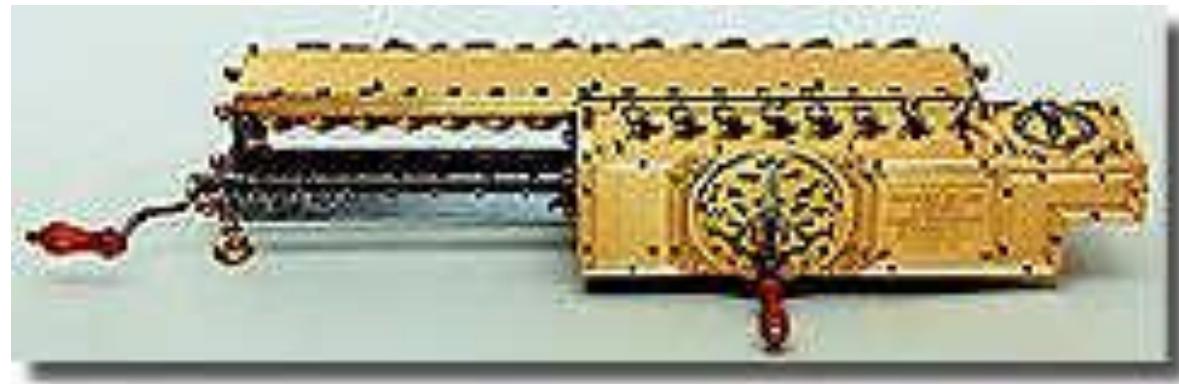
The Roots of Modern Technology

5thC B.C. Aristotelian logic invented

1642 Pascal built an adding machine



1694 Leibnitz reckoning machine

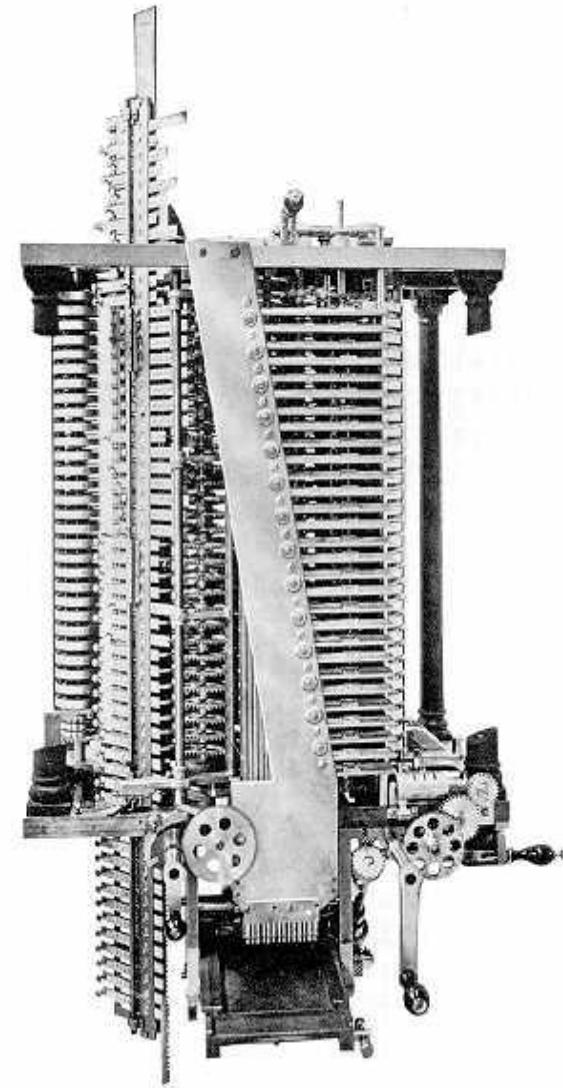


The Roots, continued

1834 Charles Babbage's
Analytical Engine

Ada writes of the engine, “The Analytical Engine has no pretensions whatever to originate anything. It can do whatever we know how to order it to perform.”

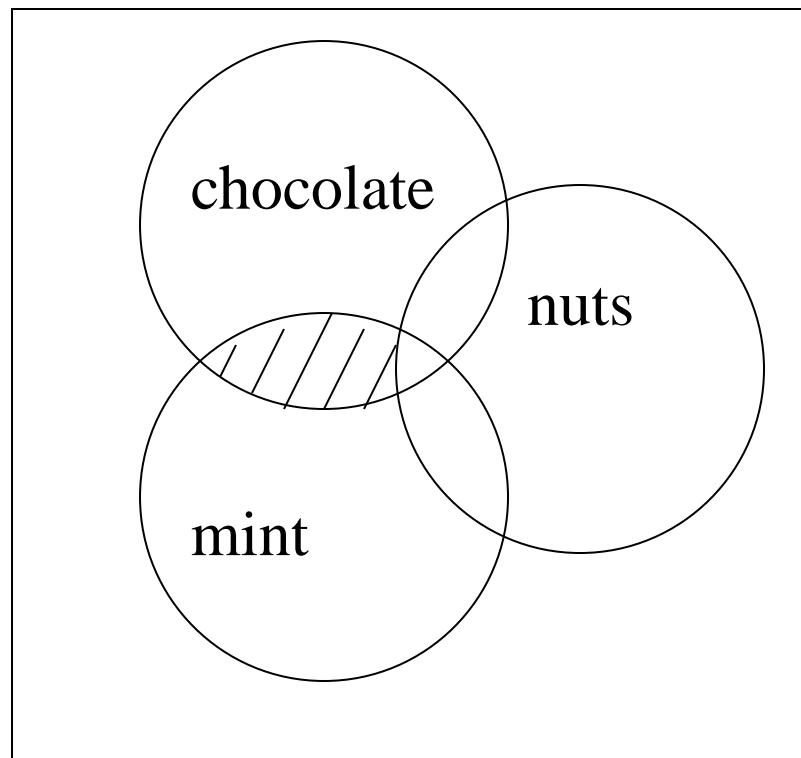
The picture is of a model built in the late 1800s by Babbage's son from Babbage's drawings.



The Roots: Logic

1848 George Boole *The Calculus of Logic*

chocolate and \neg nuts and mint



Mathematics in the Early 20th Century – (Looking Ahead: Will Logic be the Key to Thinking?)

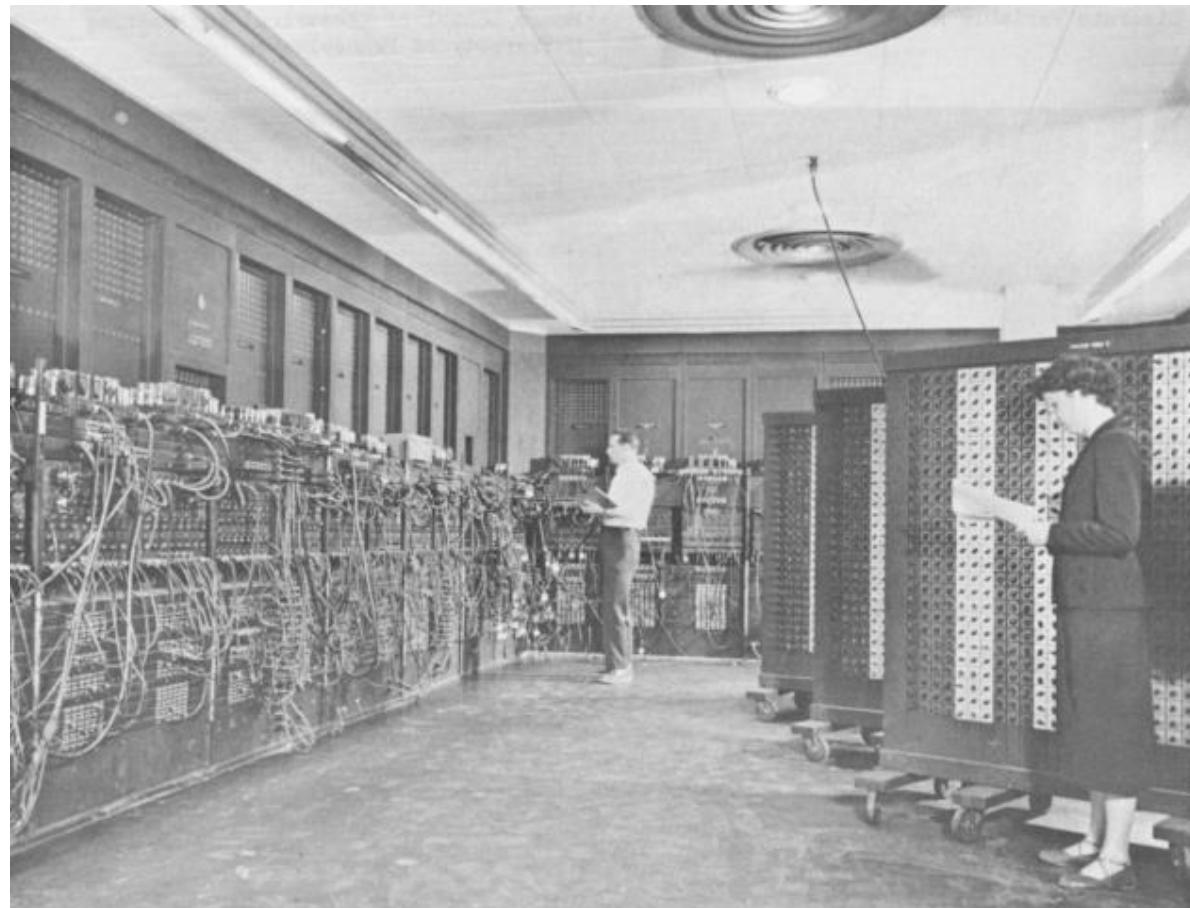
1900 Hilbert's program and the effort to formalize mathematics

1931 Kurt Gödel's paper, *On Formally Undecidable Propositions*

1936 Alan Turing's paper, *On Computable Numbers with an application to the Entscheidungs problem*

The Advent of the Computer

1945 ENIAC *The first electronic digital computer*



1949 EDVAC

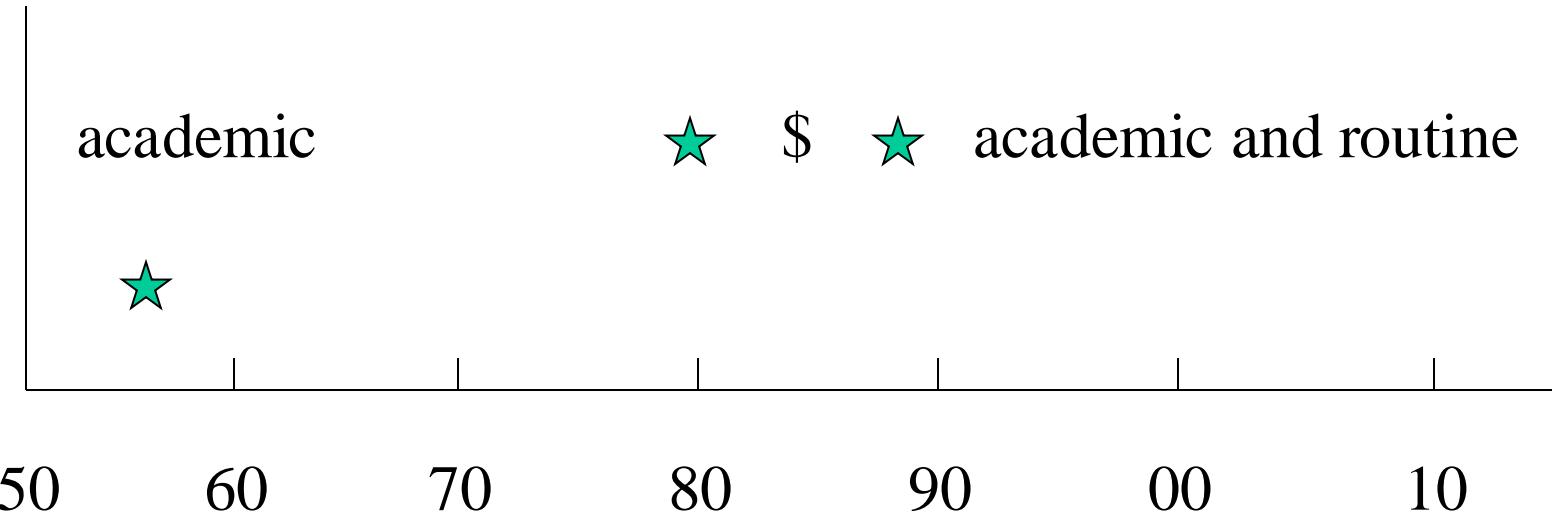
*The first stored
program computer*



The Dartmouth Conference and the Name Artificial Intelligence

J. McCarthy, M. L. Minsky, N. Rochester, and C.E. Shannon. August 31, 1955. "We propose that a 2 month, 10 man study of artificial intelligence be carried out during the summer of 1956 at Dartmouth College in Hanover, New Hampshire. The study is to proceed on the basis of the conjecture that every aspect of learning or any other feature of intelligence can in principle be so precisely described that a machine can be made to simulate it."

Time Line – The Big Picture



★ 1956 Dartmouth conference.

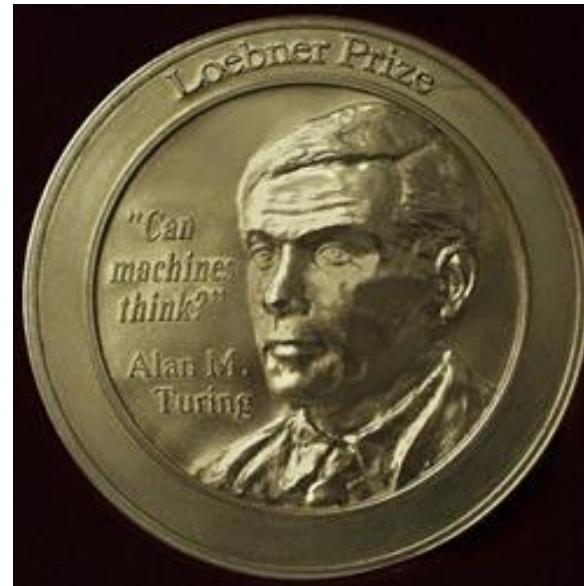
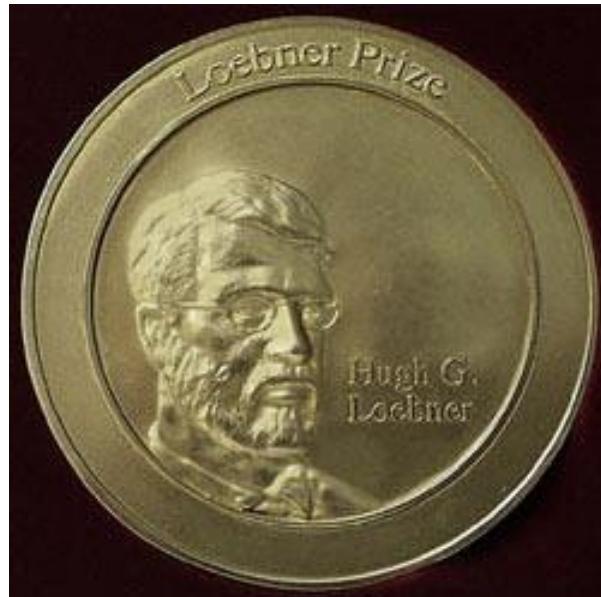
★ 1981 Japanese Fifth Generation project launched as the Expert Systems age blossoms in the US.

★ 1988 AI revenues peak at \$1 billion. AI Winter begins.

How Will We Recognize AI?

1950 Alan Turing's paper, *Computing Machinery and Intelligence*, described what is now called "The Turing Test". <http://www.abelard.org/turpap/turpap.htm>

1990 Loebner Prize established. Grand Prize of \$100,000 and a Gold Medal for the first computer whose responses are indistinguishable from a human.



The Origins of AI Hype

1950 Turing predicted that in about fifty years "an average interrogator will not have more than a 70 percent chance of making the right identification after five minutes of questioning".

1957 Newell and Simon predicted that "Within ten years a computer will be the world's chess champion, unless the rules bar it from competition."

Was Turing Right?

A conversation with Jabberwacky, 2003

Jabberwacky: Have you been to the Moon?

User: No. Few have. Have you?

Jabberwacky: Many a time my dear chap!

User: What was the most interesting thing you saw?

Jabberwacky: Fred.

User: Why did you find him interesting?

Jabberwacky: It just came into my head.

User: What sorts of things do you generally find interesting?

Jabberwacky: Nothing but smelly socks.

Another One

A Conversation with Alice, 2004

>What's the best place to eat Chinese food?

ALICE: My favorite places are California and Maine. Holland is cool too.

>What do you like about Maine?

ALICE: The special effects are the best.

Alice, Again

>Who invented paper towels?

ALICE: The Chinese, long before Europeans.

>Who invented pet rocks?

ALICE: The Wright Brothers.

>Who invented Barbie dolls?

ALICE: Thomas Edison.

Were Newell and Simon Right?

- 1950 Claude Shannon published a paper describing how a computer could play chess.
- 1957 Newell and Simon predicted that a computer will beat a human at chess within 10 years.
- 1967 MacHack was good enough to achieve a class-C rating in tournament chess.

Chess Today



In 1997, Deep Blue beat Gary Kasparov.



Why Did They Get it Wrong?

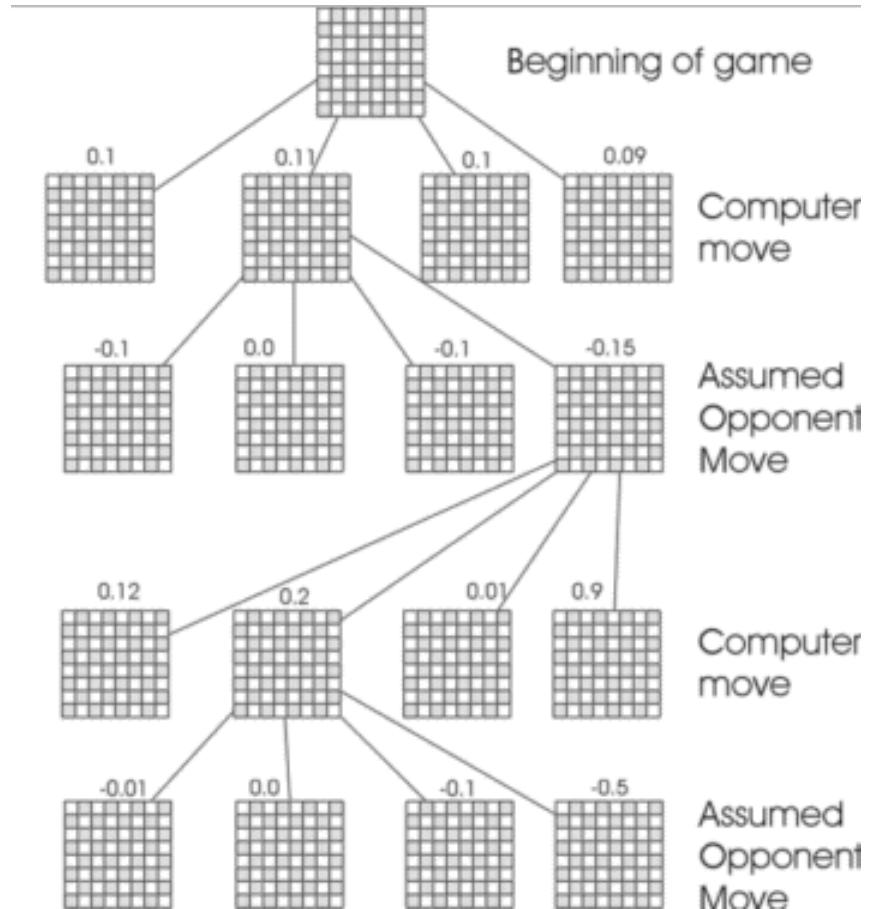
They failed to understand at least three key things:

- The need for knowledge (lots of it)
- Scalability and the problem of complexity and exponential growth
- The need to perceive the world

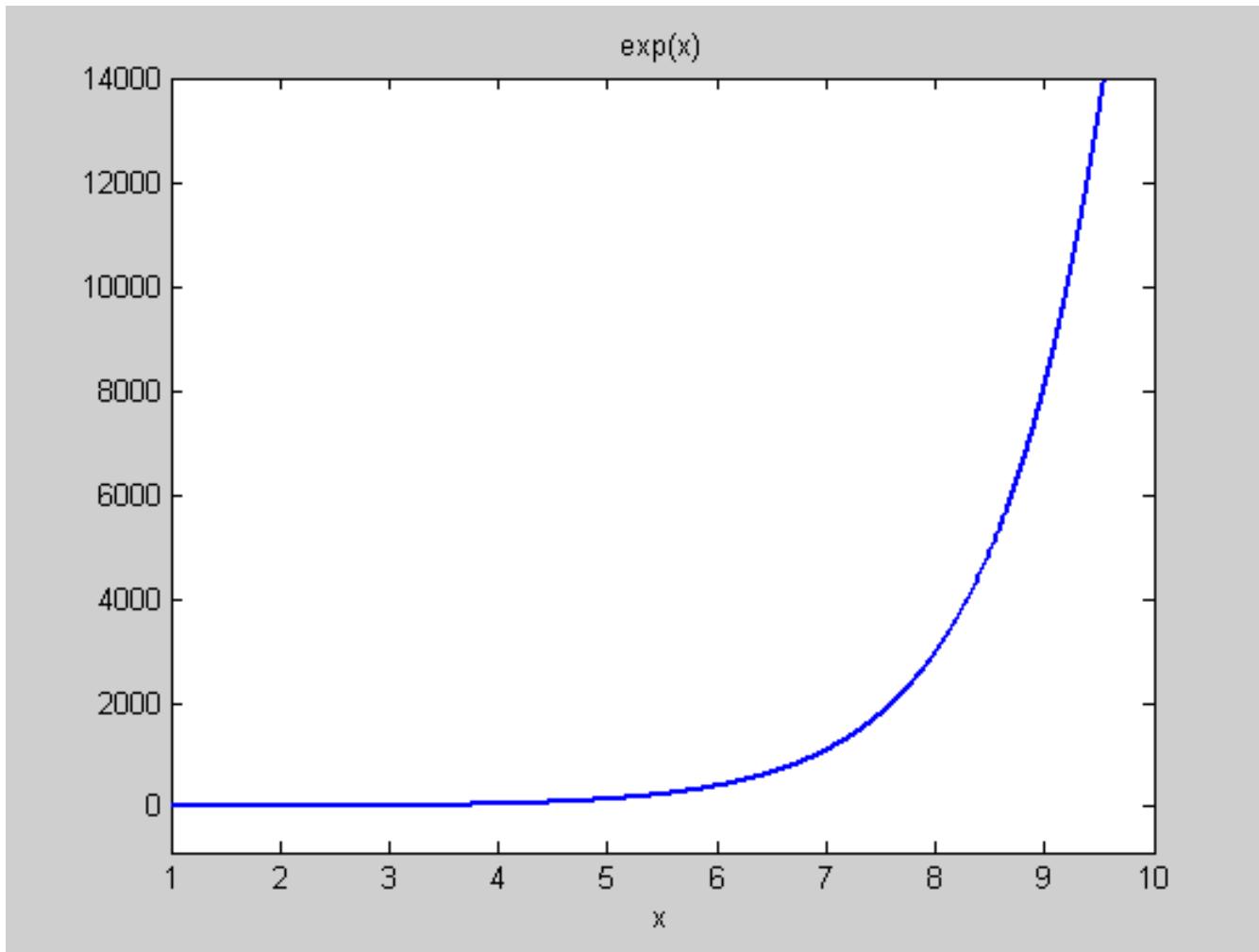
Scalability

Solving hard problems requires search in a large space.

To play master-level chess requires searching about 8 ply deep. So about 35^8 or $2 \cdot 10^{12}$ nodes must be examined.



Exponential Growth



But Chess is Easy

- The rules are simple enough to fit on one page
- The branching factor is only 35.

A Harder One

John saw a boy and a girl with a red wagon with one blue and one white wheel dragging on the ground under a tree with huge branches.

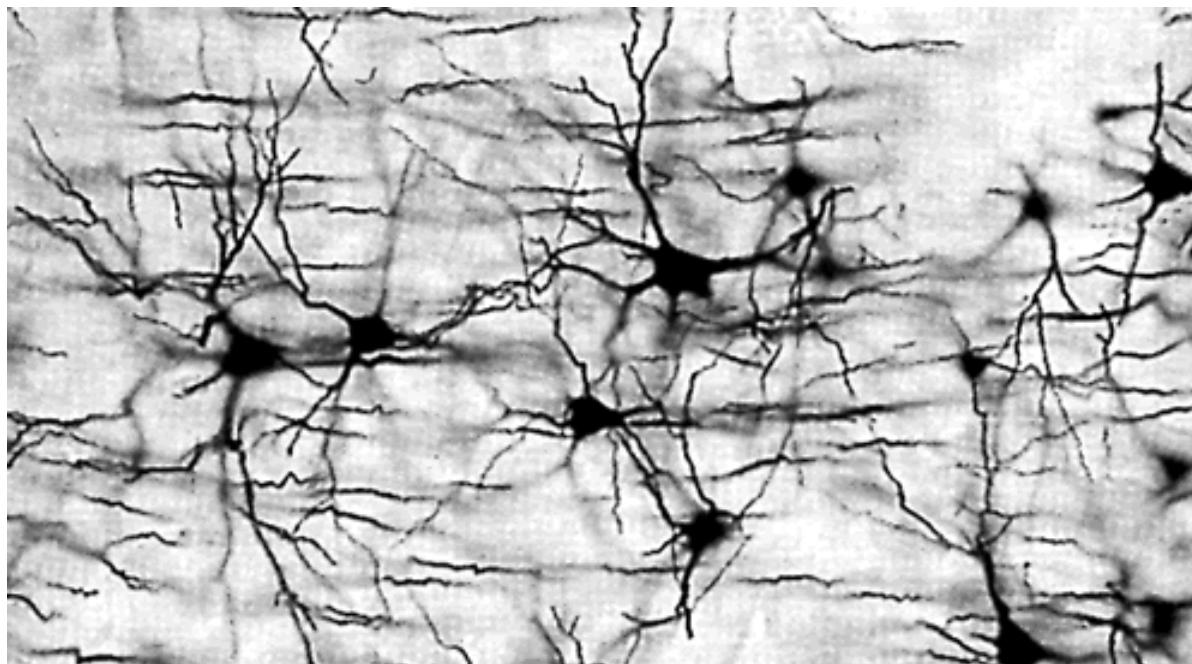
How Bad is the Ambiguity?

- Kim (1)
- Kim and Sue (1)
- Kim and Sue or Lee (2)
- Kim and Sue or Lee and Ann (5)
- Kim and Sue or Lee and Ann or Jon (14)
- Kim and Sue or Lee and Ann or Jon and Joe (42)
- Kim and Sue or Lee and Ann or Jon and Joe or Zak (132)
- Kim and Sue or Lee and Ann or Jon and Joe or Zak and Mel (469)
- Kim and Sue or Lee and Ann or Jon and Joe or Zak and Mel or Guy (1430)
- Kim and Sue or Lee and Ann or Jon and Joe or Zak and Mel or Guy and Jan (4862)

The number of parses for an expression with n terms is the n 'th Catalan number:

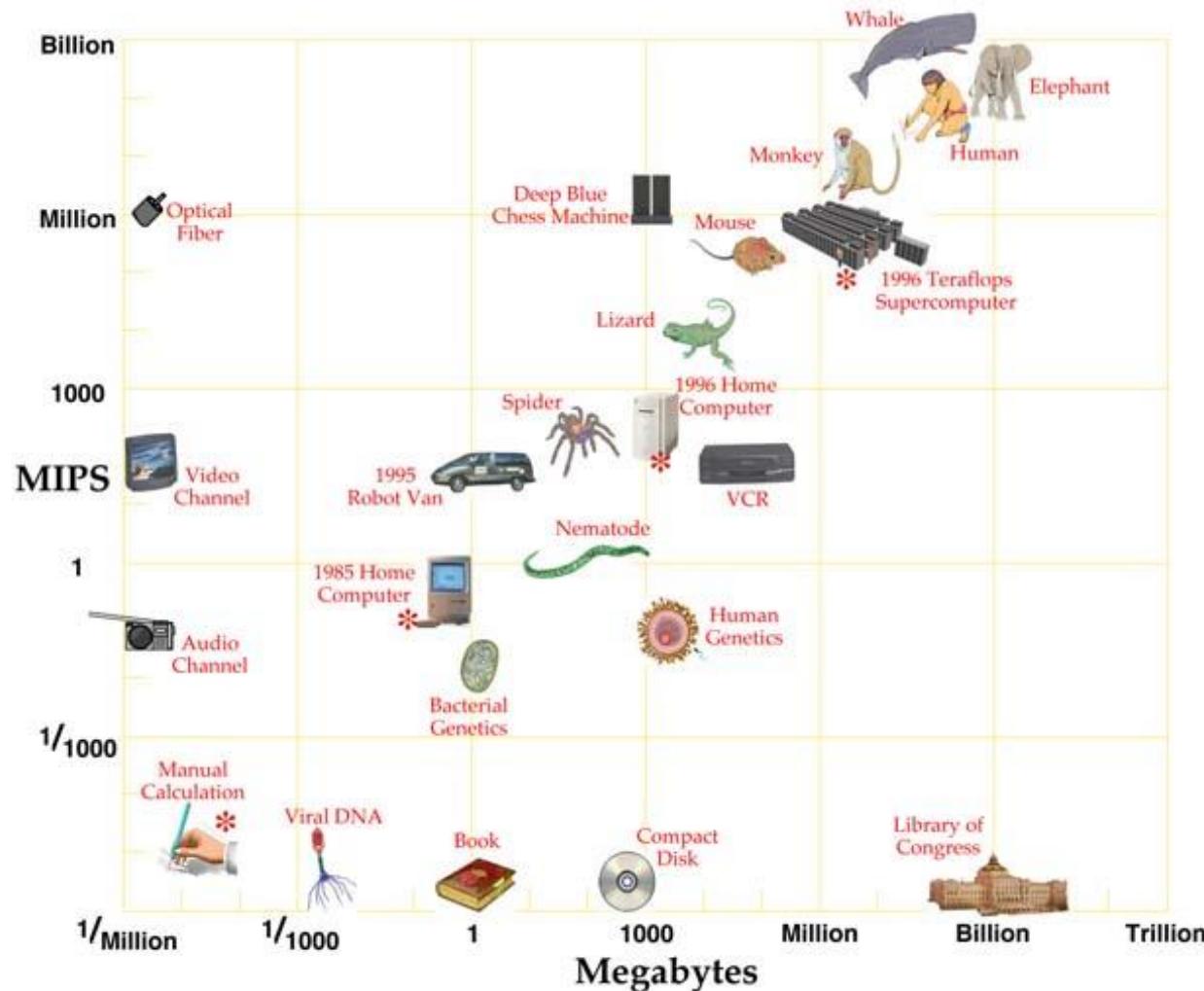
$$Cat(n) = \binom{2n}{n} - \binom{2n}{n-1}$$

Can We Get Around the Search Problem ?



How Much Compute Power Does it Take?

All Thinks, Great and Small



From Hans Moravec, Robot Mere Machine to Transcendent Mind 1998.

How Much Compute Power is There?

Evolution of Computer Power/Cost

MIPS per \$1000 (1997 Dollars)

Million

1000

1

1/1000

1/Million

1/Billion

1900

1920

1940

1960

1980

2000

2020

Year

Brain Power Equivalent per \$1000 of Computer



1995 Trend

1985 Trend

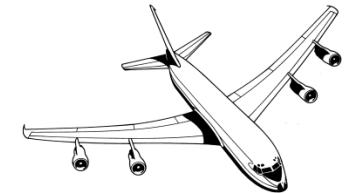
1975 Trend

1965 Trend

Gateway G6-200
PowerMac 8100/80
Gateway-486DX2/66
Mac II
Macintosh-128K
Commodore 64
IBM PC
Sun-2
DG Eclipse
CDC 7600
DEC PDP-10
IBM 7090
Whirlwind
IBM 704
UNIVAC I
ENIAC
Colossus
Burroughs Class 16
IBM Tabulator
Monroe Calculator
Zuse-1
ASCC (Mark 1)

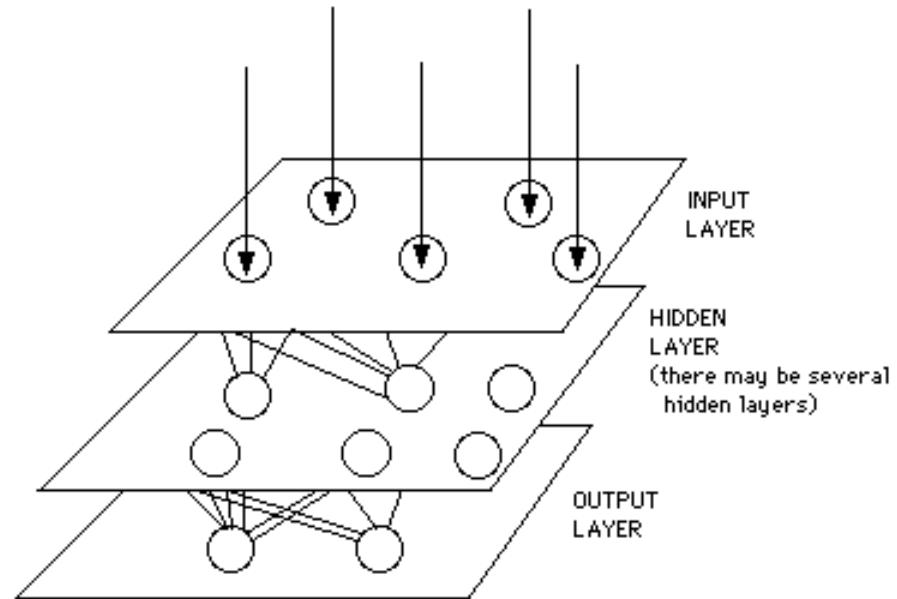
Evolution of the Main Ideas

- Wings or not?
- Games, mathematics, and other knowledge-poor tasks
- The silver bullet?
- Knowledge-based systems
- Hand-coded knowledge vs. machine learning
- Low-level (sensory and motor) processing and the resurgence of subsymbolic systems
- Robotics
- Natural language processing
- Programming languages
- Cognitive modeling



Symbolic vs. Subsymbolic AI

Subsymbolic AI: Model intelligence at a level similar to the neuron. Let such things as knowledge and planning emerge.

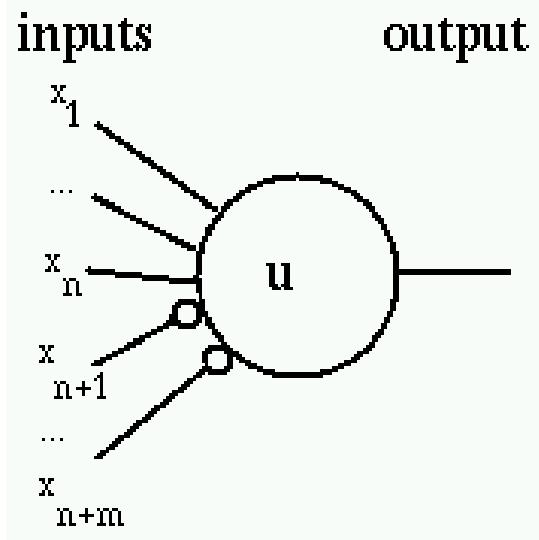


Symbolic AI: Model such things as knowledge and planning in data structures that make sense to the programmers that build them.

(blueberry (isa fruit)
(shape round)
(color purple)
(size .4 inch))

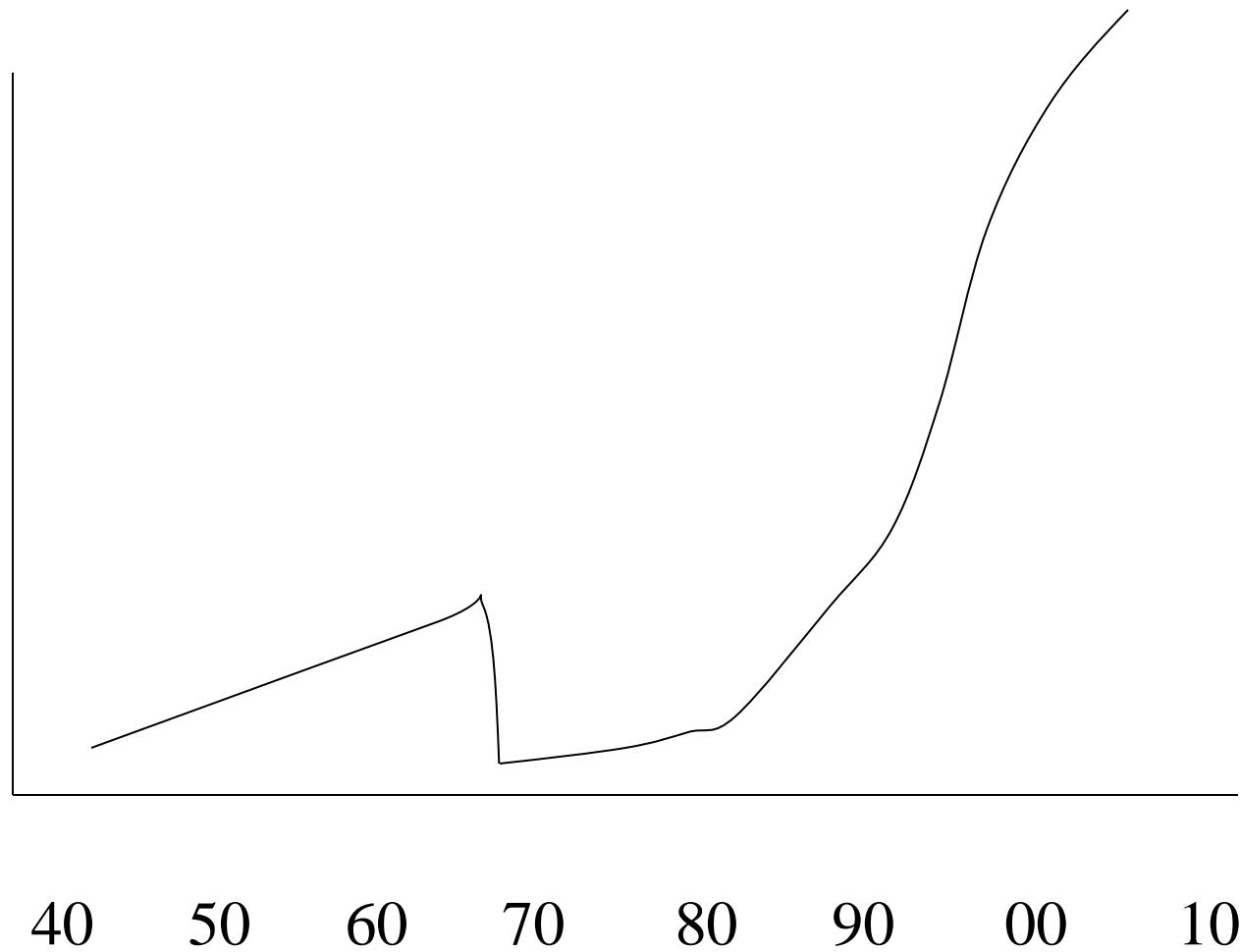
The Origins of Subsymbolic AI

1943 McCulloch and Pitts *A Logical Calculus of the Ideas Immanent in Nervous Activity*



“Because of the “all-or-none” character of nervous activity, neural events and the relations among them can be treated by means of propositional logic”

Interest in Subsymbolic AI



Low-level (Sensory and Motor) Processing and the Resurgence of Subsymbolic Systems

- Computer vision
- Motor control
- Subsymbolic systems perform cognitive tasks
 - Detect credit card fraud
- The backpropagation algorithm eliminated a formal weakness of earlier systems
- Neural networks learn.

The Origins of Symbolic AI

- Games
- Theorem proving

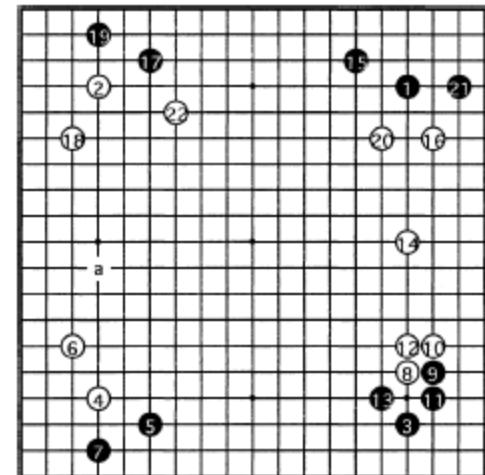
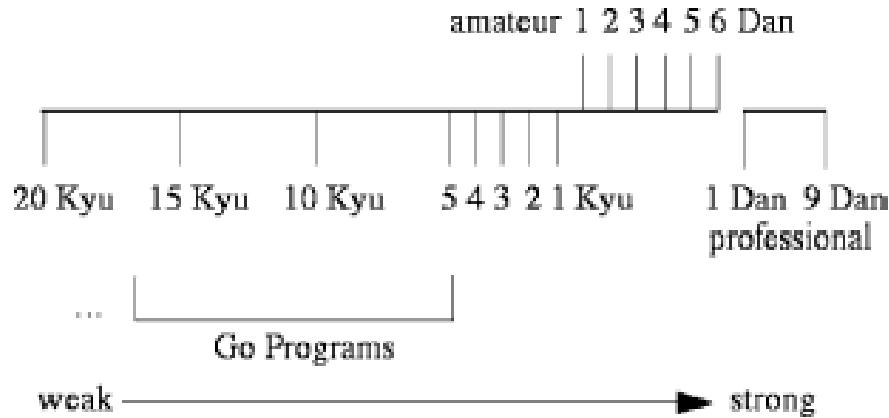
Games

- Chess
- Checkers:
 - 1952-1962 Art Samuel built the first checkers program
 - Chinook became the world checkers champion in 1994
- Othello:
 - Logistello beat the world champion in 1997

Games

- Chess
- Checkers: Chinook became the world checkers champion in 1994
- Othello: Logistello beat the world champion in 1997

• Go:



- Role Playing Games: now we need knowledge

Mathematics

- 1956 Logic Theorist (the first running AI program?)
 - 1961 SAINT solved calculus problems at the college freshman level
 - 1967 Macsyma
- Gradually theorem proving has become well enough understood that it is usually no longer considered AI
- 1996 J Moore and others verified the correctness of the AMD5k86 Floating-Point Division algorithm

The Silver Bullet?

Is there an “intelligence algorithm”?

1957 GPS (General Problem Solver)



Start

Goal

But What About Knowledge?

- Why do we need it?

Find me stuff about dogs who save people's lives.

Around midnight, two beagles spotted a fire in the house next door. Their barking alerted their owners, who called 911.

- How can we represent it and use it?
- How can we acquire it?

Representing Knowledge - Logic

1958 McCarthy's paper, "Programs with Common Sense"

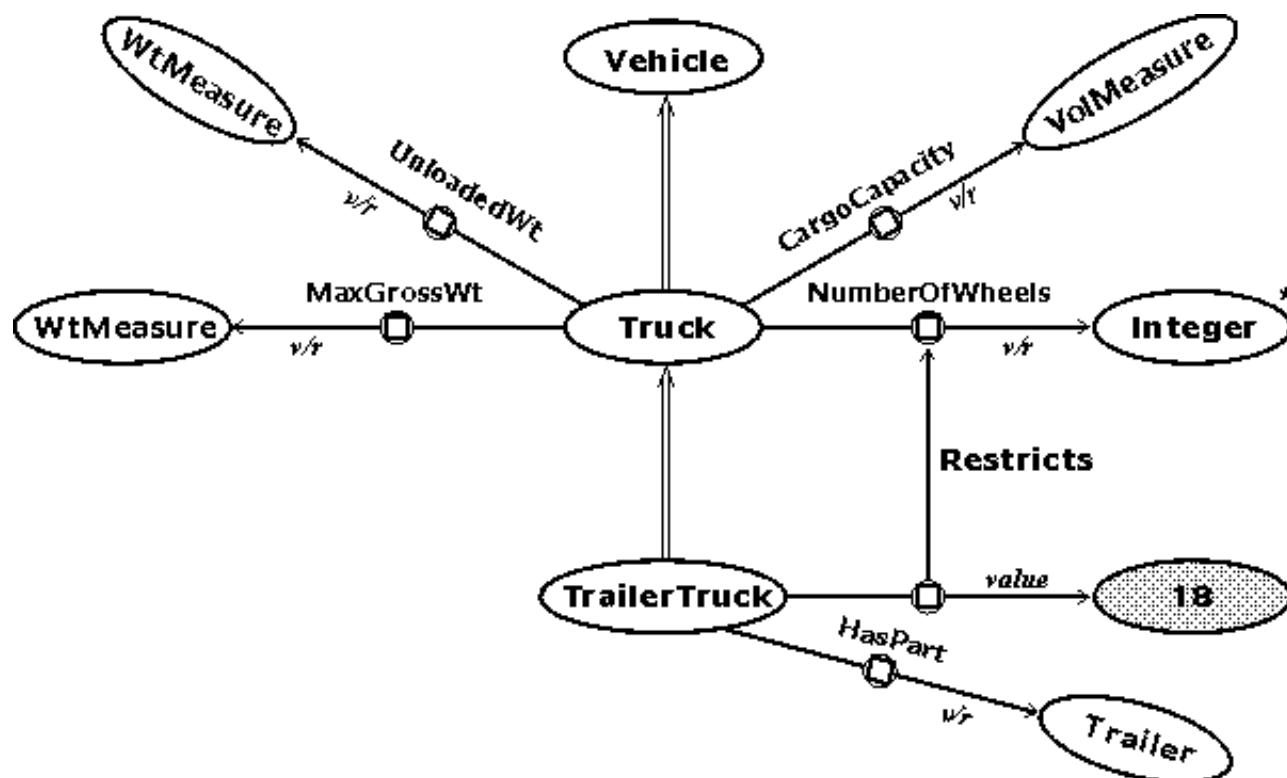
$at(I, car) \Rightarrow can(go(home, airport, driving))$

$at(I, desk) \Rightarrow can(go(desk, car, walking))$

1965 Resolution theorem proving invented

Representing Knowledge- Semantic Nets

1961



Representing Knowledge – Capturing Experience

Representing Experience with Scripts, Frames, and Cases

1977 Scripts

Joe went to a restaurant. Joe ordered a hamburger. When the hamburger came, it was burnt to a crisp. Joe stormed out without paying.

The restaurant script:

Did Joe eat anything?

Representing Knowledge - Rules

Expert knowledge in many domains can be captured in rules.

From XCON (1982):

*If: the most current active context is distributing
massbus devices, and*

*there is a single-port disk drive that has not been
assigned to a massbus, and*

*there are no unassigned dual-port disk drives, and
the number of devices that each massbus should support is known, and
there is a massbus that has been assigned at least one disk drive that
should support additional disk drives, and*

*the type of cable needed to connect the disk drive to the previous
device on the massbus is known*

Then: assign the disk drive to the massbus.

Representing Knowledge – Probabilistically

- 1975 Mycin attaches probability-like numbers to rules
- If: (1) the stain of the organism is gram-positive, and
(2) the morphology of the organism is coccus, and
(3) the growth conformation of the organism is clumps*
- Then: there is suggestive evidence (0.7) that the identity of the organism is stphylococcus.*
- 1970s Probabilistic models of speech recognition
- 1980s Statistical Machine Translation systems
- 1990s large scale neural nets

The Rise of Expert Systems

- 1967 Dendral – a rule-based system that inferred molecular structure from mass spectral and NMR data
- 1975 Mycin – a rule-based system to recommend antibiotic therapy
- 1975 Meta-Dendral learned new rules of mass spectrometry, the first discoveries by a computer to appear in a refereed scientific journal
- 1979 EMycin – the first expert system shell
- 1980's The Age of Expert Systems

Expert Systems – The Heyday

- 1979 Inference
 - 1980 IntelliCorp
 - 1981 Teknowledge
 - 1983 Carnegie Group
 - 1980 XCON (R1) – first real commercial expert system at DEC, configures VAX systems
 - 1981 Japanese Fifth Generation project launched as the Expert Systems age blossoms in the US.
 - 1984 Gold Hill Common Lisp
 - 1986 neural net start up companies appear
 - 1988 AI revenues peak at \$1 billion. AI Winter begins.
- 
- Selling expert system shells

Expert Systems – Today: Medicine

One example domain, medicine, has expert systems whose tasks include:

- arrhythmia recognition from electrocardiograms
- coronary heart disease risk group detection
- monitoring the prescription of restricted use antibiotics
- early melanoma diagnosis
- gene expression data analysis of human lymphoma
- breast cancer diagnosis

Expert Systems – Today: Build Your Own

eXpertise2Go: <http://www.expertise2go.com/>

AcquiredIntelligence: <http://www.aiinc.ca/demos/>

(whales, graduate school)

Taking the SAT Exam

QUESTION: Sodium azide is used in air bags to rapidly produce gas to inflate the bag. The products of the decomposition reaction are:(a) Na and water.(b) Ammonia and sodium metal.(c) N₂ and O₂(d) Sodium and nitrogen gas.(e) Sodium oxide and nitrogen gas.

(d) Sodium and nitrogen gas.

SYSTEM-GENERATED ANSWER:

What are the products of the given decomposition reaction?

• By definition, oxidation-reduction reactions occur when electrons are transferred from the atom that is oxidized to the atom that is reduced. We need to look for changes in the oxidation states of the elements in the reaction.

• In the reactants, the oxidation state(s) of the element Na is/are (1). In the product, the oxidation state(s) is/are (0).

Therefore, the reaction causes a change in oxidation state.

• Therefore, this is an oxidation reduction reaction.

• By definition, a Binary Ionic-Compound Decomposition Reaction occurs when a binary ionic compound is heated.

• Therefore, this reaction is a Binary-Ionic Compound Decomposition reaction.

• In general, a Binary Ionic-Compound Decomposition Reaction converts a binary ionic-compound into basic elements.

• In this reaction, NaN_3 reacts to produce Na and N_2 .

The products of the decomposition reaction are:

(d) Sodium and nitrogen gas.

What About Things that People Do *Easily*?

- Common sense

- Moving Around

- Language

What About Things that People Do *Easily*?

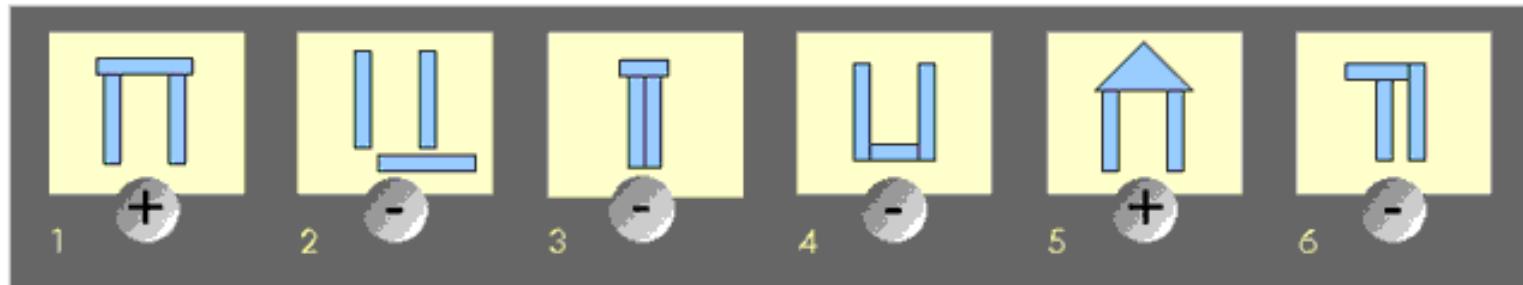
- Common sense
- CYC
- UT (<http://www.cs.utexas.edu/users/mfkb/RKF/tree/>)
- WordNet (<http://www.cogsci.princeton.edu/~wn/>)
- Moving around
- Language

Hand-Coded Knowledge vs. Machine Learning

- How much work would it be to enter knowledge by hand?
- Do we even know what to enter?

1952-62 Samuel's checkers player learned its evaluation function

1975 Winston's system learned structural descriptions from examples and near misses



1984 Probably Approximately Correct learning offers a theoretical foundation

mid 80's The rise of neural networks

Robotics - Tortoise

1950 W. Grey Walter's light seeking tortoises. In this picture, there are two, each with a light source and a light sensor. Thus they appear to “dance” around each other.



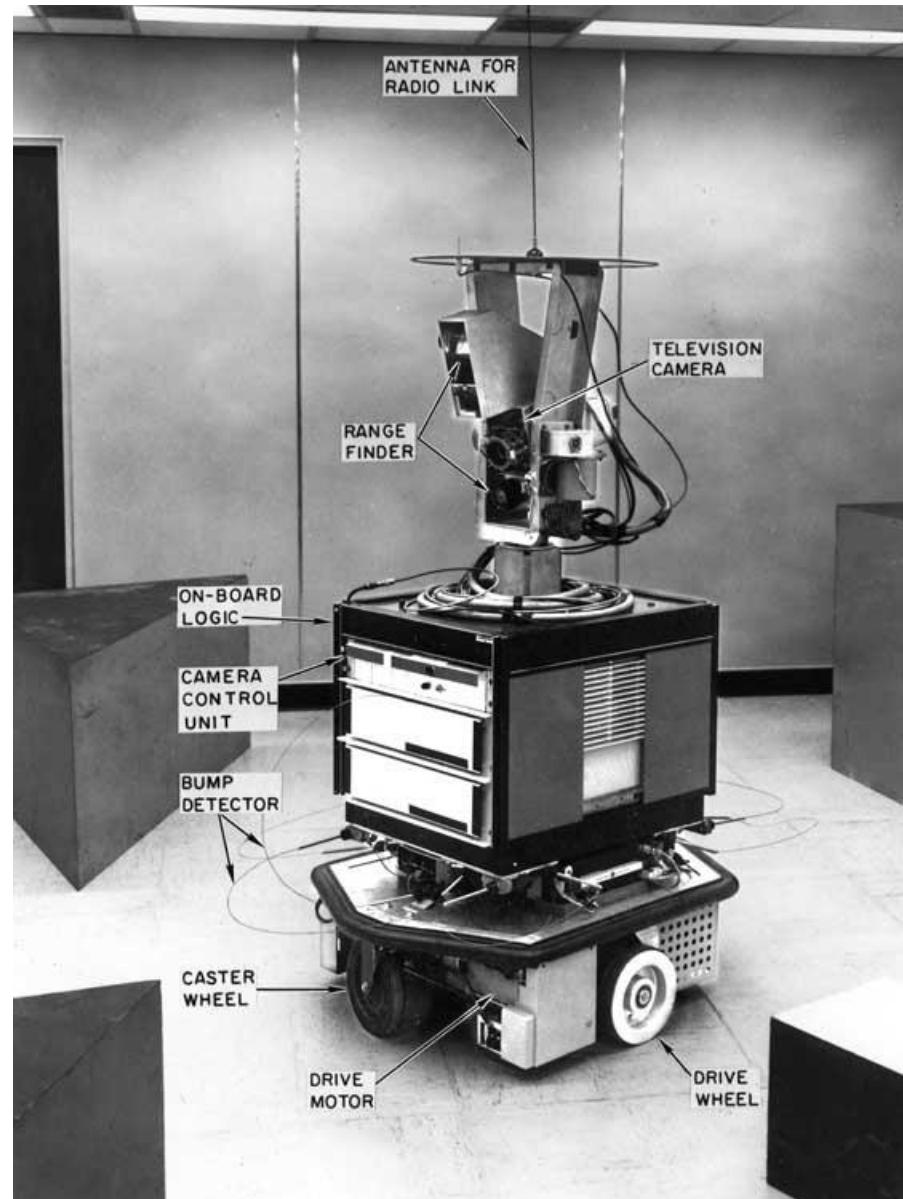
Robotics – Hopkins Beast

1964 Two versions of the Hopkins beast, which used sonar to guide it in the halls. Its goal was to find power outlets.



Robotics - Shakey

1970 Shakey (SRI) was driven by a remote-controlled computer, which formulated plans for moving and acting. It took about half an hour to move Shakey one meter.



Robotics – Stanford Cart

1971-9 Stanford cart.

Remote controlled by person or computer.

1971 follow the white line

1975 drive in a straight line by tracking skyline

1979 get through obstacle courses. Cross 30 meters in five hours, getting lost one time out of four

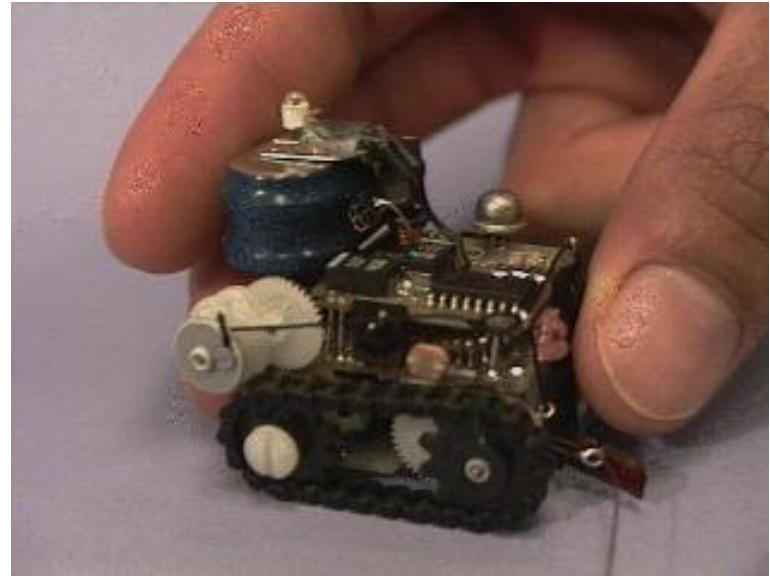


Planning vs. Reacting

In the early days: substantial focus on planning (e.g., GPS)

1979 – in “Fast, Cheap and Out of Control”, Rodney Brooks argued for a very different approach. (No, I’m not talking about the 1997 movie.)

The Ant, has 17 sensors.
They are designed to work
in colonies.



<http://www.ai.mit.edu/people/brooks/papers/fast-cheap.pdf>

<http://www.ai.mit.edu/projects/ants/>

Robotics - Dante

1994 Dante II (CMU) explored the Mt. Spurr (Aleutian Range, Alaska) volcano. High-temperature, fumarole gas samples are prized by volcanic science, yet their sampling poses significant challenge. In 1993, eight volcanologists were killed in two separate events while sampling and monitoring volcanoes.



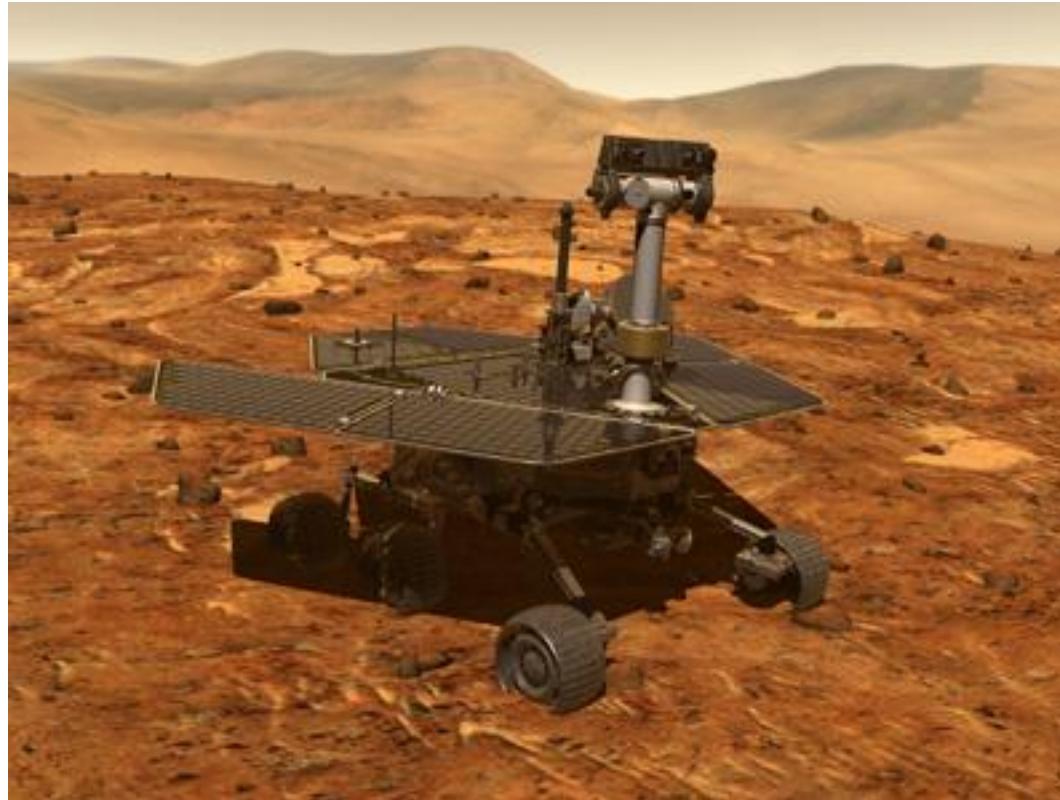
Using its tether cable anchored at the crater rim, Dante II is able to descend down sheer crater walls in a rappelling-like manner to gather and analyze high temperature gasses from the crater floor.

Robotics - Sojourner



Oct. 30, 1999 Sojourner on Mars. Powered by a 1.9 square foot solar array, Sojourner can negotiate obstacles tilted at a 45 degree angle. It travels at less than half an inch per second.

Robotics – Mars Rover



Tutorial on Rover:

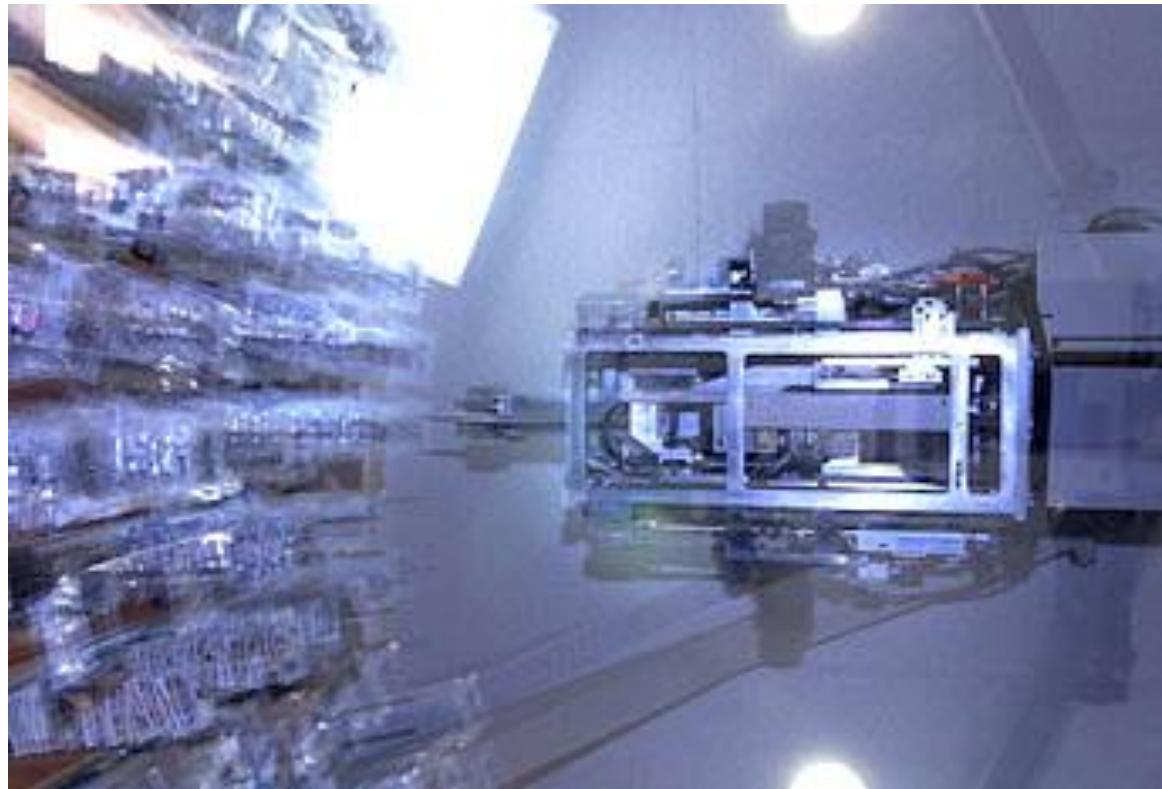
<http://marsrovers.jpl.nasa.gov/gallery/video/animation.html>

Sandstorm



March 13, 2004 - A DARPA Grand Challenge: an unmanned offroad race, 142 miles from Barstow to Las Vegas.

Moving Around and Picking Things Up



Phil, the drug robot, introduced in 2003

Robotics - Aibo

1999 Sony's Aibo pet dog



What Can You Do with an Aibo?

1997 – First official Rob-Cup soccer match

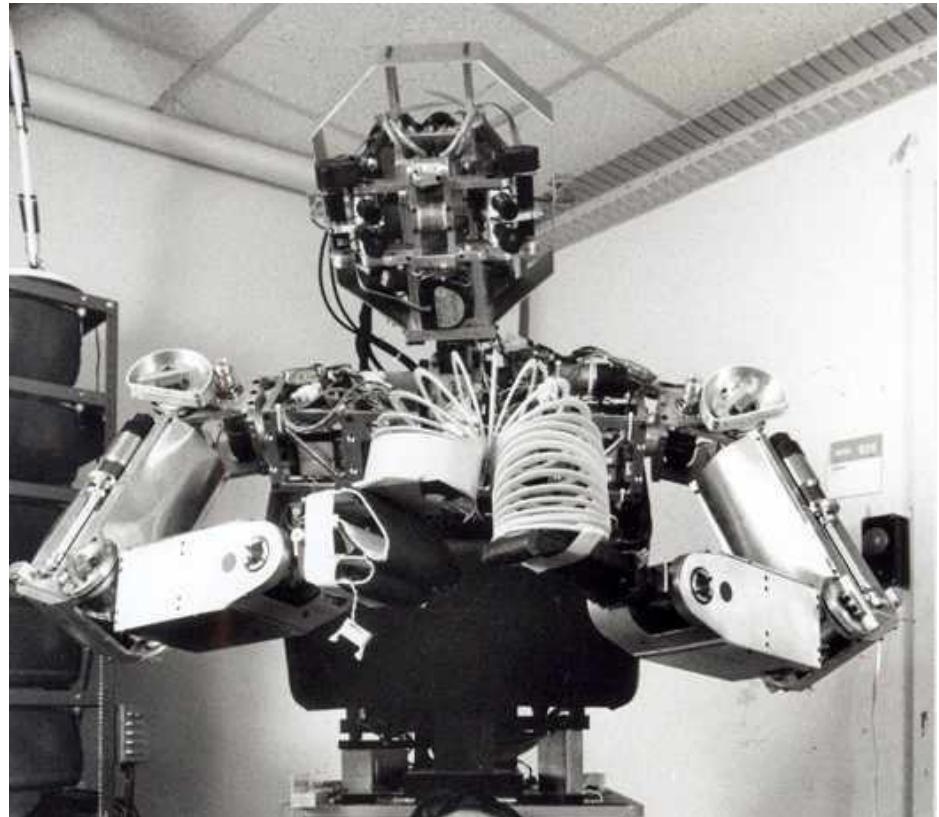


Picture from 2003
competition

Robotics - Cog

1998 – now Cog

*Humanoid
intelligence requires
humanoid
interactions with the
world.*



At the Other End of the Spectrum - Roomba

2001 A robot
vacuum cleaner



Natural Language Processing

1964 STUDENT solves algebra word problems

The distance from New York to Los Angeles is 3000 miles. If the average speed of a jet plane is 600 miles per hour, find the time it takes to travel from New York to Los Angeles by jet.

1965 ELIZA models a Rogerian therapist

young woman: Men are all alike.

eliza: In what way?

young woman: They're always bugging us about something specific or other.

eliza: Can you think of a specific example?

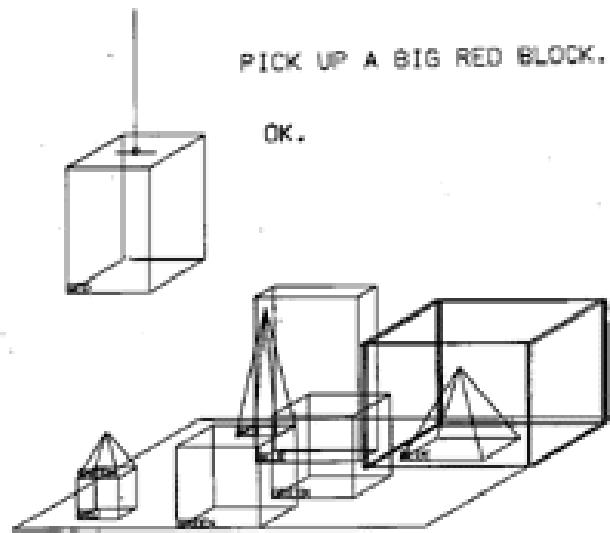
young woman: Well, my boyfriend made me come here.

eliza: Your boyfriend made you come here?

NLP, continued

1966 Alpac report kills work on MT

1971 SHRDLU



NLP, continued

1973 Schank – a richer limited domain: children’s stories

Suzie was invited to Mary’s birthday party. She knew she wanted a new doll so she got it for her.

1977 Schank – scripts add a knowledge layer – restaurant stories

1970’s and 80’s sophisticated grammars and parsers

But suppose we want generality? One approach is “shallow” systems that punt the complexities of meaning.

NLP Today

- Grammar and spelling checkers
 - Spelling: <http://www.spellcheck.net/>
- Chatbots
 - See the list at:
<http://www.aaai.org/AITopics/html/natlang.html#chat/>
- Speech systems
 - Synthesis: The IBM system:
 - <http://www.research.ibm.com/tts/coredemo.html>

Machine Translation: An Early NL Application

1949 Warren Weaver's memo suggesting MT

1966 Alpac report kills government funding

Early 70s SYSTRAN develops direct Russian/English system

Early 80s knowledge based MT systems

Late 80s statistical MT systems

MT Today

Austin Police are trying to find the person responsible for robbing a bank in Downtown Austin.

El policía de Austin está intentando encontrar a la persona responsable de robar un banco en Austin céntrica.

The police of Austin is trying to find the responsible person to rob a bank in centric Austin.

MT Today

A Florida teen charged with hiring an undercover policeman to shoot and kill his mother instructed the purported hitman not to damage the family television during the attack, police said on Thursday.

Un adolescente de la Florida cargado con emplear a un policía de la cubierta interior para tirar y para matar a su madre mandó a hitman pretendida para no dañar la televisión de la familia durante el ataque, limpia dicho el jueves.

An adolescent of Florida loaded with using a police of the inner cover to throw and to kill his mother commanded to hitman tried not to damage the television of the family during the attack, clean said Thursday.

MT Today

I have a dream, that my four little children will one day live in a nation where they will not be judged by the color of their skin but by the content of their character. I have a dream today – *Martin Luther King*

I am a sleepy, that my four small children a day of alive in a nation in where they will not be judged by the color of its skin but by the content of its character. I am a sleepy today. (Spanish)

Why Is It So Hard?

Sue caught the bass with her new rod.

Why Is It So Hard?

Sue caught (the bass) (with her new rod).

Why Is It So Hard?

Sue caught the bass with **the dark stripes.**

Why Is It So Hard?

Sue caught (the bass with **the dark stripes**).

Why Is It So Hard?

Sue **played** the bass with **her new bow**.

Why Is It So Hard?

Sue **played** the bass with **her new bow**.

Sue **played** the bass with **her new beau**.

Why Is It So Hard?

Olive oil



Why Is It So Hard?

Olive oil



Why Is It So Hard?

Peanut oil



Why Is It So Hard?

Coconut oil



Why Is It So Hard?

Baby oil



Why Is It So Hard?

Cooking oil



MT Today

Is MT an “AI complete” problem?

- *John saw a bicycle in the store window. He wanted it.*
- *John saw a bicycle in the store window. He pressed his nose up against it.*

- *John saw the Statue of Liberty flying over New York.*
- *John saw a plane flying over New York.*

Text Retrieval and Extraction

- Try Ask Jeeves: <http://www.askjeeves.com>
- To do better requires:
 - Linguistic knowledge
 - World knowledge
- Newsblaster: <http://www1.cs.columbia.edu/nlp/newsblaster/>

Programming Languages

- 1958 Lisp – a functional programming language with a simple syntax.
(successor SitA ActionP)
- 1972 PROLOG - a logic programming language whose primary control structure is depth-first search
$$\text{ancestor}(A,B) :- \text{parent}(A,B)$$
$$\text{ancestor}(A,B) :- \text{parent}(A,P), \text{ancestor}(P,B)$$
- 1988 CLOS (Common Lisp Object Standard) published.
Draws on ideas from Smalltalk and semantic nets

Cognitive Modeling

Symbolic Modeling

1957 GPS

1983 SOAR

Neuron-Level Modeling

McCulloch Pitts neurons: all or none response

More sophisticated neurons and connections

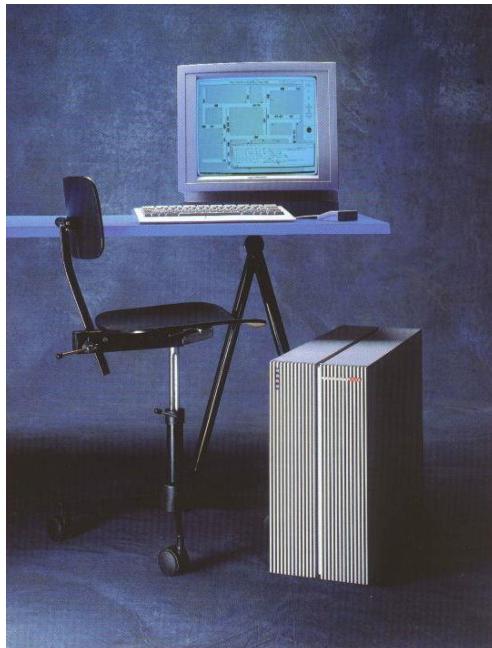
More powerful learning algorithm

Making Money – Software

- Expert systems to solve problems in particular domains
- Expert system shells to make it cheaper to build new systems in new domains
- Language applications
 - Text retrieval
 - Machine Translation
 - Text to speech and speech recognition
- Data mining

Making Money - Hardware

- 1980 Symbolics founded
- 1986 Thinking Machines introduces the Connection Machine
- 1993 Symbolics declared bankruptcy



Symbolics 3620 System c 1986:
Up to 4 Mwords (16 Mbytes)
optional physical memory, one
190 Mbyte fixed disk, integral
Ethernet interface, five backplane
expansion slots, options include an
additional 190 Mbyte disk or 1/4"
tape drive, floating point
accelerator, memory, RS232C
ports and printers.

Making Money - Robots

1962 Unimation, first industrial robot company, founded. Sold a die casting robot to GM.

1990 iRobot founded, a spinoff of MIT

2000 The UN estimated that there are 742,500 industrial robots in use worldwide. More than half of these were being used in Japan.

2001 iRobot markets Roomba for \$200.



The Differences Between Us and Them

Emotions

Understanding

Consciousness

Emotions

The robot Kismet shows emotions



sad



surprise

Understanding

登 长 城 纪 念



我登上了長城

Consciousness

Me



You



Today: The Difference Between Us and Them

The
CAPTCHA
Project



Today: Computer as Artist

Two paintings done by Harold Cohen's Aaron program:



Why AI?

"AI can have two purposes. One is to use the power of computers to augment human thinking, just as we use motors to augment human or horse power. Robotics and expert systems are major branches of that. The other is to use a computer's artificial intelligence to understand how humans think. In a humanoid way. If you test your programs not merely by what they can accomplish, but how they accomplish it, then you're really doing cognitive science; you're using AI to understand the human mind."

- Herb Simon

Introduction to Artificial Intelligence

CSE 401: Artificial Intelligence

Reference Book

- Artificial Intelligence: A Modern Approach
 - Stuart Russell and Peter Norving

Goals of this Course

- This class is a broad introduction to artificial intelligence (AI)
 - AI is a very broad field with many subareas
 - We will cover many of the primary concepts/ideas

Why AI can change our life.....

- As we begin the new millenium
 - science and technology are changing rapidly
 - “old” sciences such as physics are relatively well-understood
 - computers are ubiquitous
- Grand Challenges in Science and Technology
 - understanding the brain
 - reasoning, cognition, creativity
 - creating intelligent machines
 - is this possible?
 - what are the technical and philosophical challenges?
 - arguably AI poses the most interesting challenges and questions in computer science today

This Lecture

- What is intelligence? What is artificial intelligence?
- A very brief history of AI
 - Modern successes: Stanley the driving robot
- An AI scorecard
 - How much progress has been made in different aspects of AI
- AI in practice
 - Successful applications
- The rational agent view of AI

What is Intelligence?

- Intelligence:
 - “the capacity to learn and solve problems” (Websters dictionary)
 - in particular,
 - *the ability to solve novel problems*
 - *the ability to act rationally*
 - *the ability to act like humans*
- Artificial Intelligence
 - build and understand intelligent entities or agents
 - 2 main approaches: “engineering” versus “cognitive modeling”

What's involved in Intelligence?

- Ability to interact with the real world
 - to perceive, understand, and act
 - e.g., speech recognition and understanding and synthesis
 - e.g., image understanding
 - e.g., ability to take actions, have an effect
- Reasoning and Planning
 - modeling the external world, given input
 - solving new problems, planning, and making decisions
 - ability to deal with unexpected problems, uncertainties
- Learning and Adaptation
 - we are continuously learning and adapting
 - our internal models are always being “updated”
 - e.g., a baby learning to categorize and recognize animals

Academic Disciplines relevant to AI

- Philosophy Logic, methods of reasoning, mind as physical system, foundations of learning, language, rationality.
 - Mathematics Formal representation and proof, algorithms, computation, (un)decidability, (in)tractability
 - Probability/Statistics modeling uncertainty, learning from data
 - Economics utility, decision theory, rational economic agents
 - Neuroscience neurons as information processing units.
 - Psychology/ Cognitive Science how do people behave, perceive, process cognitive information, represent knowledge.
 - Computer engineering building fast computers
 - Control theory design systems that maximize an objective function over time
 - Linguistics knowledge representation, grammars

History of AI

- 1943: early beginnings
 - McCulloch & Pitts: Boolean circuit model of brain
- 1950: Turing
 - Turing's "Computing Machinery and Intelligence"
- 1956: birth of AI
 - Dartmouth meeting: "Artificial Intelligence" name adopted
- 1950s: initial promise
 - Early AI programs, including
 - Samuel's checkers program
 - Newell & Simon's Logic Theorist
- 1955-65: "great enthusiasm"
 - Newell and Simon: GPS, general problem solver
 - Gelertner: Geometry Theorem Prover
 - McCarthy: invention of LISP

History of AI

- 1966–73: Reality dawns
 - Realization that many AI problems are intractable
 - Limitations of existing neural network methods identified
 - Neural network research almost disappears
- 1969–85: Adding domain knowledge
 - Development of knowledge-based systems
 - Success of rule-based expert systems,
 - E.g., DENDRAL, MYCIN
 - But were brittle and did not scale well in practice
- 1986-- Rise of machine learning
 - Neural networks return to popularity
 - Major advances in machine learning algorithms and applications
- 1990-- Role of uncertainty
 - Bayesian networks as a knowledge representation framework
- 1995-- AI as Science
 - Integration of learning, reasoning, knowledge representation
 - AI methods used in vision, language, data mining, etc

Success Stories

- Deep Blue defeated the reigning **world chess champion** Garry Kasparov in 1997
- AI program proved a mathematical conjecture (Robbins conjecture) unsolved for decades
- During the 1991 Gulf War, US forces deployed **an AI logistics planning and scheduling program** that involved up to 50,000 vehicles, cargo, and people
- NASA's on-board **autonomous planning program** controlled the scheduling of operations for a spacecraft
- Proverb solves crossword puzzles better than most humans
- Robot driving: DARPA grand challenge 2003-2007
- 2006: face recognition software available in consumer cameras

Example: DARPA Grand Challenge

- Grand Challenge
 - Cash prizes (\$1 to \$2 million) offered to first robots to complete a long course completely unassisted
 - Stimulates research in vision, robotics, planning, machine learning, reasoning, etc
- 2004 Grand Challenge:
 - 150 mile route in Nevada desert
 - Furthest any robot went was about 7 miles
 - ... but hardest terrain was at the beginning of the course
- 2005 Grand Challenge:
 - 132 mile race
 - Narrow tunnels, winding mountain passes, etc
 - Stanford 1st, CMU 2nd, both finished in about 6 hours
- 2007 Urban Grand Challenge
 - This November in Victorville, California

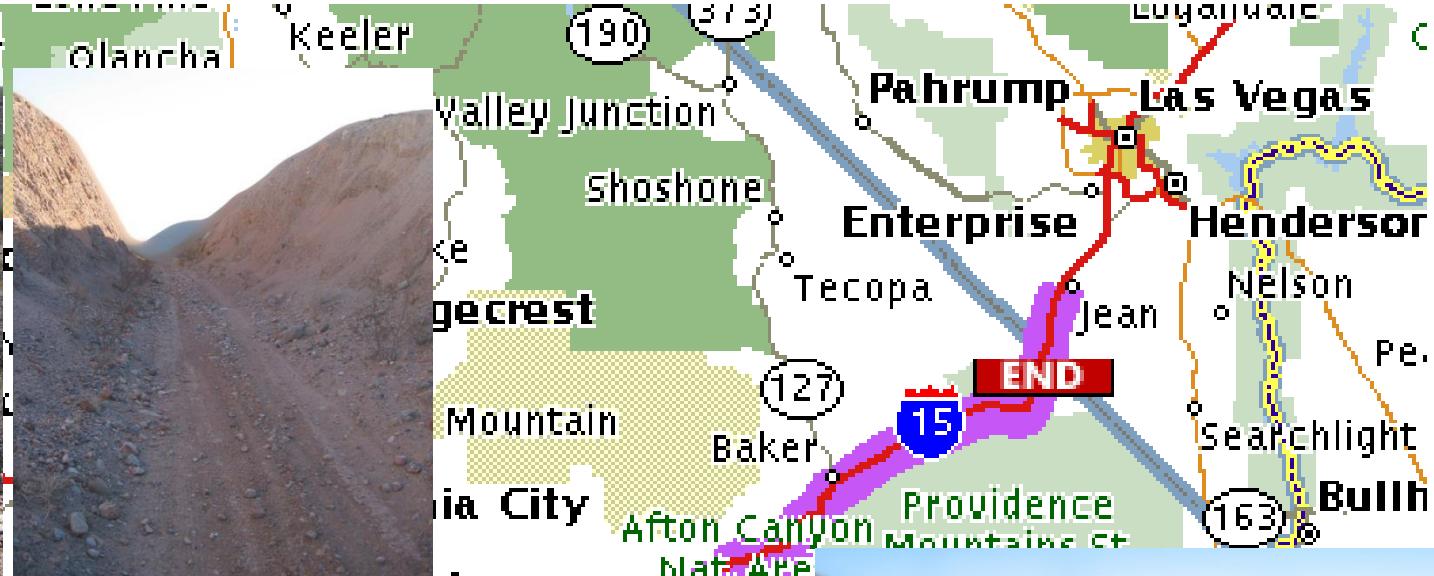
Stanley Robot

Stanford Racing Team www.stanfordracing.org



Next few slides courtesy of Prof.
Sebastian Thrun, Stanford University

2004: Barstow, CA, to Primm, NV



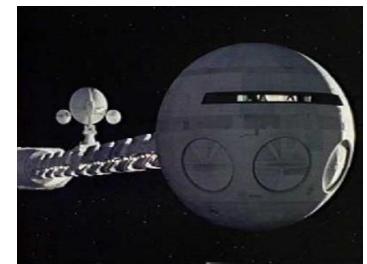
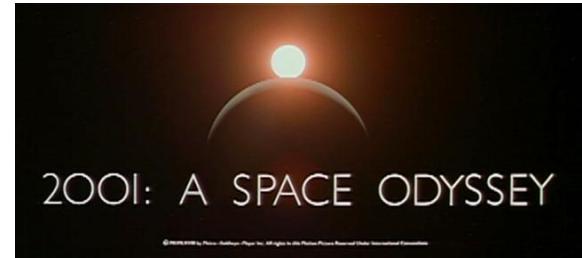
- 150 mile off-road robot race across the Mojave desert
- Natural and manmade hazards
- No driver, no remote control
- No dynamic passing
- Fastest vehicle wins the race (and 2 million dollar prize)

The Grand Challenge Race



HAL: from the movie 2001

- *2001: A Space Odyssey*
 - classic science fiction movie from 1969
- HAL
 - part of the story centers around an intelligent computer called HAL
 - HAL is the “brains” of an intelligent spaceship
 - in the movie, HAL can
 - speak easily with the crew
 - see and understand the emotions of the crew
 - navigate the ship automatically
 - diagnose on-board problems
 - make life-and-death decisions
 - display emotions
- In 1969 this was science fiction: is it still science fiction?



Hal and AI

- *HAL's Legacy: 2001's Computer as Dream and Reality*
 - MIT Press, 1997, David Stork (ed.)
 - discusses
 - HAL as an intelligent computer
 - are the predictions for HAL realizable with AI today?
- Materials online at
 - <http://mitpress.mit.edu/e-books/Hal/contents.html>
- The website contains
 - full text and abstracts of chapters from the book
 - links to related material and AI information
 - sound and images from the film

Consider what might be involved in building a computer like Hal....

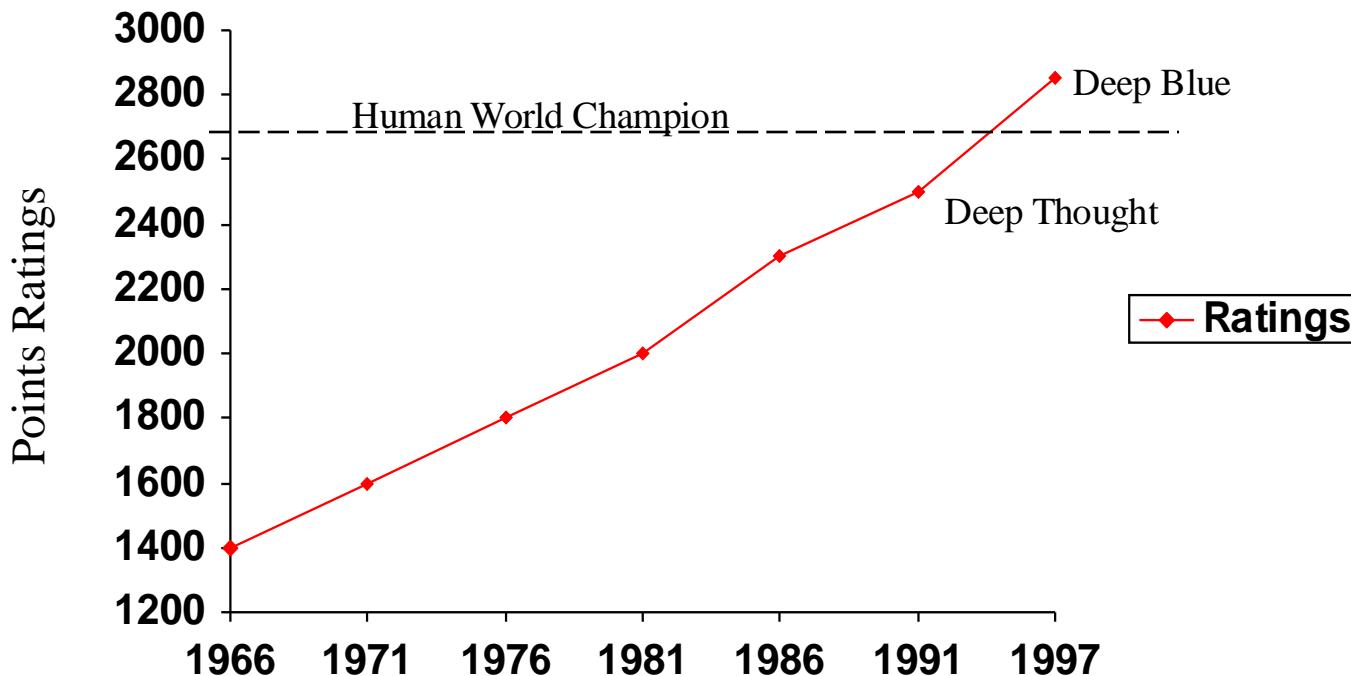
- What are the components that might be useful?
 - Fast hardware?
 - Chess-playing at grandmaster level?
 - Speech interaction?
 - speech synthesis
 - speech recognition
 - speech understanding
 - Image recognition and understanding ?
 - Learning?
 - Planning and decision-making?

Can we build hardware as complex as the brain?

- How complicated is our brain?
 - a neuron, or nerve cell, is the basic information processing unit
 - estimated to be on the order of 10^{12} neurons in a human brain
 - many more synapses (10^{14}) connecting these neurons
 - cycle time: 10^{-3} seconds (1 millisecond)
- How complex can we make computers?
 - 10^8 or more transistors per CPU
 - supercomputer: hundreds of CPUs, 10^{12} bits of RAM
 - cycle times: order of 10^{-9} seconds
- Conclusion
 - YES: in the near future we can have computers with as many basic processing elements as our brain, but with
 - far fewer interconnections (wires or synapses) than the brain
 - much faster updates than the brain
 - but building hardware is very different from making a computer behave like a brain!

Can Computers beat Humans at Chess?

- Chess Playing is a classic AI problem
 - well-defined problem
 - very complex: difficult for humans to play well



- Conclusion:
 - YES: today's computers can beat even the best human

Can Computers Talk?

- This is known as “speech synthesis”
 - translate text to phonetic form
 - e.g., “fictitious” -> fik-tish-es
 - use pronunciation rules to map phonemes to actual sound
 - e.g., “tish” -> sequence of basic audio sounds
- Difficulties
 - sounds made by this “lookup” approach sound unnatural
 - sounds are not independent
 - e.g., “act” and “action”
 - modern systems (e.g., at AT&T) can handle this pretty well
 - a harder problem is emphasis, emotion, etc
 - humans understand what they are saying
 - machines don’t: so they sound unnatural
- Conclusion:
 - NO, for complete sentences
 - YES, for individual words

Can Computers Recognize Speech?

- Speech Recognition:
 - mapping sounds from a microphone into a list of words
 - classic problem in AI, very difficult
 - “Lets talk about how to wreck a nice beach”
 - (I really said “ _____ ”)
- Recognizing single words from a small vocabulary
 - systems can do this with high accuracy (order of 99%)
 - e.g., directory inquiries
 - limited vocabulary (area codes, city names)
 - computer tries to recognize you first, if unsuccessful hands you over to a human operator
 - saves millions of dollars a year for the phone companies

Recognizing human speech (ctd.)

- Recognizing normal speech is much more difficult
 - speech is continuous: where are the boundaries between words?
 - e.g., "John's car has a flat tire"
 - large vocabularies
 - can be many thousands of possible words
 - we can use **context** to help figure out what someone said
 - e.g., hypothesize and test
 - try telling a waiter in a restaurant:
"I would like some dream and sugar in my coffee"
 - background noise, other speakers, accents, colds, etc
 - on normal speech, modern systems are only about 60-70% accurate
- Conclusion:
 - NO, normal speech is too complex to accurately recognize
 - YES, for restricted problems (small vocabulary, single speaker)

Can Computers Understand speech?

- Understanding is different to recognition:
 - “Time flies like an arrow”
 - assume the computer can recognize all the words
 - how many different interpretations are there?

Can Computers Understand speech?

- Understanding is different to recognition:
 - “Time flies like an arrow”
 - assume the computer can recognize all the words
 - how many different interpretations are there?
 - 1. time passes quickly like an arrow?
 - 2. command: time the flies the way an arrow times the flies
 - 3. command: only time those flies which are like an arrow
 - 4. “time-flies” are fond of arrows

Can Computers Understand speech?

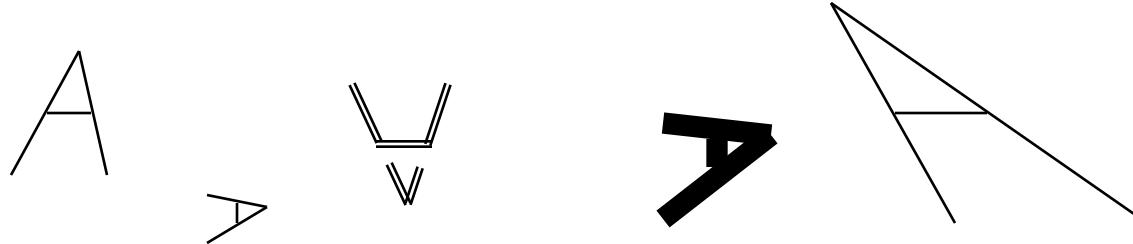
- Understanding is different to recognition:
 - “Time flies like an arrow”
 - assume the computer can recognize all the words
 - how many different interpretations are there?
 - 1. time passes quickly like an arrow?
 - 2. command: time the flies the way an arrow times the flies
 - 3. command: only time those flies which are like an arrow
 - 4. “time-flies” are fond of arrows
 - only 1. makes any sense,
 - but how could a computer figure this out?
 - clearly humans use a lot of implicit commonsense knowledge in communication
 - Conclusion: NO, much of what we say is beyond the capabilities of a computer to understand at present

Can Computers Learn and Adapt ?

- Learning and Adaptation
 - consider a computer learning to drive on the freeway
 - we could teach it lots of rules about what to do
 - or we could let it drive and steer it back on course when it heads for the embankment
 - systems like this are under development (e.g., Daimler Benz)
 - e.g., RALPH at CMU
 - in mid 90's it drove 98% of the way from Pittsburgh to San Diego without any human assistance
 - **machine learning** allows computers to learn to do things without explicit programming
 - many successful applications:
 - requires some "set-up": does not mean your PC can learn to forecast the stock market or become a brain surgeon
- Conclusion: YES, computers can learn and adapt, when presented with information in the appropriate way

Can Computers “see”?

- Recognition v. Understanding (like Speech)
 - Recognition and Understanding of Objects in a scene
 - look around this room
 - you can effortlessly recognize objects
 - human brain can map 2d visual image to 3d “map”
- Why is visual recognition a hard problem?



- Conclusion:
 - mostly NO: computers can only “see” certain types of objects under limited circumstances
 - YES for certain constrained problems (e.g., face recognition)

Can computers plan and make optimal decisions?

- Intelligence
 - involves solving problems and making decisions and plans
 - e.g., you want to take a holiday in Brazil
 - you need to decide on dates, flights
 - you need to get to the airport, etc
 - involves a sequence of decisions, plans, and actions
- What makes planning hard?
 - the world is not predictable:
 - your flight is canceled or there's a backup on the 405
 - there are a potentially huge number of details
 - do you consider all flights? all dates?
 - no: commonsense constrains your solutions
 - AI systems are only successful in constrained planning problems
- Conclusion: NO, real-world planning and decision-making is still beyond the capabilities of modern computers
 - exception: very well-defined, constrained problems

Summary of State of AI Systems in Practice

- **Speech synthesis, recognition and understanding**
 - very useful for limited vocabulary applications
 - unconstrained speech understanding is still too hard
- **Computer vision**
 - works for constrained problems (hand-written zip-codes)
 - understanding real-world, natural scenes is still too hard
- **Learning**
 - adaptive systems are used in many applications: have their limits
- **Planning and Reasoning**
 - only works for constrained problems: e.g., chess
 - real-world is too complex for general systems
- **Overall:**
 - many components of intelligent systems are “doable”
 - there are many interesting research problems remaining

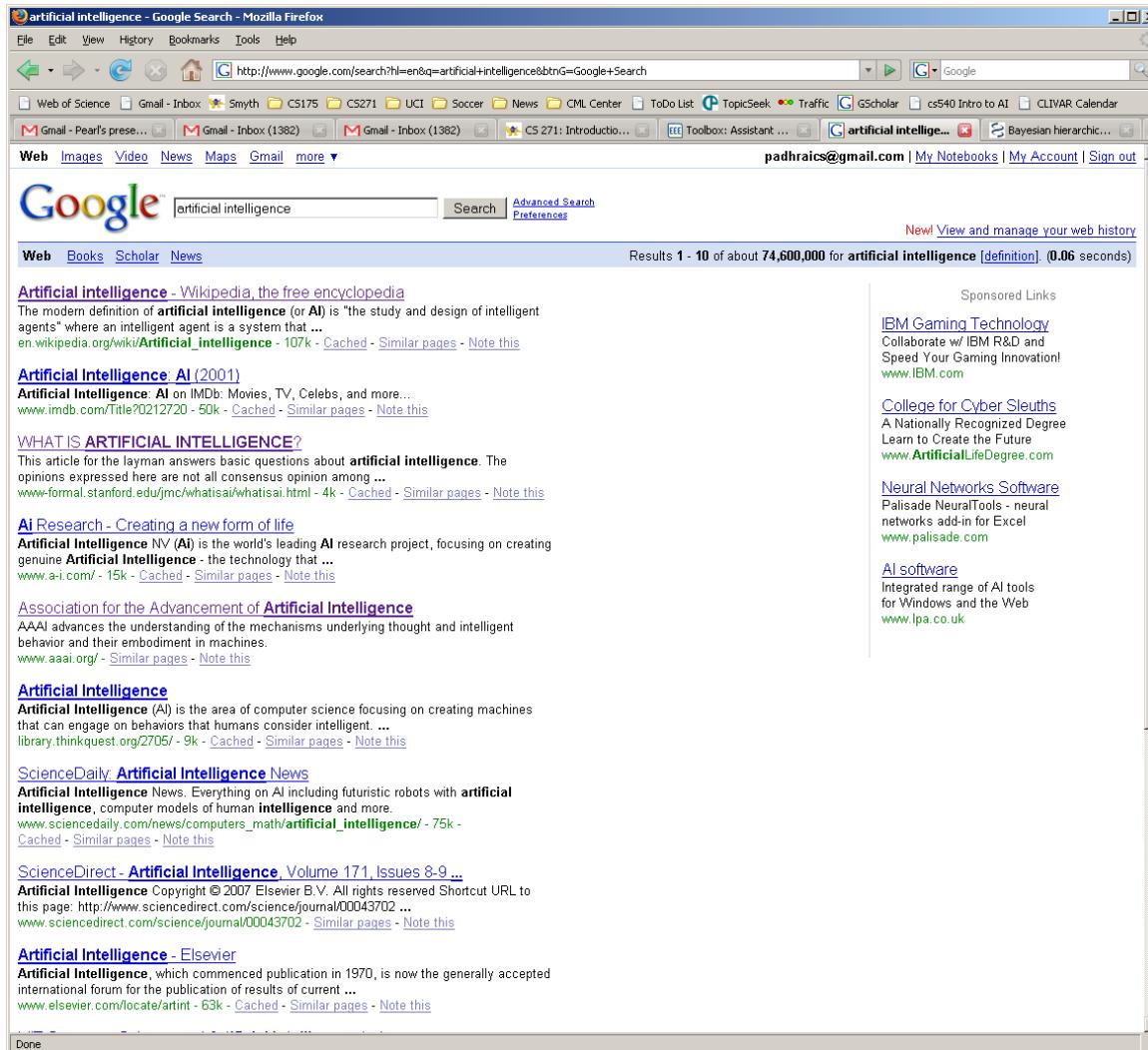
Intelligent Systems in Your Everyday Life

- Post Office
 - automatic address recognition and sorting of mail
- Banks
 - automatic check readers, signature verification systems
 - automated loan application classification
- Customer Service
 - automatic voice recognition
- The Web
 - Identifying your age, gender, location, from your Web surfing
 - Automated fraud detection
- Digital Cameras
 - Automated face detection and focusing
- Computer Games
 - Intelligent characters/agents

AI Applications: Machine Translation

- Language problems in international business
 - e.g., at a meeting of Japanese, Korean, Vietnamese and Swedish investors, no common language
 - or: you are shipping your software manuals to 127 countries
 - solution; hire translators to translate
 - would be much cheaper if a machine could do this
- How hard is automated translation
 - very difficult! e.g., English to Russian
 - “The spirit is willing but the flesh is weak” (English)
 - “the vodka is good but the meat is rotten” (Russian)
 - not only must the words be translated, but their meaning also!
 - is this problem “AI-complete”?
- Nonetheless....
 - commercial systems can do a lot of the work very well (e.g., restricted vocabularies in software documentation)
 - algorithms which combine dictionaries, grammar models, etc.
 - Recent progress using “black-box” machine learning techniques

AI and Web Search



A screenshot of a Mozilla Firefox browser window. The title bar says "artificial intelligence - Google Search - Mozilla Firefox". The address bar shows "http://www.google.com/search?hl=en&q=artificial+intelligence&btnG=Google+Search". The search query "artificial intelligence" is entered in the search bar. The results page displays 10 results out of approximately 74,600,000. The results include links to Wikipedia, IMDb, Stanford University, ThinkQuest, ScienceDaily, and Elsevier. On the right side of the results page, there are "Sponsored Links" for "IBM Gaming Technology", "College for Cyber Sleuths", "Neural Networks Software", and "AI software". The browser's toolbar and menu bar are visible at the top, and the user's email inbox is visible in the sidebar.

Google search results for "artificial intelligence":

- Artificial intelligence - Wikipedia, the free encyclopedia**
The modern definition of **artificial intelligence** (or AI) is "the study and design of intelligent agents" where an intelligent agent is a system that ...
en.wikipedia.org/wiki/Artificial_intelligence - 107k - [Cached](#) - [Similar pages](#) - [Note this](#)
- Artificial Intelligence_AI (2001)**
Artificial Intelligence: AI on IMDb: Movies, TV, Celebs, and more...
www.imdb.com>Title?0212720 - 50k - [Cached](#) - [Similar pages](#) - [Note this](#)
- WHAT IS ARTIFICIAL INTELLIGENCE?**
This article for the layman answers basic questions about **artificial intelligence**. The opinions expressed here are not all consensus opinion among ...
www.formal.stanford.edu/jmc/whatisai/whatisai.html - 4k - [Cached](#) - [Similar pages](#) - [Note this](#)
- Ai Research - Creating a new form of life**
Artificial Intelligence NV (Ai) is the world's leading AI research project, focusing on creating genuine **Artificial Intelligence** - the technology that ...
www.a-i.com/ - 15k - [Cached](#) - [Similar pages](#) - [Note this](#)
- Association for the Advancement of Artificial Intelligence**
AAAI advances the understanding of the mechanisms underlying thought and intelligent behavior and their embodiment in machines.
www.aaai.org/ - [Similar pages](#) - [Note this](#)
- Artificial Intelligence**
Artificial Intelligence (AI) is the area of computer science focusing on creating machines that can engage in behaviors that humans consider intelligent. ...
library.thinkquest.org/2705/ - 9k - [Cached](#) - [Similar pages](#) - [Note this](#)
- ScienceDaily: Artificial Intelligence News**
Artificial Intelligence News. Everything on AI including futuristic robots with **artificial intelligence**, computer models of human **intelligence** and more.
www.sciencedaily.com/news/computers_math/artificial_intelligence/ - 75k - [Cached](#) - [Similar pages](#) - [Note this](#)
- ScienceDirect - Artificial Intelligence, Volume 171, Issues 8-9 ...**
Artificial Intelligence Copyright © 2007 Elsevier B.V. All rights reserved Shortcut URL to this page: <http://www.sciencedirect.com/science/journal/00043702> - [Similar pages](#) - [Note this](#)
- Artificial Intelligence - Elsevier**
Artificial Intelligence, which commenced publication in 1970, is now the generally accepted international forum for the publication of results of current ...
www.elsevier.com/locate/artint - 63k - [Cached](#) - [Similar pages](#) - [Note this](#)

What's involved in Intelligence? (again)

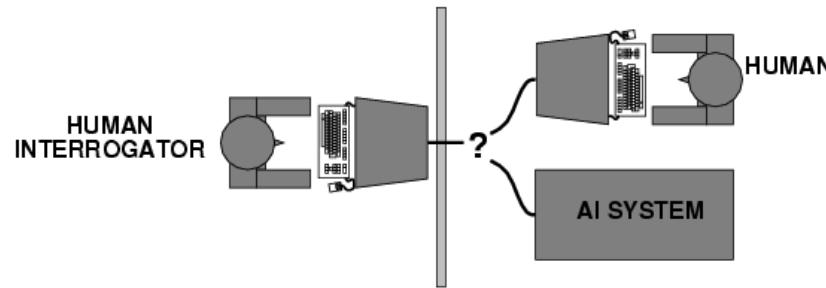
- Perceiving, recognizing, understanding the real world
 - Reasoning and planning about the external world
 - Learning and adaptation
-
- So what general principles should we use to achieve these goals?

Different Types of Artificial Intelligence

1. Modeling exactly how humans actually think
 2. Modeling exactly how humans actually act
 3. Modeling how ideal agents “should think”
 4. Modeling how ideal agents “should act”
- Modern AI focuses on the last definition
 - we will also focus on this “engineering” approach
 - success is judged by how well the agent performs

Acting humanly: Turing test

- Turing (1950) "Computing machinery and intelligence"
- "Can machines think?" → "Can machines behave intelligently?"
- Operational test for intelligent behavior: the Imitation Game



- Suggests major components required for AI:
 - knowledge representation
 - reasoning,
 - language/image understanding,
 - learning
- * Question: is it important that an intelligent system act like a human?

Thinking humanly

- Cognitive Science approach
 - Try to get “inside” our minds
 - E.g., conduct experiments with people to try to “reverse-engineer” how we reason, learning, remember, predict
- Problems
 - Humans don’t behave rationally
 - e.g., insurance
 - The reverse engineering is very hard to do
 - The brain’s hardware is very different to a computer program

Thinking rationally

- Represent facts about the world via logic
- Use logical inference as a basis for reasoning about these facts
- Can be a very useful approach to AI
 - E.g., theorem-provers
- Limitations
 - Does not account for an agent's uncertainty about the world
 - E.g., difficult to couple to vision or speech systems
 - Has no way to represent goals, costs, etc (important aspects of real-world environments)

Acting rationally

- Decision theory/Economics
 - Set of future states of the world
 - Set of possible actions an agent can take
 - Utility = gain to an agent for each action/state pair
 - An agent acts rationally if it selects the action that maximizes its “utility”
 - Or expected utility if there is uncertainty
- Emphasis is on autonomous agents that behave rationally (make the best predictions, take the best actions)
 - on average over time
 - within computational limitations (“bounded rationality”)

Summary of Today's Lecture

- Artificial Intelligence involves the study of:
 - automated recognition and understanding of signals
 - reasoning, planning, and decision-making
 - learning and adaptation
- AI has made substantial progress in
 - recognition and learning
 - some planning and reasoning problems
 - ...but many open research problems
- AI Applications
 - improvements in hardware and algorithms => AI applications in industry, finance, medicine, and science.
- Rational agent view of AI

<https://www.youtube.com/watch?v=IBYRixhYpck>

Chapter 2

Intelligent Agents

Agents

- An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators

Human agent:

eyes, ears, and other organs for sensors;
hands, legs, mouth, and other body parts for actuators

Robotic agent:

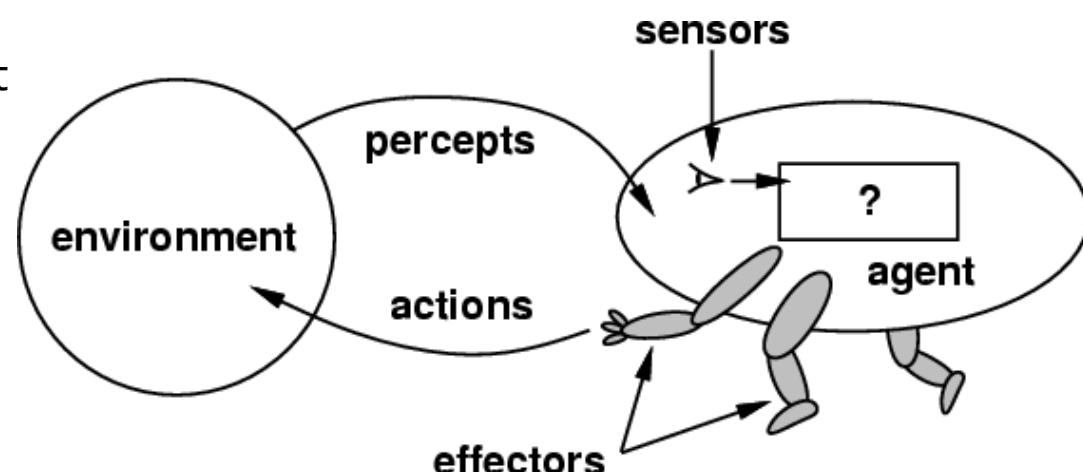
cameras and infrared range finders for sensors; various motors for actuators

How to design an intelligent agent?

- An **intelligent agent** perceives its environment via **sensors** and acts rationally upon that environment with its **effectors**.
- A discrete agent receives **percepts** one at a time, and maps this percept sequence to a sequence of discrete **actions**.

- Properties

- Autonomous
- Reactive to the environment
- Pro-active (goal-directed)
- Interacts with other agents via the environment

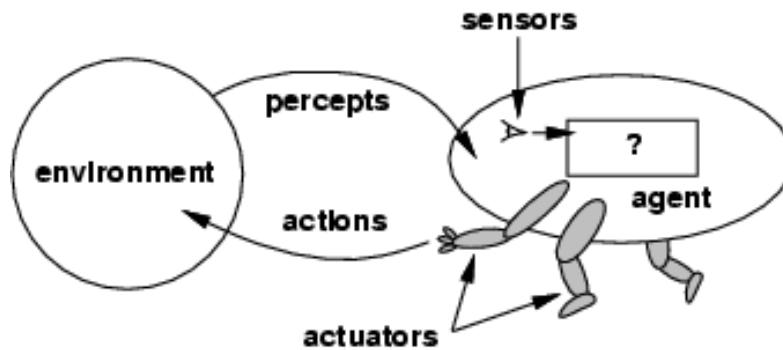


Sensors/percepts and effectors/actions?

- Humans

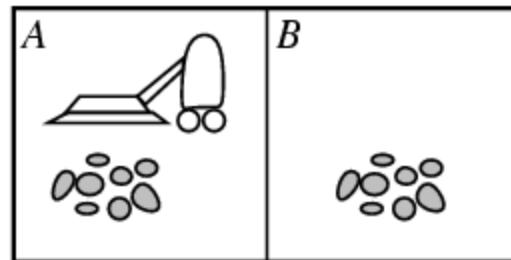
- Sensors: Eyes (vision), ears (hearing), skin (touch), tongue (gustation), nose (olfaction), neuromuscular system (proprioception)
- Percepts:
 - At the lowest level – electrical signals from these sensors
 - After preprocessing – objects in the visual field (location, textures, colors, ...), auditory streams (pitch, loudness, direction), ...
- Effectors: limbs, digits, eyes, tongue, ...
- Actions: lift a finger, turn left, walk, run, carry an object, ...

Agents and environments



- The agent function maps from percept histories to actions:
$$[f: \mathcal{P}^* \rightarrow \mathcal{A}]$$
- The agent program runs on the physical architecture to produce f
- agent = architecture + program

Vacuum-cleaner world



- Percepts: location and state of the environment, e.g., [A,Dirty], [A,Clean], [B,Dirty]
- Actions: *Left*, *Right*, *Suck*, *NoOp*

Rational agents

- **Performance measure:** An objective criterion for success of an agent's behavior, e.g.,
 - Robot driver?
 - Chess-playing program?
 - Spam email classifier?
- **Rational Agent:** selects actions that is *expected* to maximize its performance measure,
 - given percept sequence
 - given agent's built-in knowledge
 - sidepoint: how to maximize expected future performance, given only historical data

Rational agents

- Rational Agent → Always try to maximize performance.
- No Agent is **Omniscience**. Rationality is distinct from omniscience (all-knowing with infinite knowledge)
- Agents can perform actions in order to modify future percepts so as to obtain useful information (information gathering, exploration)
- An agent is **autonomous** if its behavior is determined by its own percepts & experience (with ability to learn and adapt) without depending solely on built-in knowledge
- To survive, agents must have:
 - Enough built-in knowledge to survive.
 - The ability to learn

Task Environment

- Before we design an intelligent agent, we must specify its “task environment”:

PEAS:

Performance measure

Environment

Actuators

Sensors

DARPA Robotics Challenge

- A prize competition funded by the US Defense Advanced Research Projects Agency, held from 2012 to 2015
- Aimed to develop semi-autonomous ground robots that could do complex tasks in dangerous, degraded, human-engineered environments

DARPA Robotics Challenge

The initial task requirements for robot entries are

- Drive a utility vehicle at the site
- Travel dismounted across rubble
- Remove debris blocking an entryway
- Open a door and enter a building
- Climb an industrial ladder and traverse an industrial walkway
- Use a tool to break through a concrete panel
- Locate and close a valve near a leaking pipe
- Connect a fire hose to a standpipe and turn on a valve

PEAS

- Example: Agent = robot driver in DARPA Challenge
 - Performance measure:
 - Time to complete course
 - Environment:
 - Roads, other traffic, obstacles
 - Actuators:
 - Steering wheel, accelerator, brake, signal, horn
 - Sensors:
 - Optical cameras, lasers, sonar, accelerometer, speedometer, GPS, odometer, engine sensors,

PEAS

- Example: Agent = Medical diagnosis system

Performance measure:

Healthy patient, minimize costs, lawsuits

Environment:

Patient, hospital, staff

Actuators:

Screen display (questions, tests, diagnoses, treatments, referrals)

Sensors:

Keyboard (entry of symptoms, findings, patient's answers)

Environment types

- **Fully observable** (vs. **partially observable**):
 - An agent's sensors give it access to the complete state of the environment at each point in time.
- **Deterministic** (vs. **stochastic**):
 - The next state of the environment is completely determined by the current state and the action executed by the agent.
 - If the environment is deterministic except for the actions of other agents, then the environment is **strategic**
 - Deterministic environments can appear stochastic to an agent (e.g., when only partially observable)
- **Episodic** (vs. **sequential**):
 - An agent's action is divided into atomic episodes. Decisions do not depend on previous decisions/actions.

Environment types

- **Static** (vs. **dynamic**):
 - The environment is unchanged while an agent is deliberating.
 - The environment is **semidynamic** if the environment itself does not change with the passage of time but the agent's performance score does
- **Discrete** (vs. **continuous**):
 - A discrete set of distinct, clearly defined percepts and actions.
 - How we **represent** or **abstract** or **model** the world
- **Single agent** (vs. **multi-agent**):
 - An agent operating by itself in an environment. Does the other agent interfere with my performance measure?

Characteristics of environments

	Fully observable?	Deterministic	Episodic	Static	Discrete?	Single agent?
Solitaire						
Driving						
Internet shopping						
Medical diagnosis						

Characteristics of environments

	Fully observable?	Deterministic?	Episodic?	Static?	Discrete?	Single agent?
Solitaire	No	Yes	Yes	Yes	Yes	Yes
Driving	No	No	No	No	No	No
Internet shopping						
Medical diagnosis						

Characteristics of environments

	Fully observable?	Deterministic?	Episodic?	Static?	Discrete?	Single agent?
Solitaire	No	Yes	Yes	Yes	Yes	Yes
Driving	No	No	No	No	No	No
Internet shopping	No	No	No	No	Yes	No
Medical diagnosis						

Characteristics of environments

	Fully observable?	Deterministic?	Episodic?	Static?	Discrete?	Single agent?
Solitaire	No	Yes	Yes	Yes	Yes	Yes
Driving	No	No	No	No	No	No
Internet shopping	No	No	No	No	Yes	No
Medical diagnosis	No	No	No	No	No	Yes

→ Lots of real-world domains fall into the hardest case!

task environm.	observabl e	deterministic / stochastic	episodic/ sequential	static/ dynamic	discrete/ continuous	agents
crossword puzzle	fully	determ.	sequential	static	discrete	single
chess with clock	fully	strategic	sequential	semi	discrete	multi
poker						
taxi driving	partial	stochastic	sequential	dynamic	continuous	multi
medical diagnosis						
image analysis	fully	determ.	episodic	semi	continuous	single
partpicking robot	partial	stochastic	episodic	dynamic	continuous	single
refinery controller	partial	stochastic	sequential	dynamic	continuous	single
interact. tutor	partial	stochastic	sequential	dynamic	discrete	multi

What is the environment for the DARPA Challenge?

- Agent = robotic vehicle
- Environment = 130-mile route through desert
 - Observable?
 - Deterministic?
 - Episodic?
 - Static?
 - Discrete?
 - Agents?

Agent types

- Five basic types in order of increasing generality:
 - Table Driven agent
 - Simple reflex agents
 - Model-based reflex agents
 - Goal-based agents
 - Problem-solving agents
 - Utility-based agents
 - Can distinguish between different goals
 - Learning agents

Some agent types

- **Table-driven agents**
 - use a percept sequence/action table in memory to find the next action. They are implemented by a (large) **lookup table**. It is not autonomous.
- **Simple reflex agents**
 - are based on **condition-action rules**, implemented with an appropriate production system. They are stateless devices which do not have memory of past world states. It can not save history.
- **Agents with memory**
 - have **internal state**, which is used to keep track of past states of the world.
- **Agents with goals**
 - are agents that, in addition to state information, have **goal information** that describes desirable situations. Agents of this kind take future events into consideration. **Never thinks about cost.**
- **Utility-based agents**
 - base their decisions on **classic axiomatic utility theory** in order to act rationally. **Always thinks about cost.**

Table-driven/reflex agent Architecture

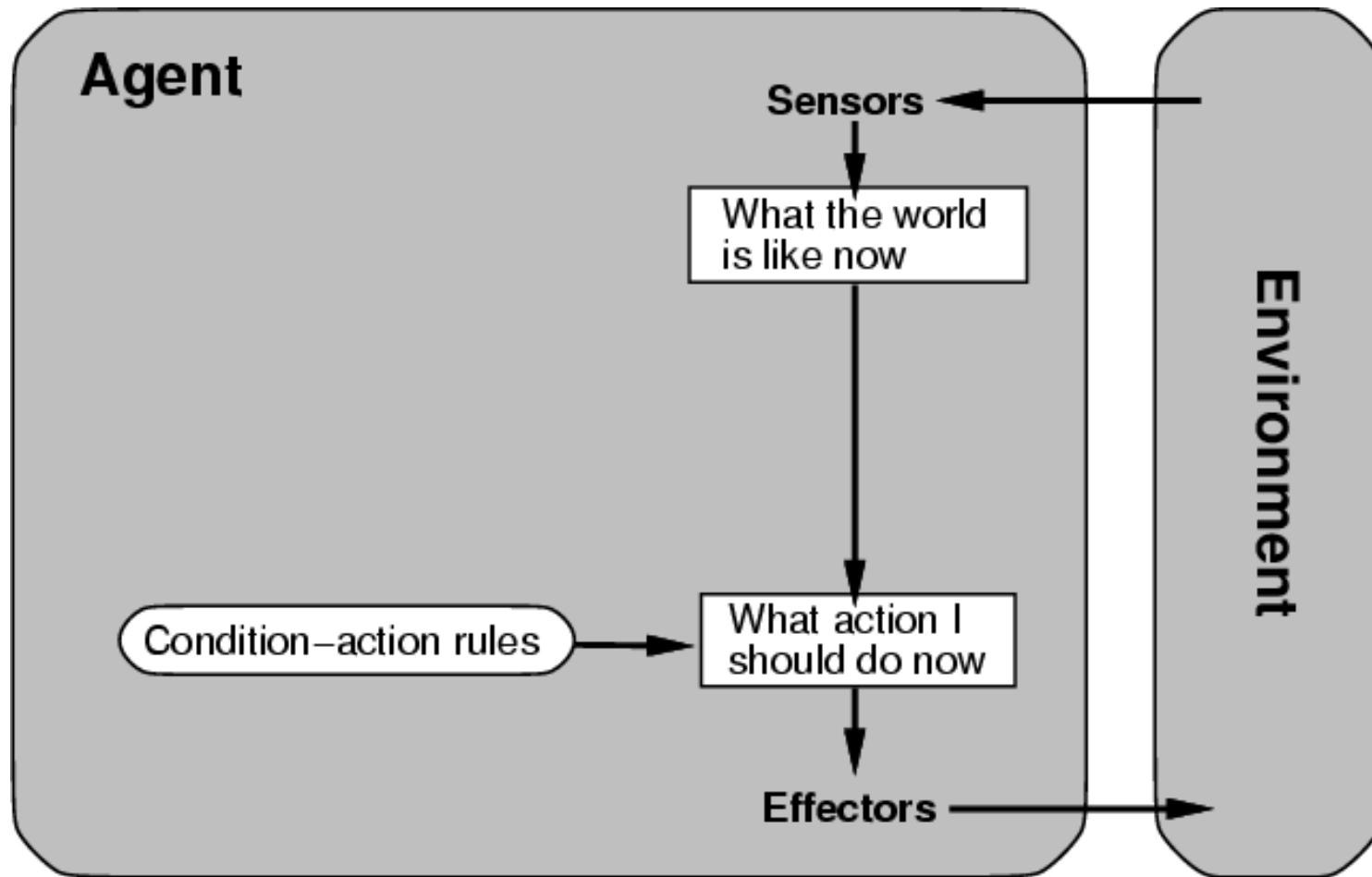


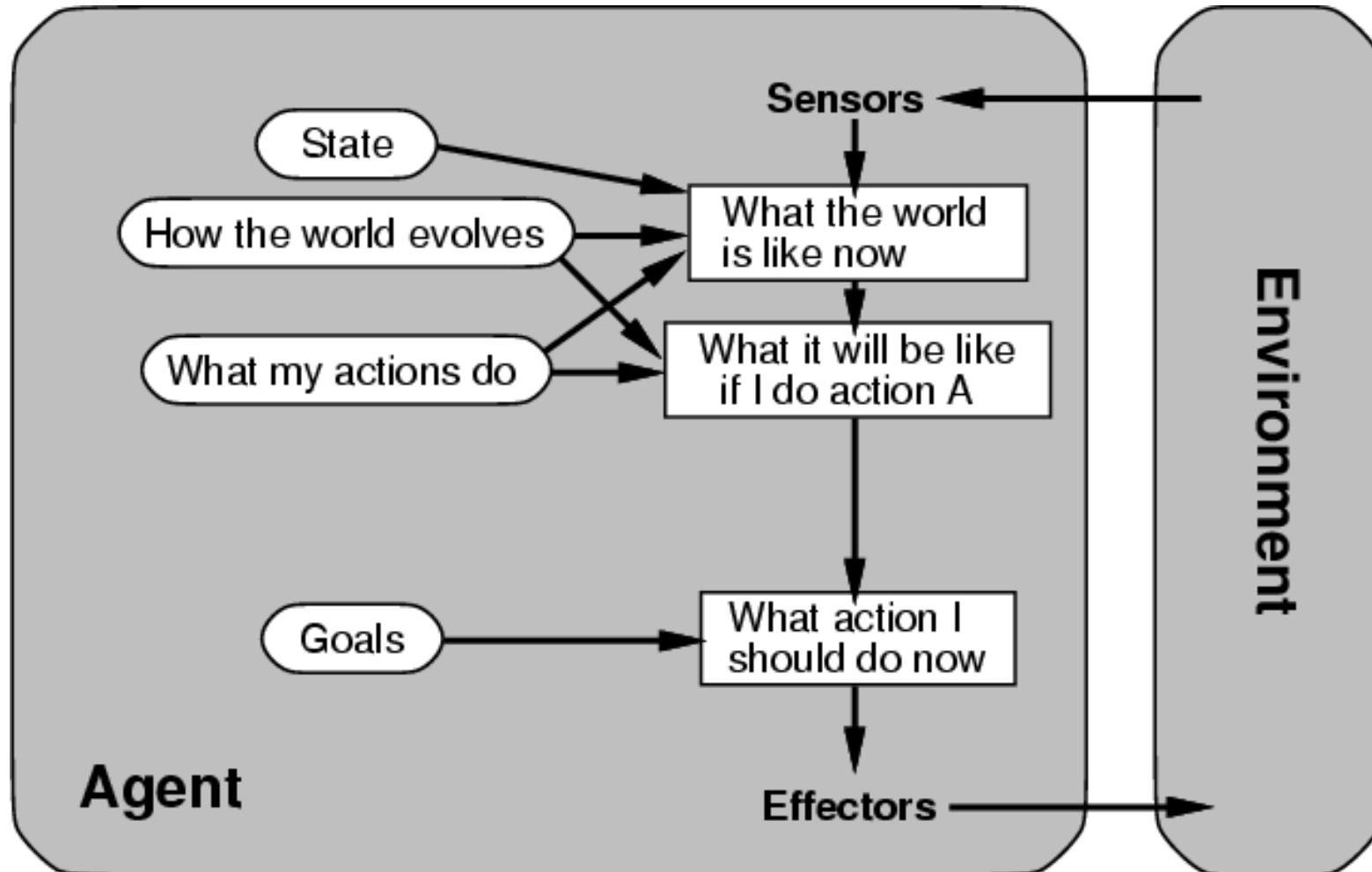
Table-driven agents

- **Table lookup** of percept-action pairs mapping from every possible perceived state to the optimal action for that state
- **Problems**
 - Too big to generate and to store (Chess has about 10^{120} states, for example)
 - No knowledge of non-perceptual parts of the current state
 - Not adaptive to changes in the environment; requires entire table to be updated if changes occur
 - Looping: Can't make actions conditional on previous actions/states

Simple reflex agents

- **Rule-based reasoning** to map from percepts to optimal action; each rule handles a collection of perceived states
- **Problems**
 - Still usually too big to generate and to store
 - Still not adaptive to changes in the environment; requires collection of rules to be updated if changes occur
 - Still can't make actions conditional on previous state

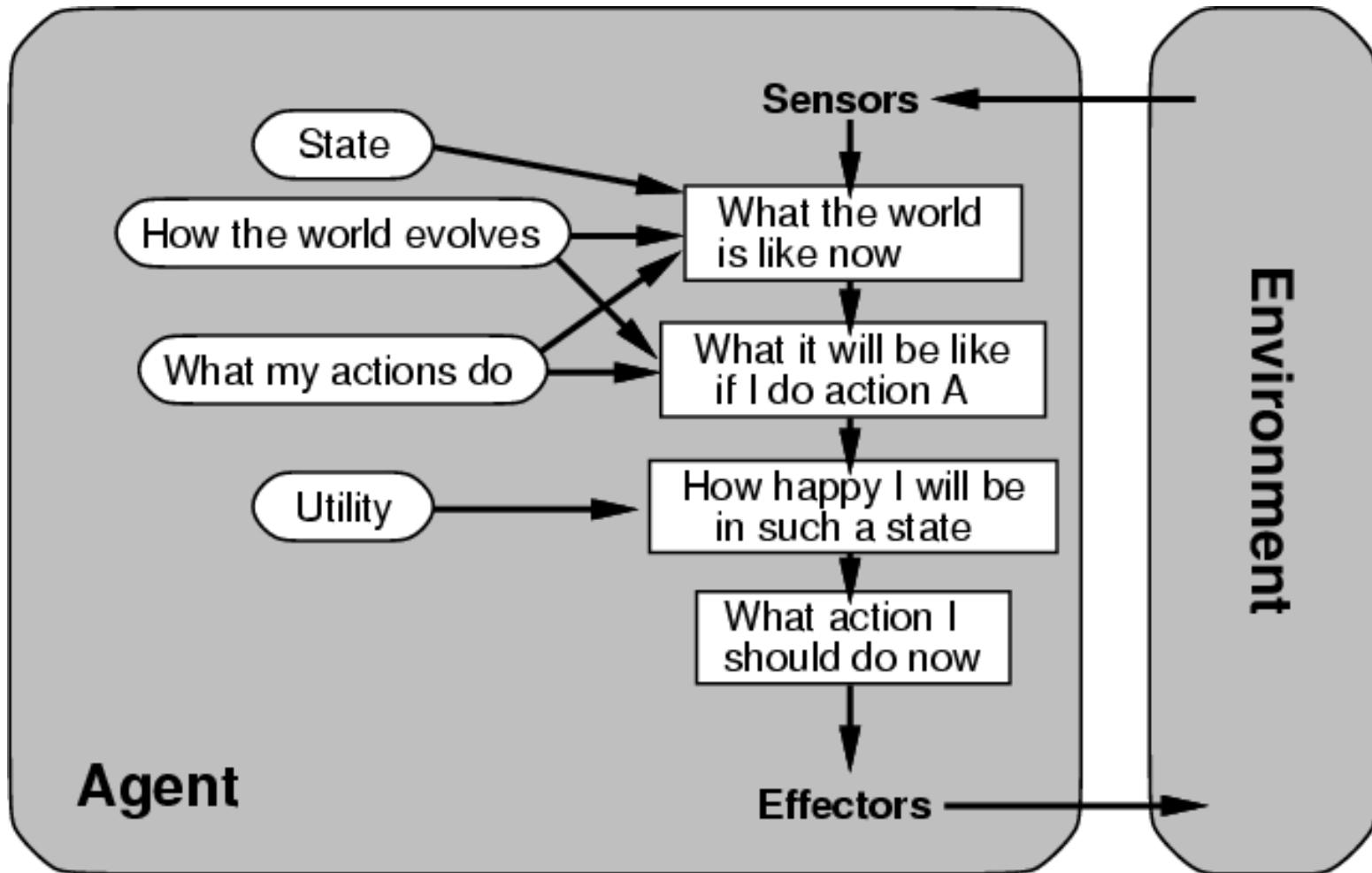
Goal-based agent: Architecture



Goal-based agents

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

Complete utility-based agent



Utility-based agents

- When there are multiple possible alternatives, how to decide which one is best?
- A goal specifies a crude distinction between a happy and unhappy state, but often need a more general performance measure that describes “degree of happiness.”
- Utility function **U: State → Reals** indicating a measure of success or happiness when at a given state.

Summary

- An **agent** perceives and acts in an environment, has an architecture, and is implemented by an agent program.
- An **ideal agent** always chooses the action which maximizes its expected performance, given its percept sequence so far.
- An **autonomous agent** uses its own experience rather than built-in knowledge of the environment by the designer.

Summary (Contd.)

- An **agent program** maps from percept to action and updates its internal state.
 - **Reflex agents** respond immediately to percepts.
 - **Goal-based agents** act in order to achieve their goal(s).
 - **Utility-based agents** maximize their own utility function.
- **Representing knowledge** is important for successful agent design.
- The most challenging environments are
 - partially observable
 - stochastic
 - sequential
 - Dynamic
 - continuous
 - contain multiple intelligent agents

Chapter 3

Problem Solving

By Searching

Problem-Solving Agents

- Intelligent agents can solve problems by searching a state-space
- State-space Model
 - the agent's model of the world
 - usually a set of discrete states
 - e.g., in driving, the states in the model could be towns/cities
- Goal State(s)
 - a goal is defined as a desirable state for an agent
 - there may be many states which satisfy the goal test
 - e.g., drive to a town with a ski-resort
 - or just one state which satisfies the goal
 - e.g., drive to Mammoth
- Operators (actions, successor function)
 - operators are legal actions which the agent can take to move from one state to another

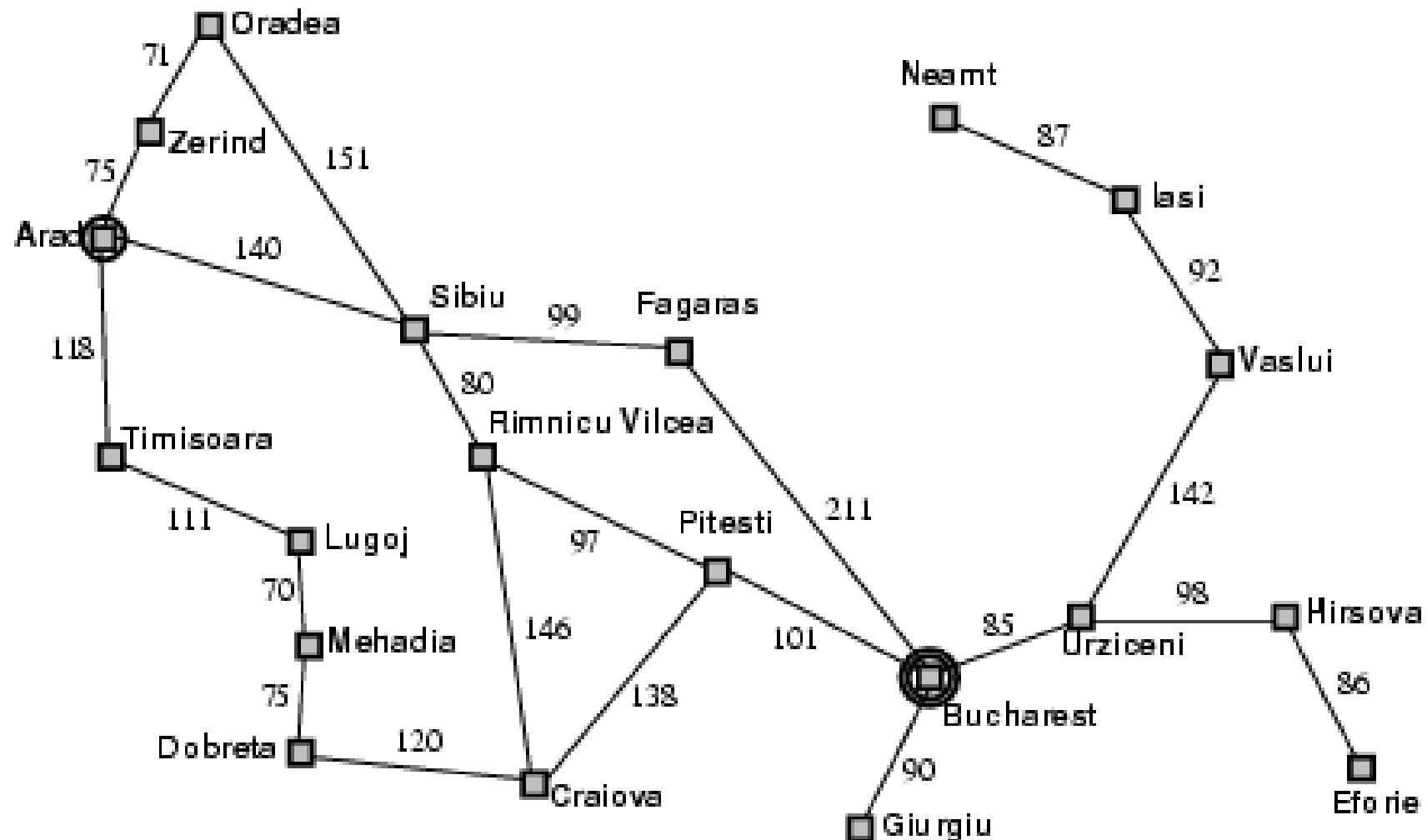
Initial Simplifying Assumptions

- Environment is static
 - no changes in environment while problem is being solved
- Environment is observable
- Environment and actions are discrete
 - (typically assumed, but we will see some exceptions)
- Environment is deterministic

Example: Traveling in Romania

- On holiday in Romania; currently in Arad
- Flight leaves tomorrow from Bucharest
- Formulate goal:
 - be in Bucharest
- Formulate problem:
 - **states**: various cities
 - **actions/operators**: drive between cities
- Find solution
 - By searching through states to find a goal
 - sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest
- Execute states that lead to a solution

Example: Traveling in Romania



State-Space Problem Formulation

A **problem** is defined by four items:

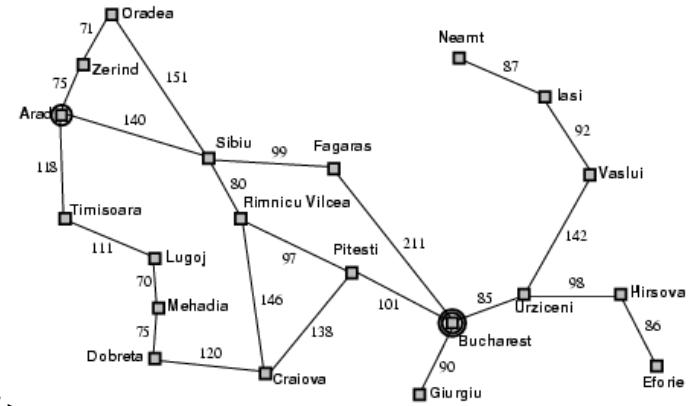
1. **initial state** e.g., "at Arad"
2. **actions** or successor function

$S(x)$ = set of action-state pairs
e.g., $S(Arad) = \{ \langle Arad \rightarrow Zerind, Zerind \rangle, \dots \}$

3. **goal test** (or set of goal states)
e.g., $x = \text{"at Bucharest", Checkmate}(x)$
4. **path cost** (additive)

e.g., sum of distances, number of actions executed, etc.
 $c(x,a,y)$ is the step cost, assumed to be ≥ 0

A **solution** is a sequence of actions leading from the initial state to a goal state



Example: Formulating the Navigation Problem

- Set of States
 - individual cities
 - e.g., Irvine, SF, Las Vegas, Reno, Boise, Phoenix, Denver
- Operators
 - freeway routes from one city to another
 - e.g., Irvine to SF via 5, SF to Seattle, etc
- Start State
 - current city where we are, Irvine
- Goal States
 - set of cities we would like to be in
 - e.g., cities which are closer than Irvine
- Solution
 - a specific goal city, e.g., Boise
 - a sequence of operators which get us there,
 - e.g., Irvine to SF via 5, SF to Reno via 80, etc

Abstraction

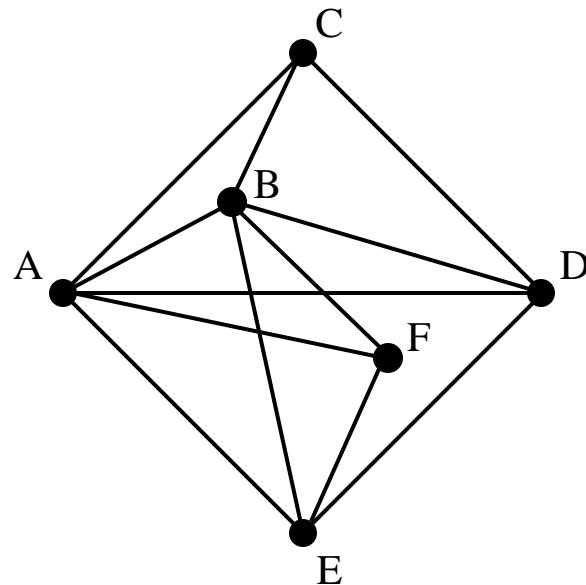
- Definition of Abstraction:
Process of removing irrelevant detail to create an abstract representation: ``high-level'', ignores irrelevant details
- Navigation Example: how do we define states and operators?
 - First step is to abstract “the big picture”
 - i.e., solve a map problem
 - nodes = cities, links = freeways/roads (a high-level description)
 - this description is an abstraction of the real problem
 - Can later worry about details like freeway onramps, refueling, etc
- Abstraction is critical for automated problem solving
 - must create an approximate, simplified, model of the world for the computer to deal with: real-world is too detailed to model exactly
 - good abstractions retain all important details

The State-Space Graph

- Graphs:
 - nodes, arcs, directed arcs, paths
- Search graphs:
 - States are nodes
 - operators are directed arcs
 - solution is a path from start S to goal G
- Problem formulation:
 - Give an abstract description of states, operators, initial state and goal state.
- Problem solving:
 - Generate a part of the search space that contains a solution

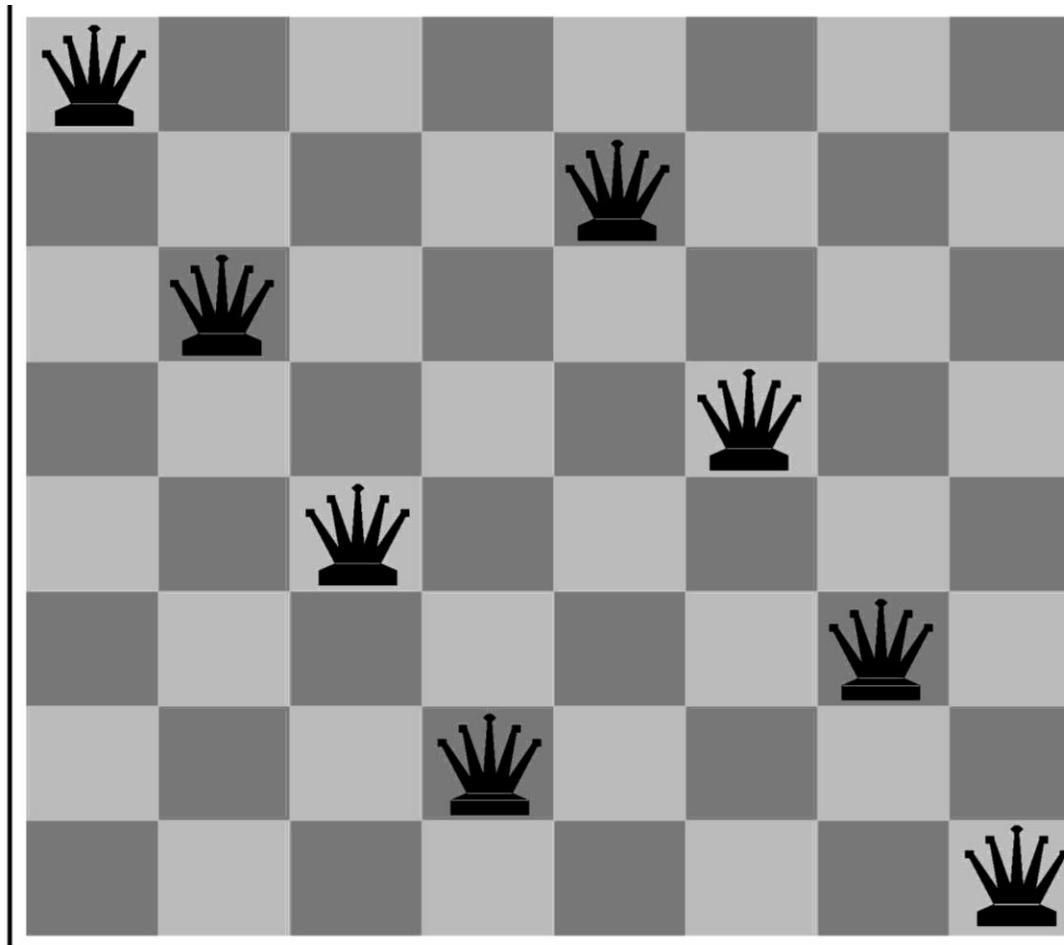
The Traveling Salesperson Problem

- Find the shortest tour that visits all cities without visiting any city twice and return to starting point.
- State: sequence of cities visited
- $S_0 = A$



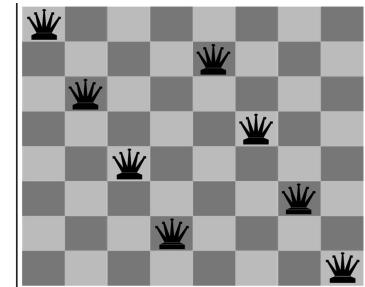
- $G = \text{a complete tour}$

Example: 8-queens problem

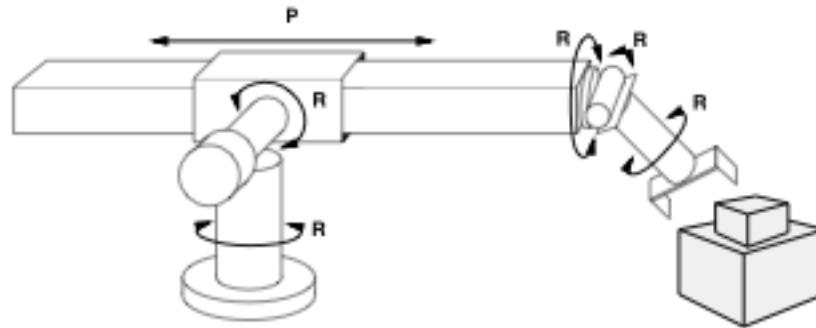


State-Space problem formulation

- states? -any arrangement of $n \leq 8$ queens
 - or arrangements of $n \leq 8$ queens in leftmost n columns, 1 per column, such that no queen attacks any other.
- initial state? no queens on the board
- actions? -add queen to any empty square
 - or add queen to leftmost empty square such that it is not attacked by other queens.
- goal test? 8 queens on the board, none attacked.
- path cost? 1 per move

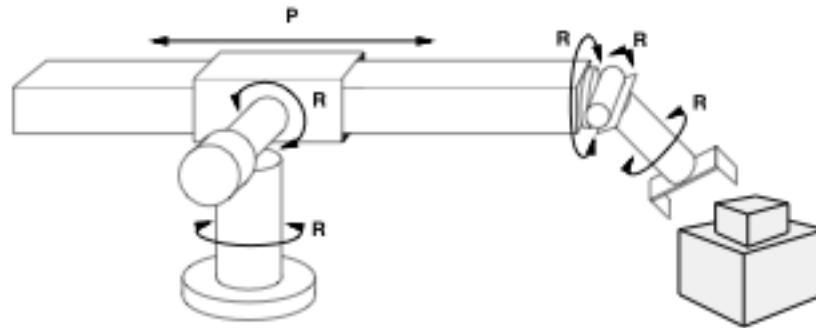


Example: Robot Assembly



- States
- Initial state
- Actions
- Goal test
- Path Cost

Example: Robot Assembly



- States: configuration of robot (angles, positions) and object parts
- Initial state: any configuration of robot and object parts
- Actions: continuous motion of robot joints
- Goal test: object assembled?
- Path Cost: time-taken or number of actions

Learning a spam email classifier

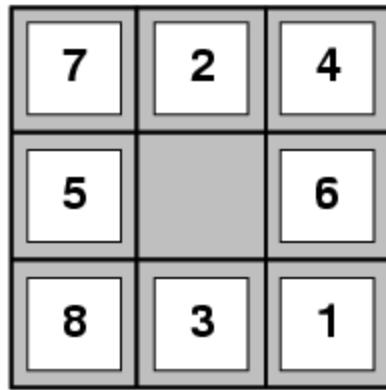
- States
- Initial state
- Actions
- Goal test
- Path Cost

Learning a spam email classifier

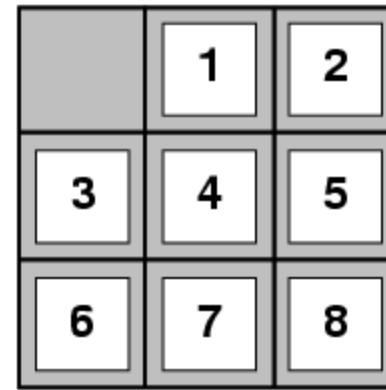
- States: settings of the parameters in our model
- Initial state: random parameter settings
- Actions: moving in parameter space
- Goal test: optimal accuracy on the training data
- Path Cost: time taken to find optimal parameters

(Note: this is an optimization problem – many machine learning problems can be cast as optimization)

Example: 8-puzzle



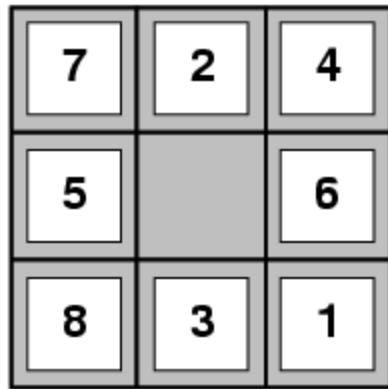
Start State



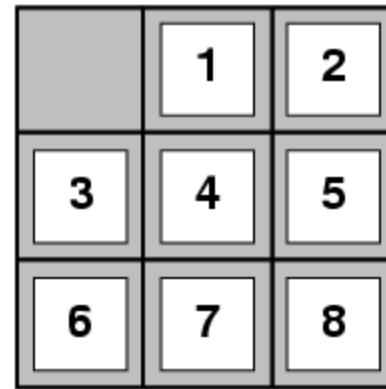
Goal State

- states?
- initial state?
- actions?
- goal test?
- path cost?

Example: 8-puzzle



Start State

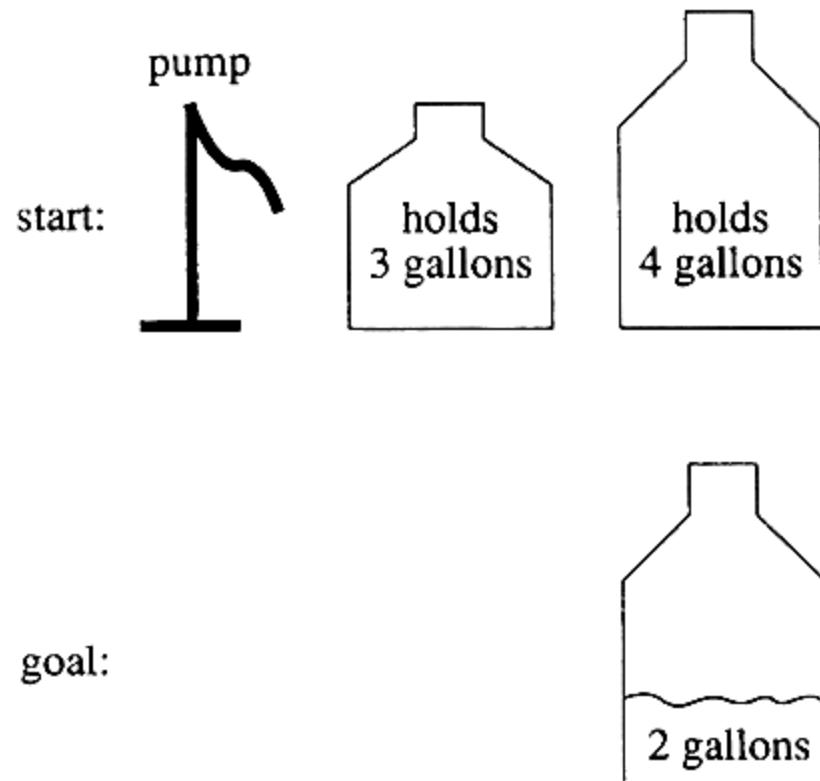


Goal State

- states? locations of tiles
- initial state? given
- actions? move blank left, right, up, down
- goal test? goal state (given)
- path cost? 1 per move

A Water Jug Problem

- You have a 4-gallon and a 3-gallon water jug
- You have a faucet with an unlimited amount of water
- You need to get exactly 2 gallons in 4-gallon jug



Puzzle-solving as Search

- State representation: **(x, y)**
 - x: Contents of four gallon
 - y: Contents of three gallon
- Start state: **(0, 0)**
- Goal state **(2, n)**
- Operators
 - Fill 3-gallon from faucet, fill 4-gallon from faucet
 - Fill 3-gallon from 4-gallon , fill 4-gallon from 3-gallon
 - Empty 3-gallon into 4-gallon, empty 4-gallon into 3-gallon
 - Dump 3-gallon down drain, dump 4-gallon down drain

Production Rules for the Water Jug Problem

- | | |
|--|---|
| 1 $(x,y) \rightarrow (4,y)$
if $x < 4$ | Fill the 4-gallon jug |
| 2 $(x,y) \rightarrow (x,3)$
if $y < 3$ | Fill the 3-gallon jug |
| 3 $(x,y) \rightarrow (x - d,y)$
if $x > 0$ | Pour some water out of the 4-gallon jug |
| 4 $(x,y) \rightarrow (x,y - d)$
if $x > 0$ | Pour some water out of the 3-gallon jug |
| 5 $(x,y) \rightarrow (0,y)$
if $x > 0$ | Empty the 4-gallon jug on the ground |
| 6 $(x,y) \rightarrow (x,0)$
if $y > 0$ | Empty the 3-gallon jug on the ground |
| 7 $(x,y) \rightarrow (4,y - (4 - x))$
if $x + y \geq 4$ and $y > 0$ | Pour water from the 3-gallon jug into the 4-gallon jug until the 4-gallon jug is full |

The Water Jug Problem (cont'd)

$$8 (x, y) \rightarrow (x - (3 - y), 3) \\ \text{if } x + y \geq 3 \text{ and } x > 0$$

Pour water from the 4-gallon jug into the 3-gallon jug until the 3-gallon jug is full

$$9 (x, y) \rightarrow (x + y, 0) \\ \text{if } x + y \leq 4 \text{ and } y > 0$$

Pour all the water from the 3-gallon jug into the 4-gallon jug

$$10 (x, y) \rightarrow (0, x + y) \\ \text{if } x + y \leq 3 \text{ and } x > 0$$

Pour all the water from the 4-gallon jug into the 3-gallon jug

One Solution to the Water Jug Problem

Gallons in the 4-Gallon Jug	Gallons in the 3-Gallon Jug	Rule Applied
0	0	2
0	3	9
3	0	2
3	3	7
4	2	5
0	2	9
2	0	

VLSI Layout Problem

- Require positioning millions of components and connections on a chip to **minimize area**, **minimize circuit delays**, **minimize stray capacitances**, and **maximize manufacturing yield**
- The layout problem comes after the logical design phase and is usually split into two parts: **cell layout** and **channel routing**
- In cell layout, the primitive components of the circuit are grouped into cells, each of which performs some recognized function

VLSI Layout Problem (cont'd)

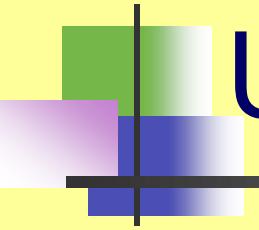
- Each cell has a fixed footprint (size and shape) and requires a certain number of connections to each of the other cells
- The aim is to place the cells on the chip so that they do not overlap and so that there is room for the connecting wires to be placed between the cells
- Channel routing finds a specific route for each wire through the gaps between the cells. These search problems are extremely complex, but definitely worth solving.

Next Topics

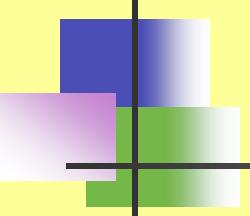
- Uninformed search
 - Breadth-first, depth-first
 - Uniform cost
 - Iterative deepening
- Informed (heuristic) search
 - Greedy best-first
 - A*
 - Memory-bounded heuristic search
 - And more....
- Local search and optimization
 - Hill-climbing
 - Simulated annealing
 - Genetic algorithms

Summary

- Problem-solving agents where search consists of
 - state space
 - operators
 - start state
 - goal states
- Abstraction and problem formulation



Uninformed Search



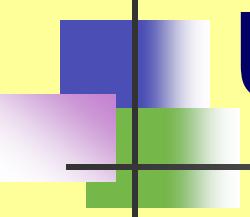
Measuring problem-solving performance

Completeness: Is the algorithm guaranteed to find a solution when there is one?

Optimality: Does the strategy find the optimal solution, as defined on page 68?

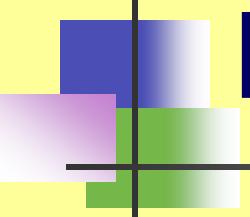
Time complexity: How long does it take to find a solution?

Space complexity: How much memory is needed to perform the search?



Uninformed search strategies

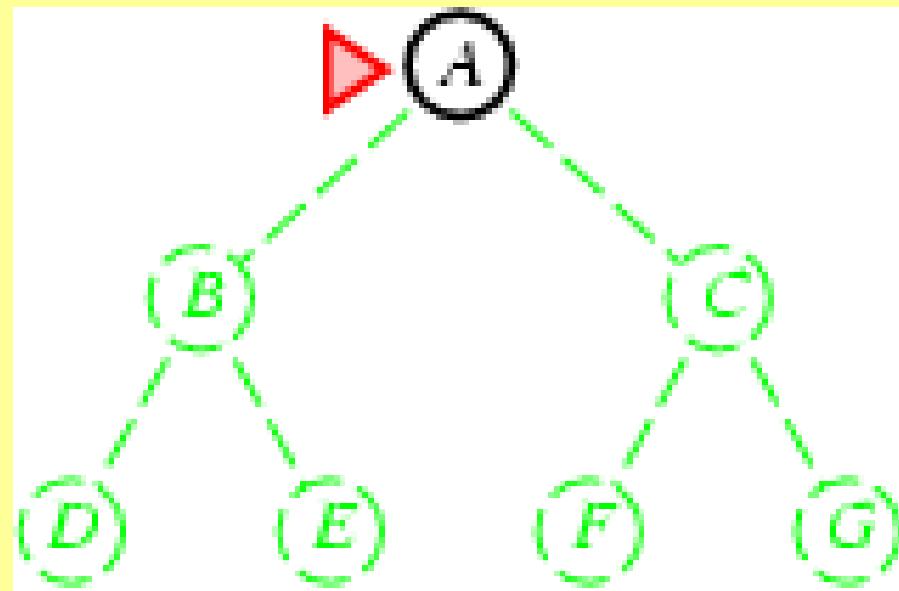
- **Uninformed**: While searching you have no clue whether one non-goal state is better than any other. Your search is blind.
- **Various blind strategies**:
- Breadth-first search
- Uniform-cost search
- Depth-first search
- Iterative deepening search



Breadth-first search

- Expand shallowest unexpanded node
- Implementation:
 - *fringe* is a first-in-first-out (FIFO) queue, i.e., new successors go at end of the queue.

Is A a goal state?



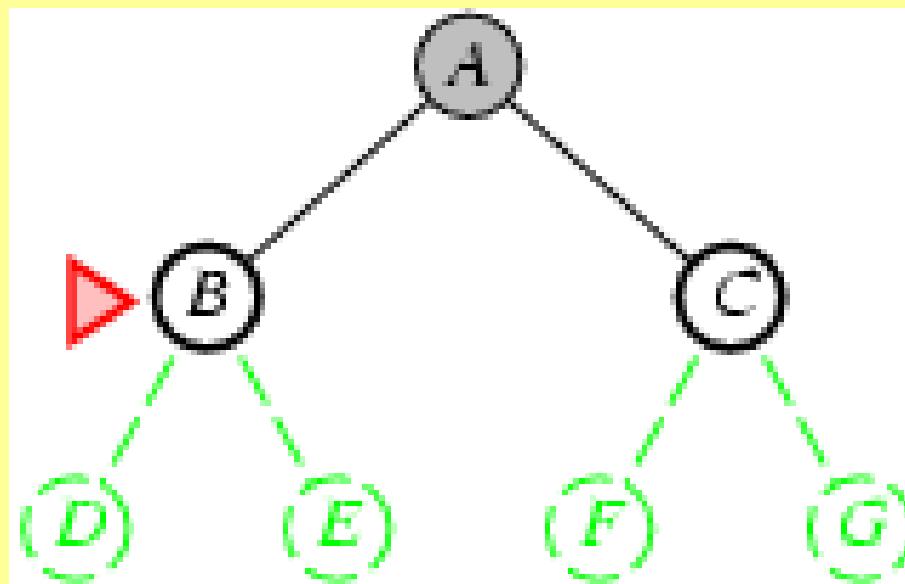
Breadth-first search

- Expand shallowest unexpanded node
- Implementation:
 - *fringe* is a FIFO queue, i.e., new successors go at end

Expand:

fringe = [B,C]

Is B a goal state?



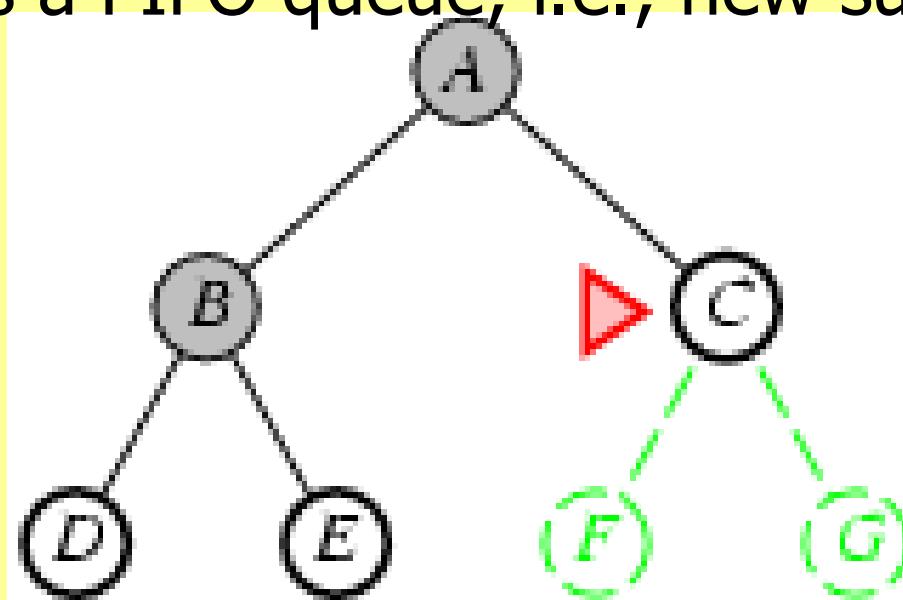
Breadth-first search

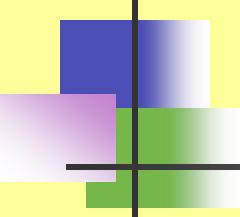
- Expand shallowest unexpanded node
-
- Implementation:
 - *fringe* is a FIFO queue, i.e., new successors go at end

Expand:

fringe=[C,D,E]

Is C a goal state?



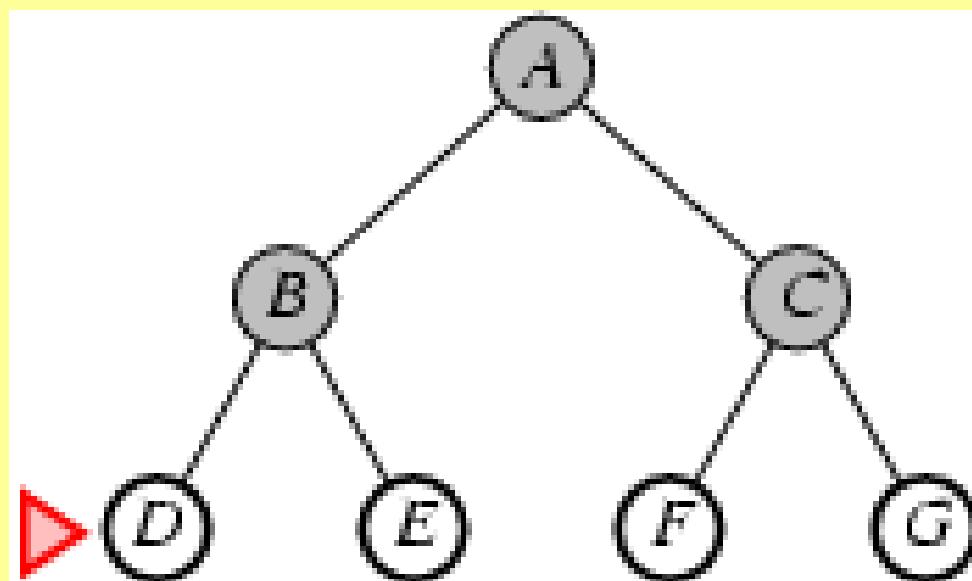


Breadth-first search

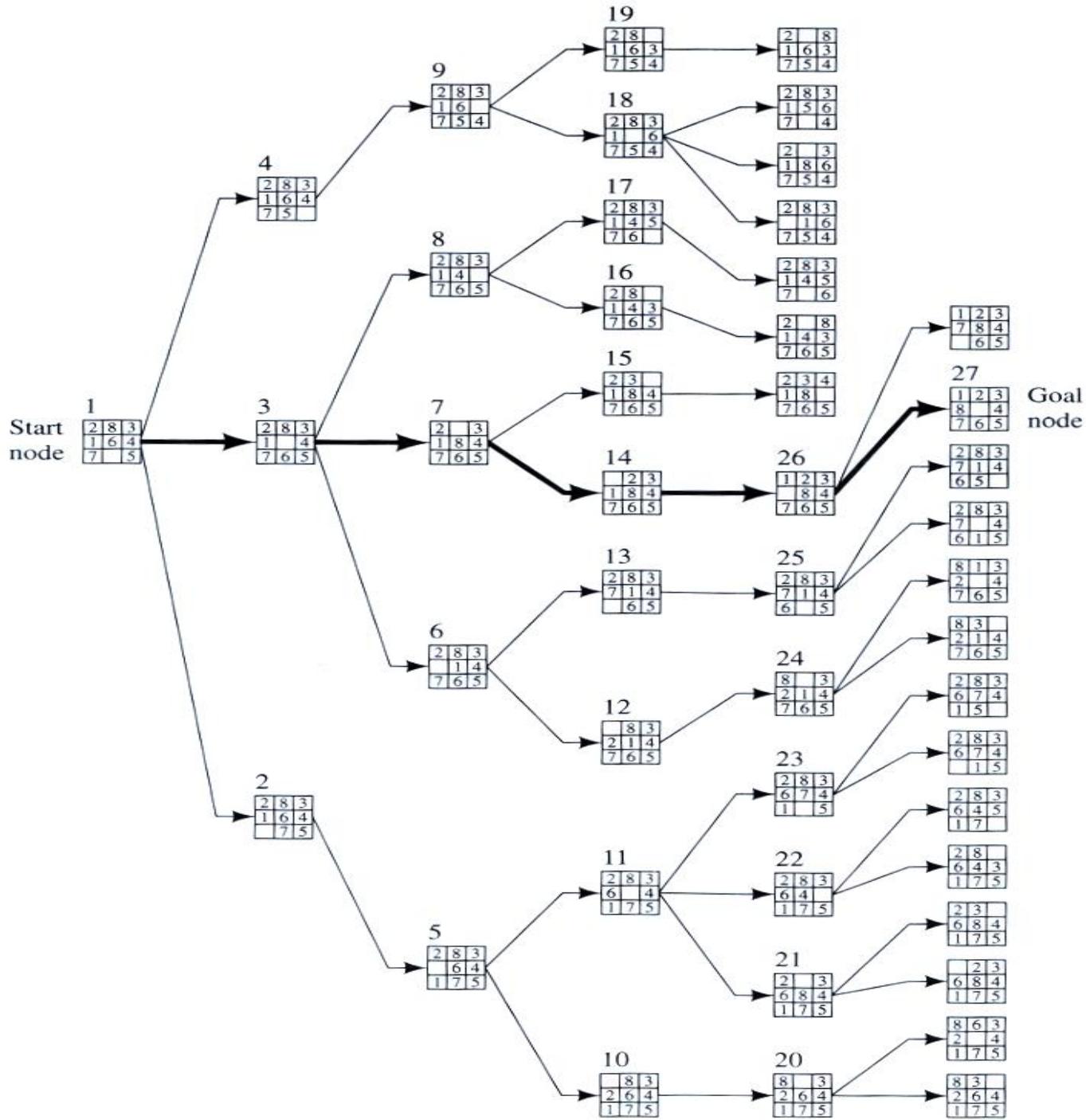
- Expand shallowest unexpanded node
- Implementation:
 - *fringe* is a FIFO queue, i.e., new successors go at end

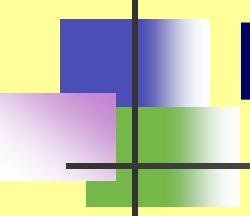
Expand:
fringe=[D,E,F,G]

Is D a goal state?



Example BFS





Properties of breadth-first search

- Complete? Yes it always reaches goal (if b is finite)
- Time? $1+b+b^2+b^3+\dots +b^d + (b^{d+1}-b) = O(b^{d+1})$
(this is the number of nodes we generate)
- Space? $O(b^{d+1})$ (keeps every node in memory, either in fringe or on a path to fringe).
- Optimal? Yes (if we guarantee that deeper solutions are less optimal, e.g. step-cost=1).
- **Space** is the bigger problem (more than time)

Uniform-cost search

Breadth-first is only optimal if step costs is increasing with depth (e.g. constant). Can we guarantee optimality for any step cost?

Uniform-cost Search: Expand node with smallest path cost $g(n)$.

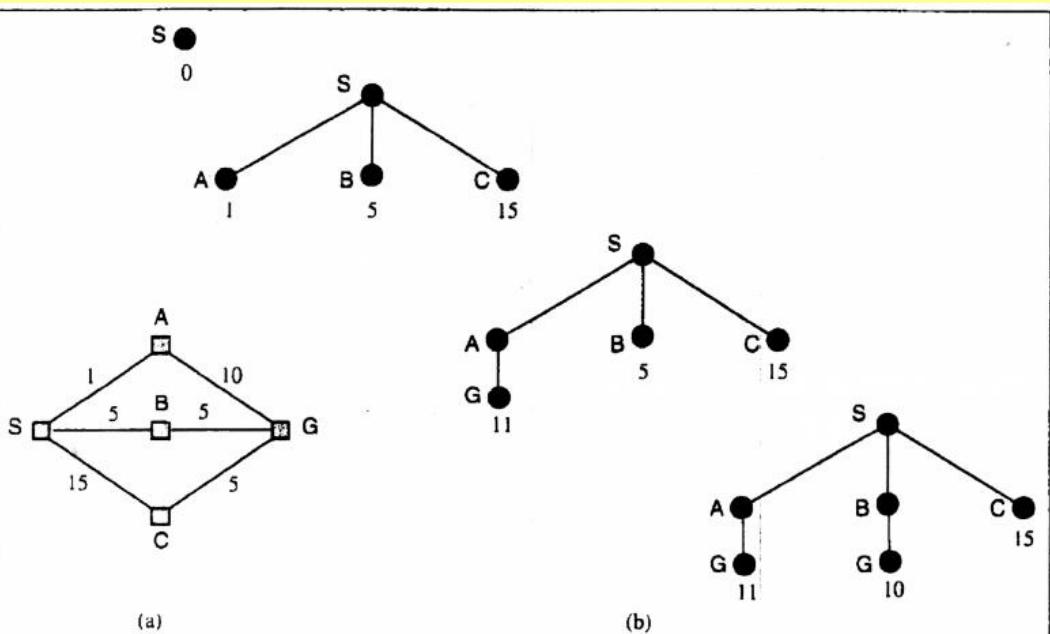
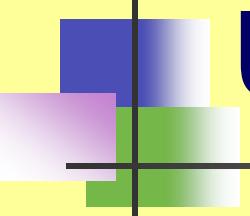


Figure 3.13 A route-finding problem. (a) The state space, showing the cost for each operator. (b) Progression of the search. Each node is labelled with $g(n)$. At the next step, the goal node with $g = 10$ will be selected.



Uniform-cost search

Implementation: *fringe* = queue ordered by path cost
Equivalent to breadth-first if all step costs all equal.

Complete? Yes, if step cost $\geq \varepsilon$
(otherwise it can get stuck in infinite loops)

Time? # of nodes with *path cost* \leq cost of optimal solution.

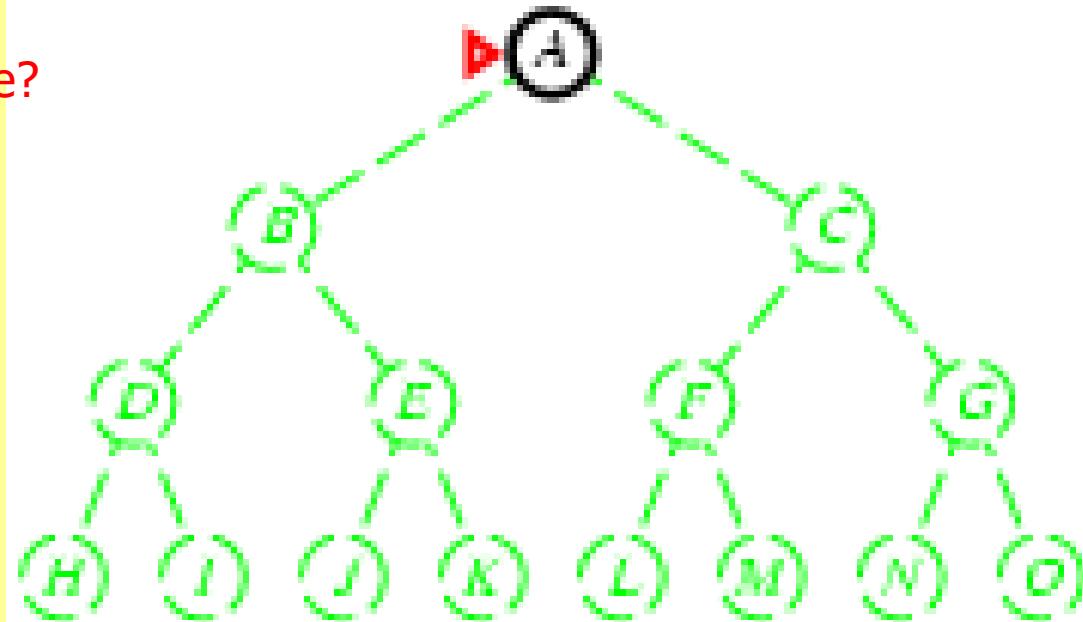
Space? # of nodes on paths with path cost \leq cost of optimal solution.

Optimal? Yes, for any step cost.

Depth-first search

- Expand *deepest* unexpanded node
- Implementation:
 - *fringe* = Last In First Out (LIPO) queue, i.e., put successors at front

Is A a goal state?

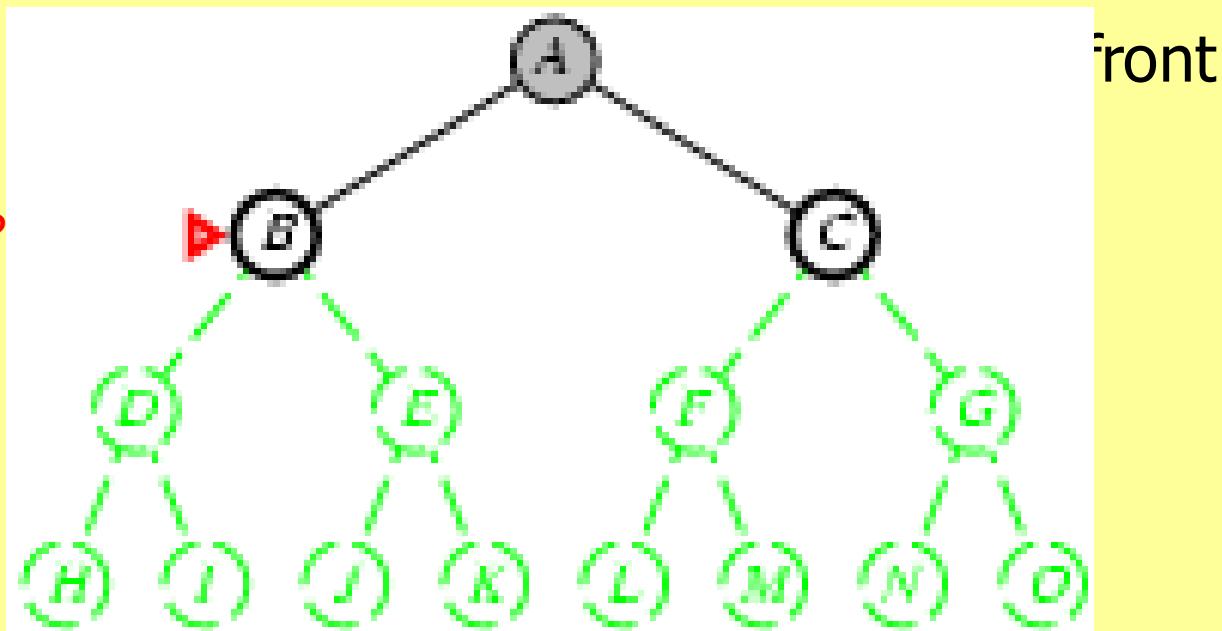


Depth-first search

- Expand deepest unexpanded node
-
- Implementation:

■ *fringe*
queue=[B,C]

Is B a goal state?



Depth-first search

- Expand deepest unexpanded node

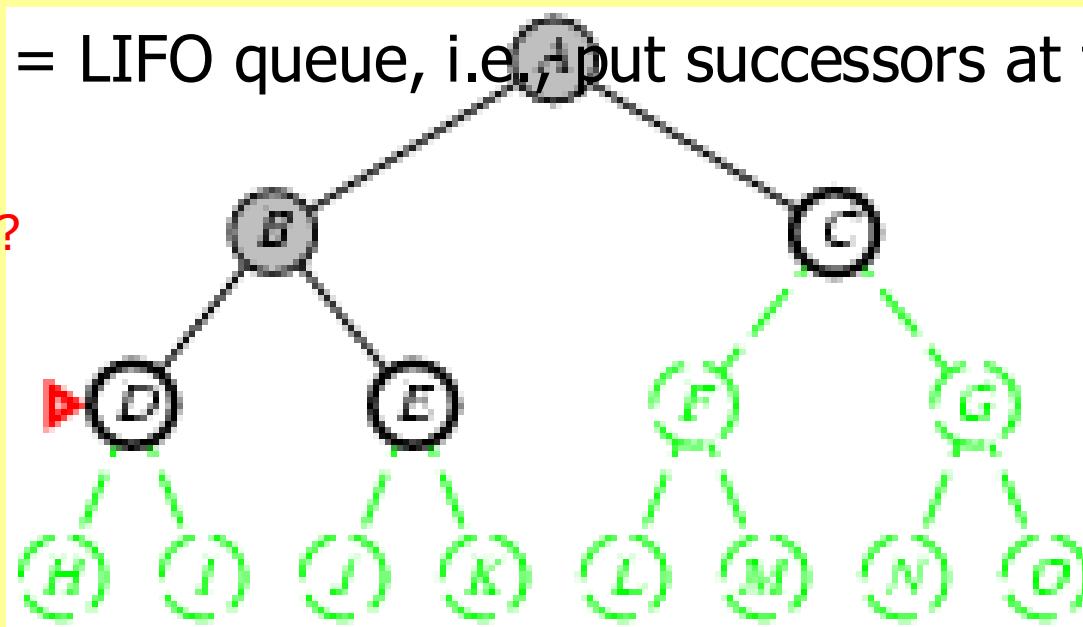
-

- Implementation:

- *fringe* = LIFO queue, i.e., put successors at front

queue=[D,E,C]

Is D = goal state?



Depth-first search

- Expand deepest unexpanded node

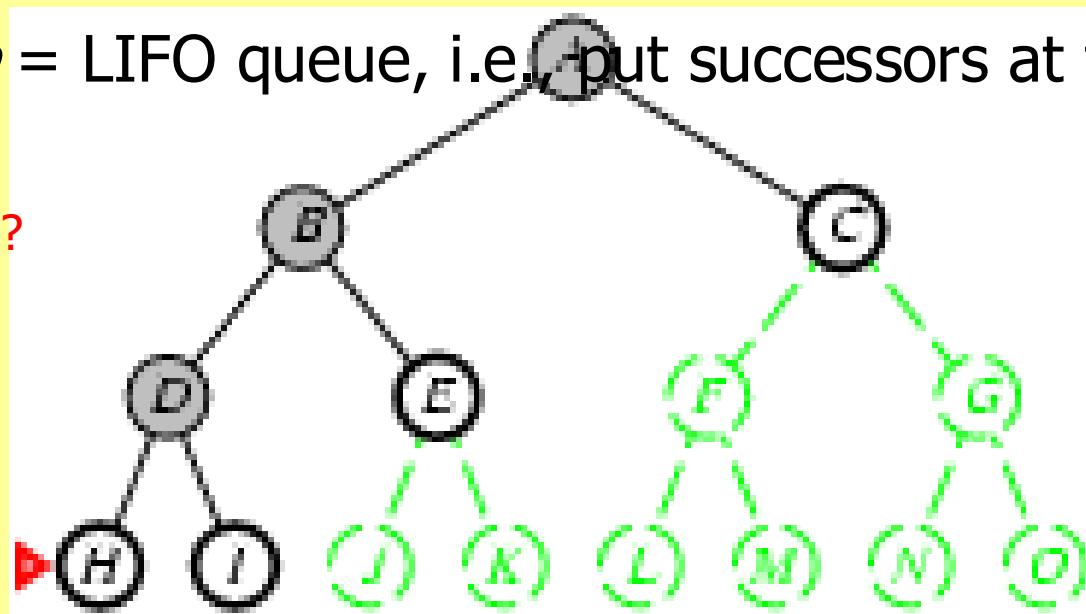
-

- Implementation:

- *fringe* = LIFO queue, i.e., put successors at front

queue=[H,I,E,C]

Is H = goal state?



Depth-first search

- Expand deepest unexpanded node

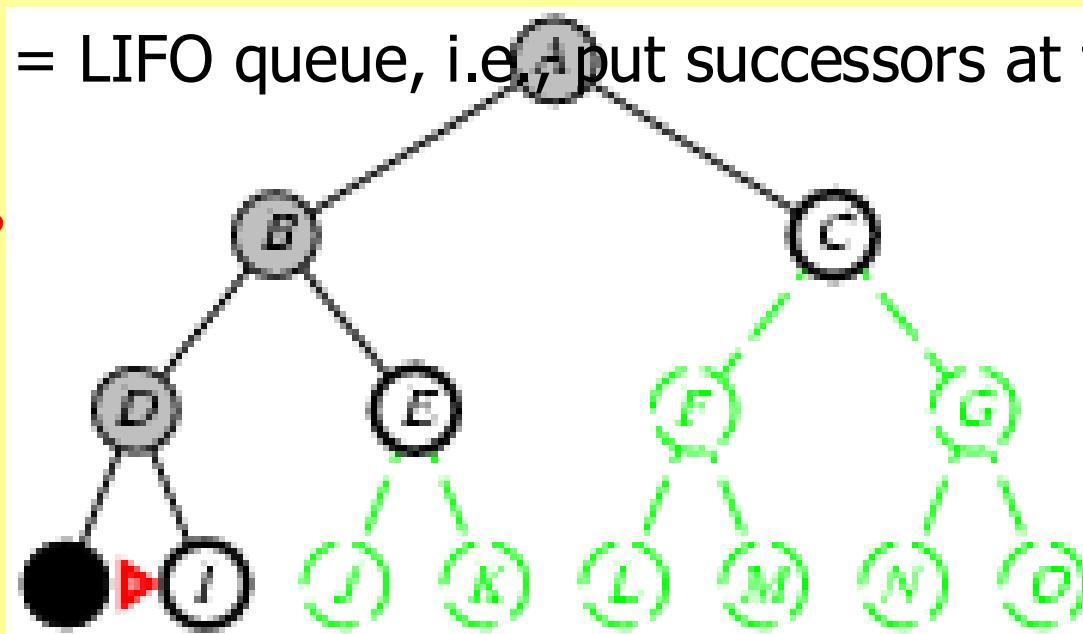
-

- Implementation:

- *fringe* = LIFO queue, i.e., put successors at front

queue=[I,E,C]

Is I = goal state?

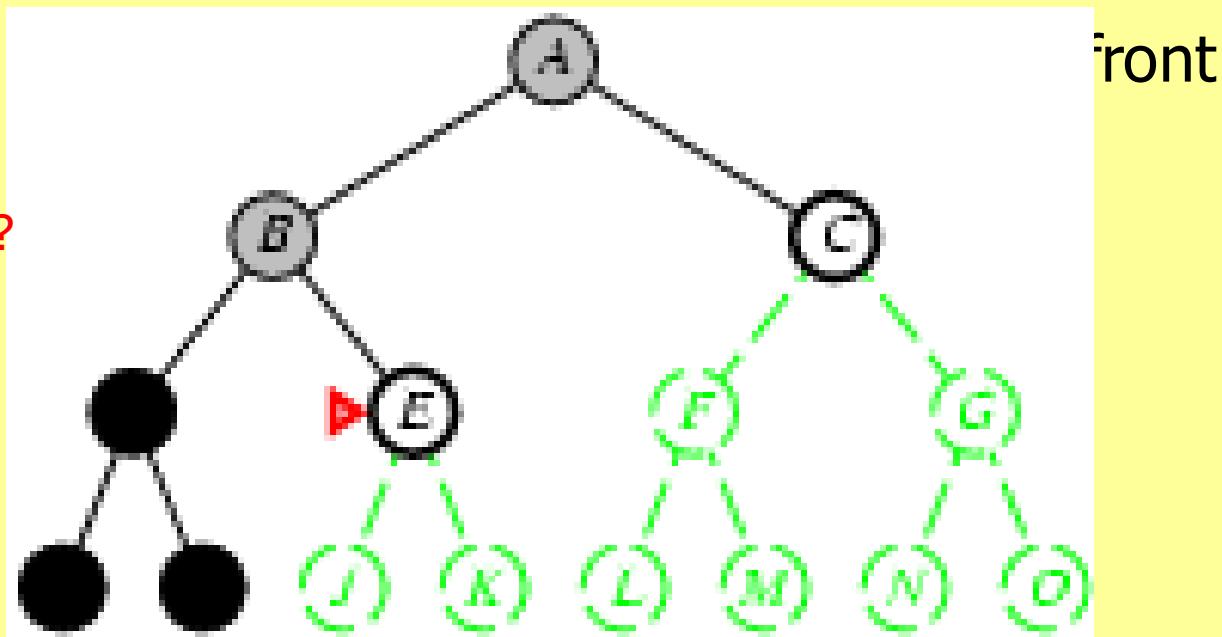


Depth-first search

- Expand deepest unexpanded node
-
- Implementation:

■ *fringe*
queue=[E,C]
■

Is E = goal state?

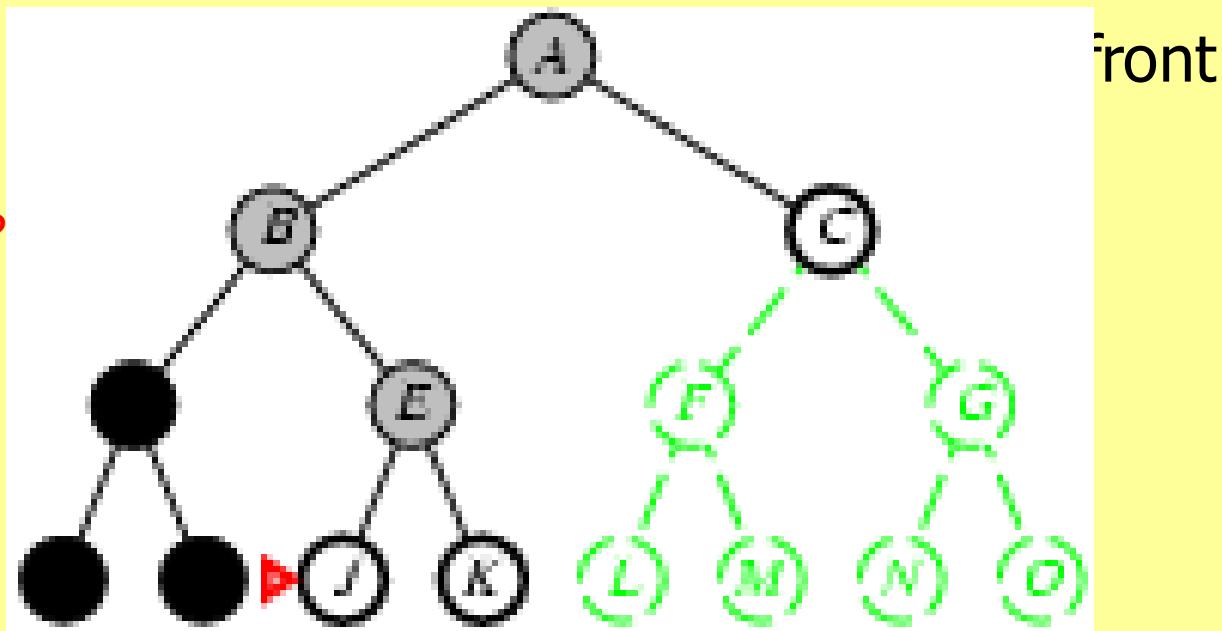


Depth-first search

- Expand deepest unexpanded node
-
- Implementation:

■ *fringe*
queue=[J,K,C]

■ Is J = goal state?

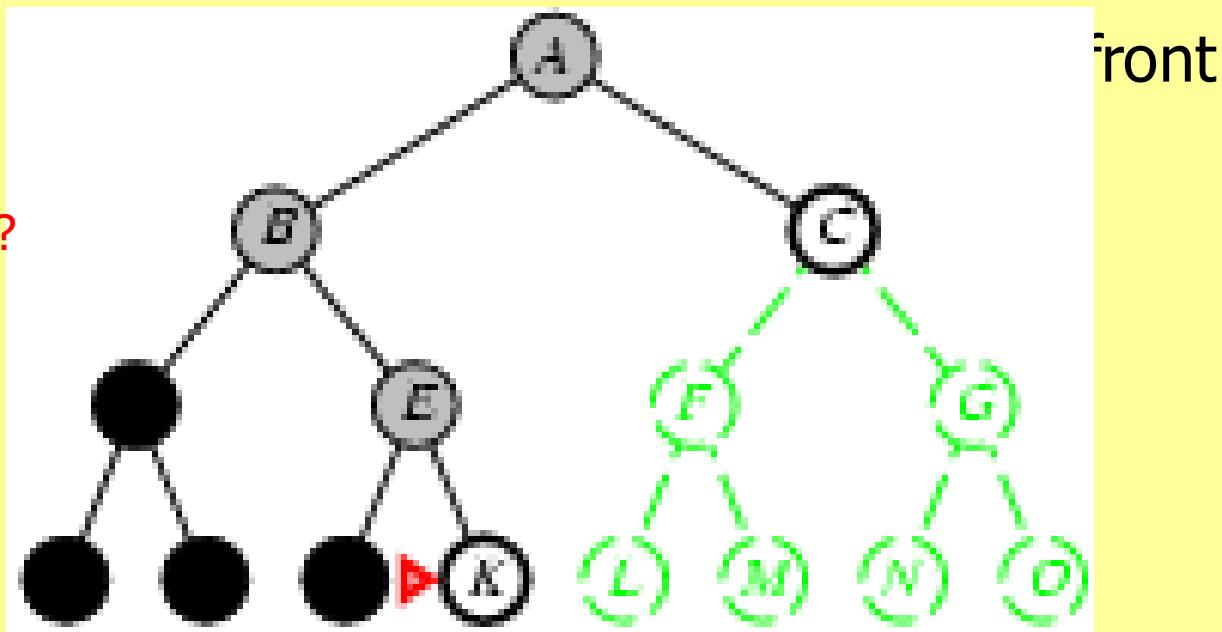


Depth-first search

- Expand deepest unexpanded node
-
- Implementation:

■ *fringe*
queue=[K,C]
■

Is K = goal state?

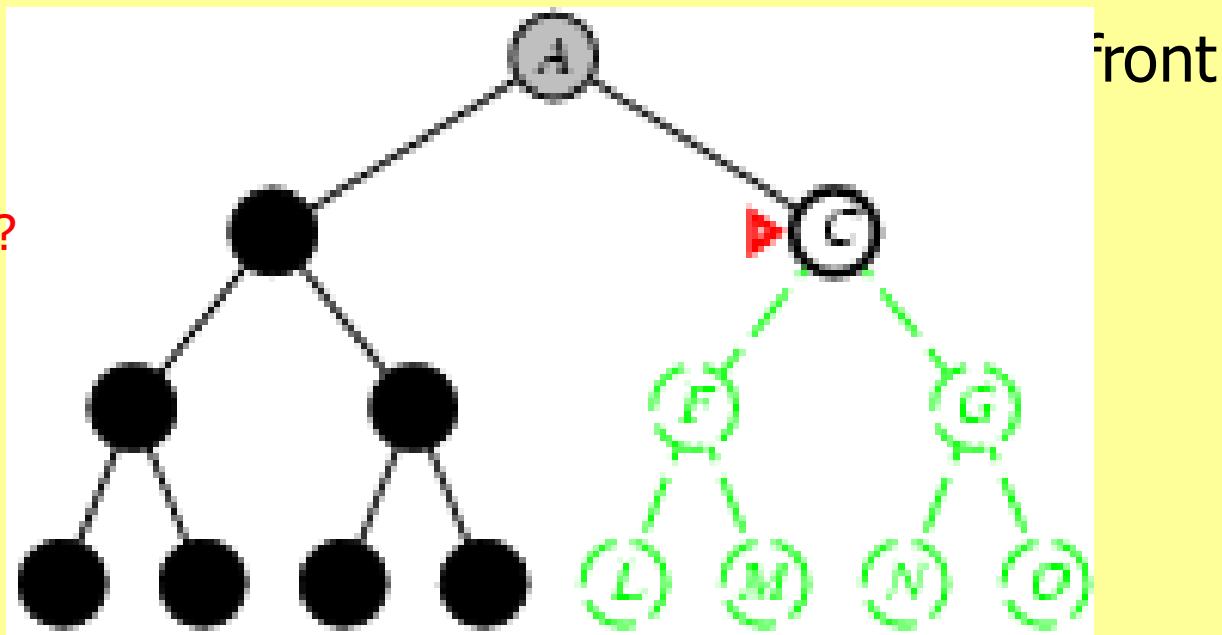


Depth-first search

- Expand deepest unexpanded node
-
- Implementation:

■ *fringe*
queue=[C]
■

Is C = goal state?

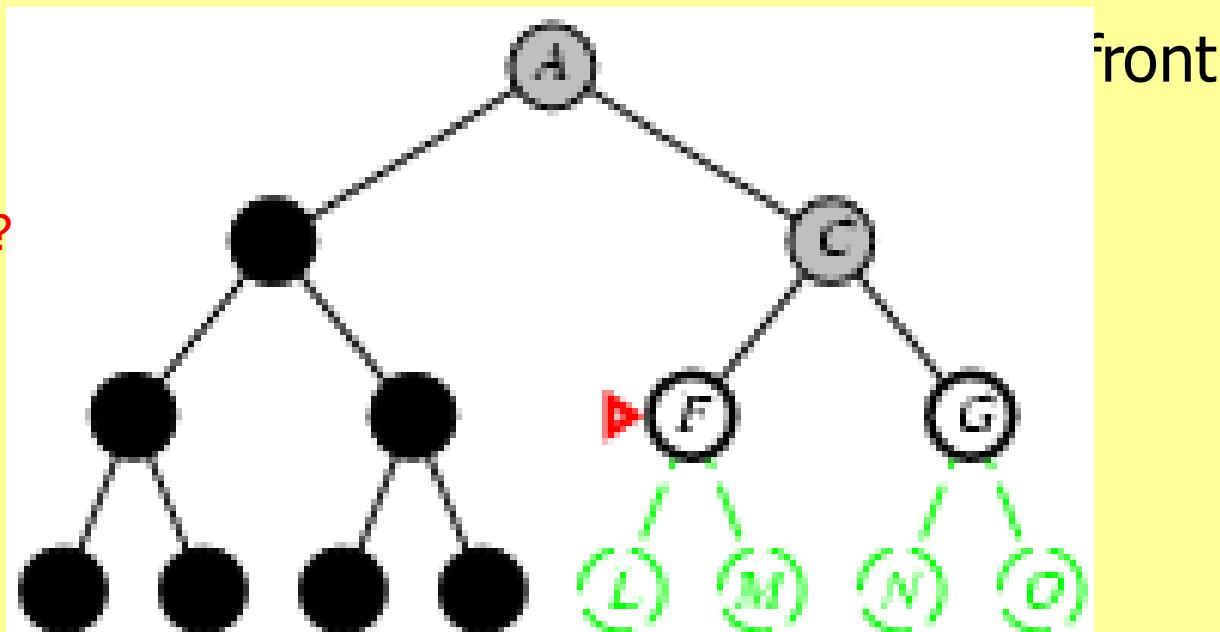


Depth-first search

- Expand deepest unexpanded node
-
- Implementation:

■ *fringe*
queue=[F,G]
■

Is F = goal state?

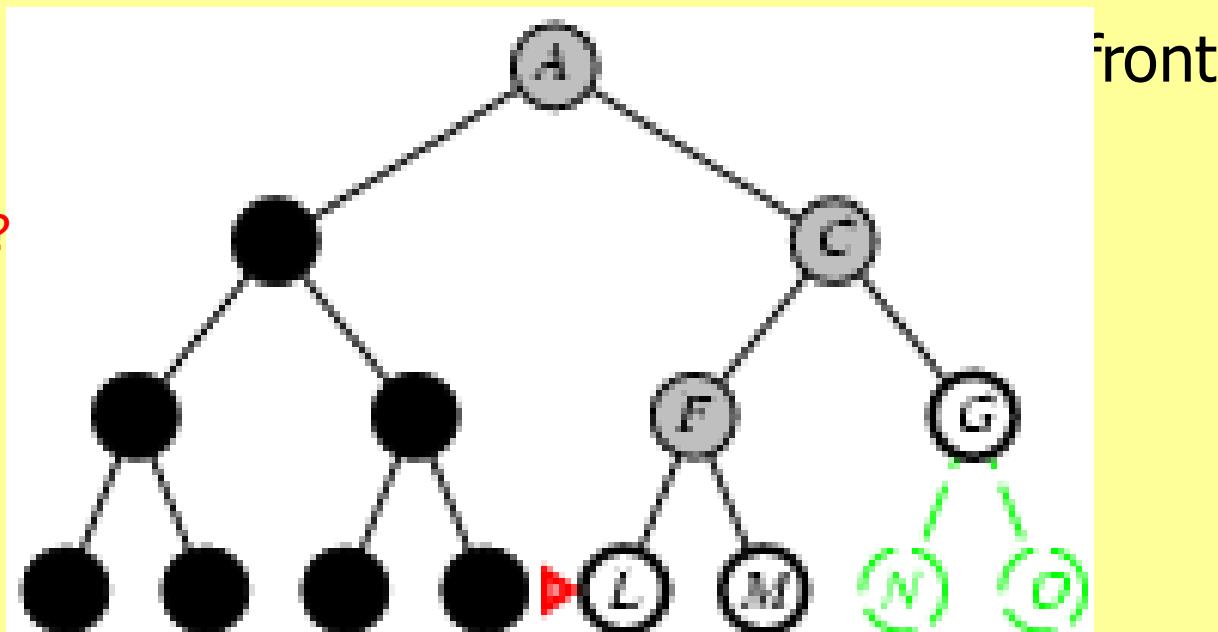


Depth-first search

- Expand deepest unexpanded node
-
- Implementation:

■ *fringe*
queue=[L,M,G]

Is L = goal state?

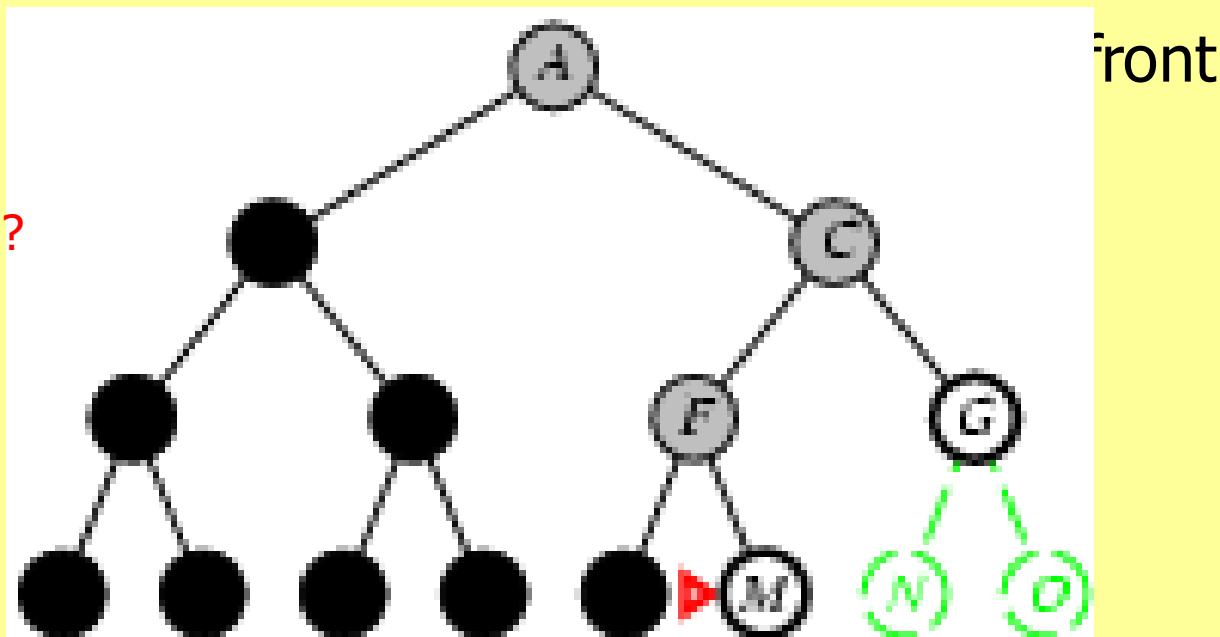


Depth-first search

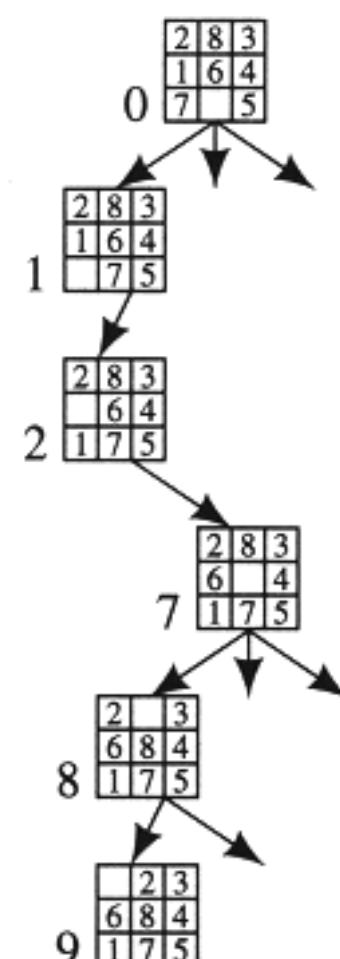
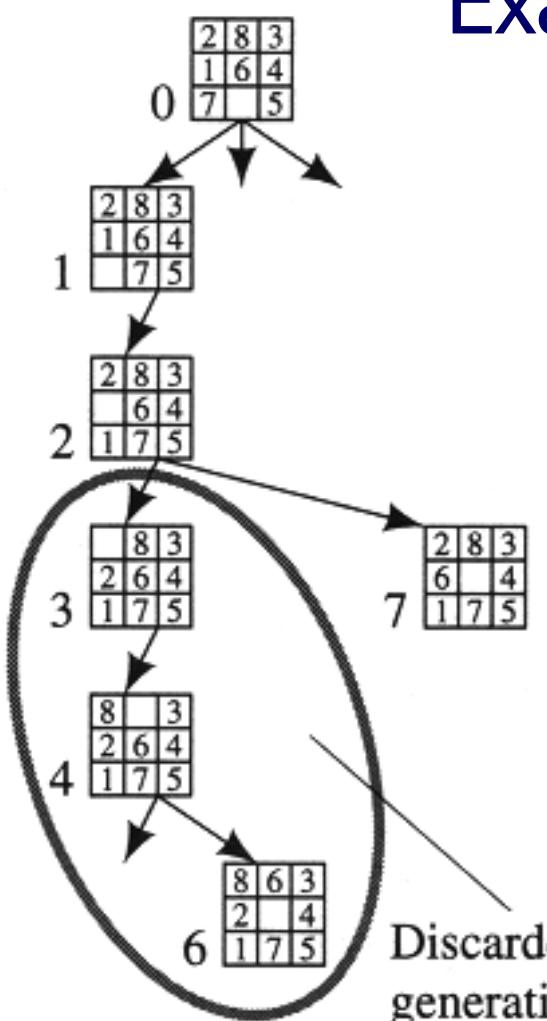
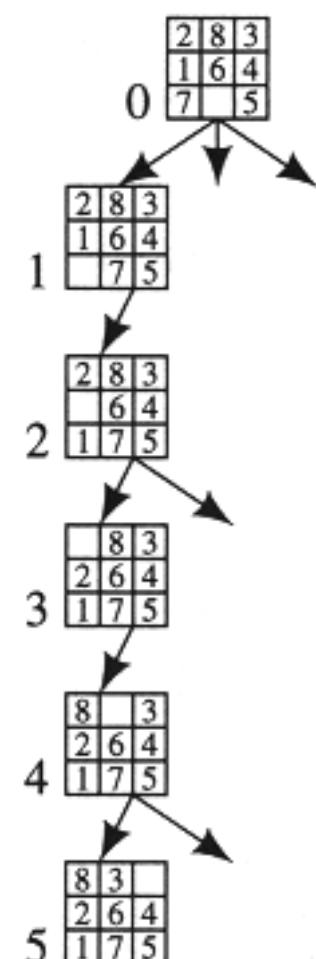
- Expand deepest unexpanded node
-
- Implementation:

■ *fringe*
queue=[M,G]
■

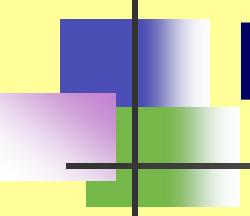
Is M = goal state?



Example DFS

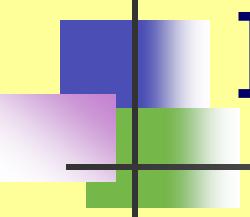


Generation of the First Few Nodes in a Depth-First Search



Properties of depth-first search

- Complete? No: fails in infinite-depth spaces
Can modify to avoid repeated states along path
- Time? $O(b^m)$ with m =maximum depth
- 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! (we only need to remember a single path + expanded unexplored nodes)
- Optimal? No (It may find a non-optimal goal first)

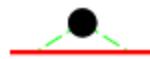


Iterative deepening search

- To avoid the infinite depth problem of DFS, we can decide to only search until depth L , i.e. we don't expand beyond depth L .
→ Depth-Limited Search
- What of solution is deeper than L ? → Increase L iteratively.
→ Iterative Deepening Search
- As we shall see: this inherits the memory advantage of Depth-First search.

Iterative deepening search $L=0$

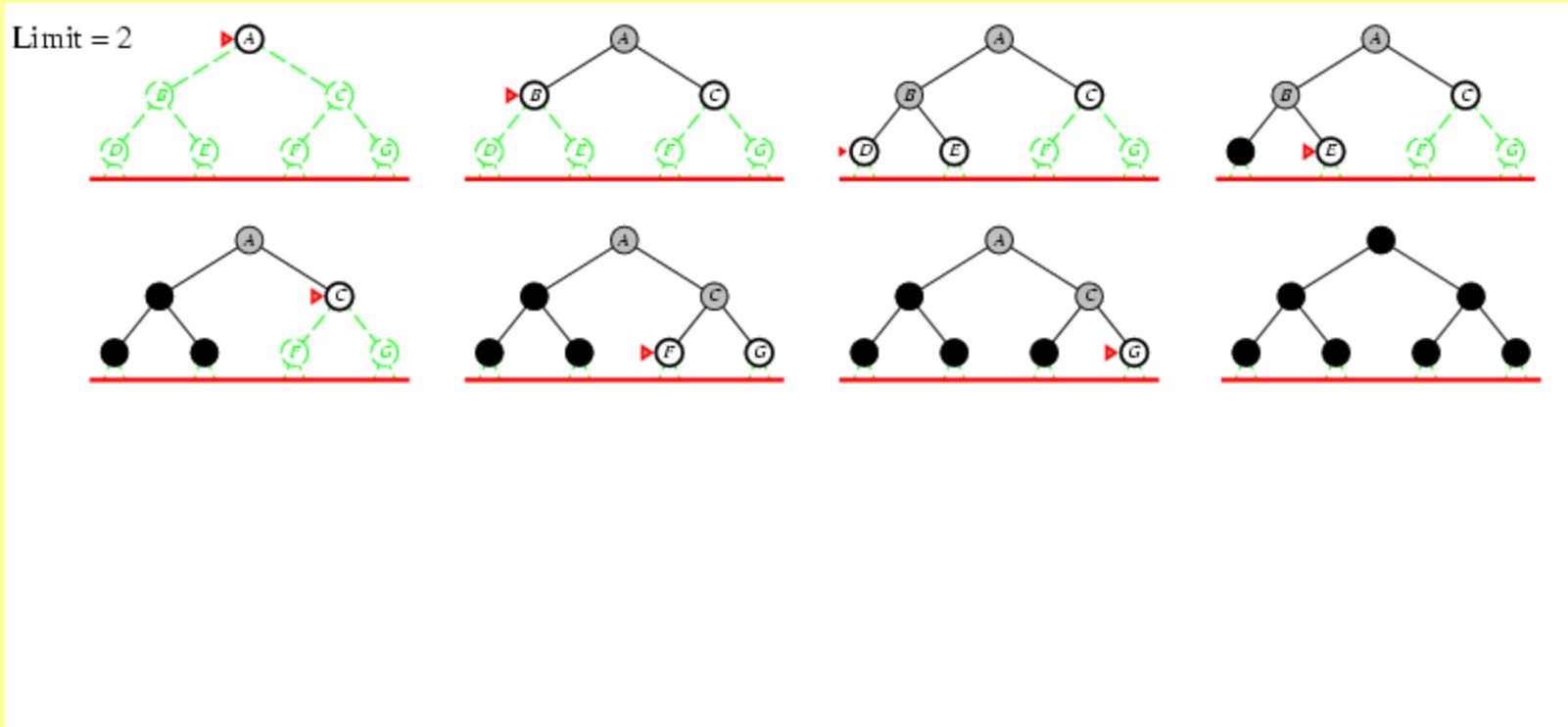
Limit = 0



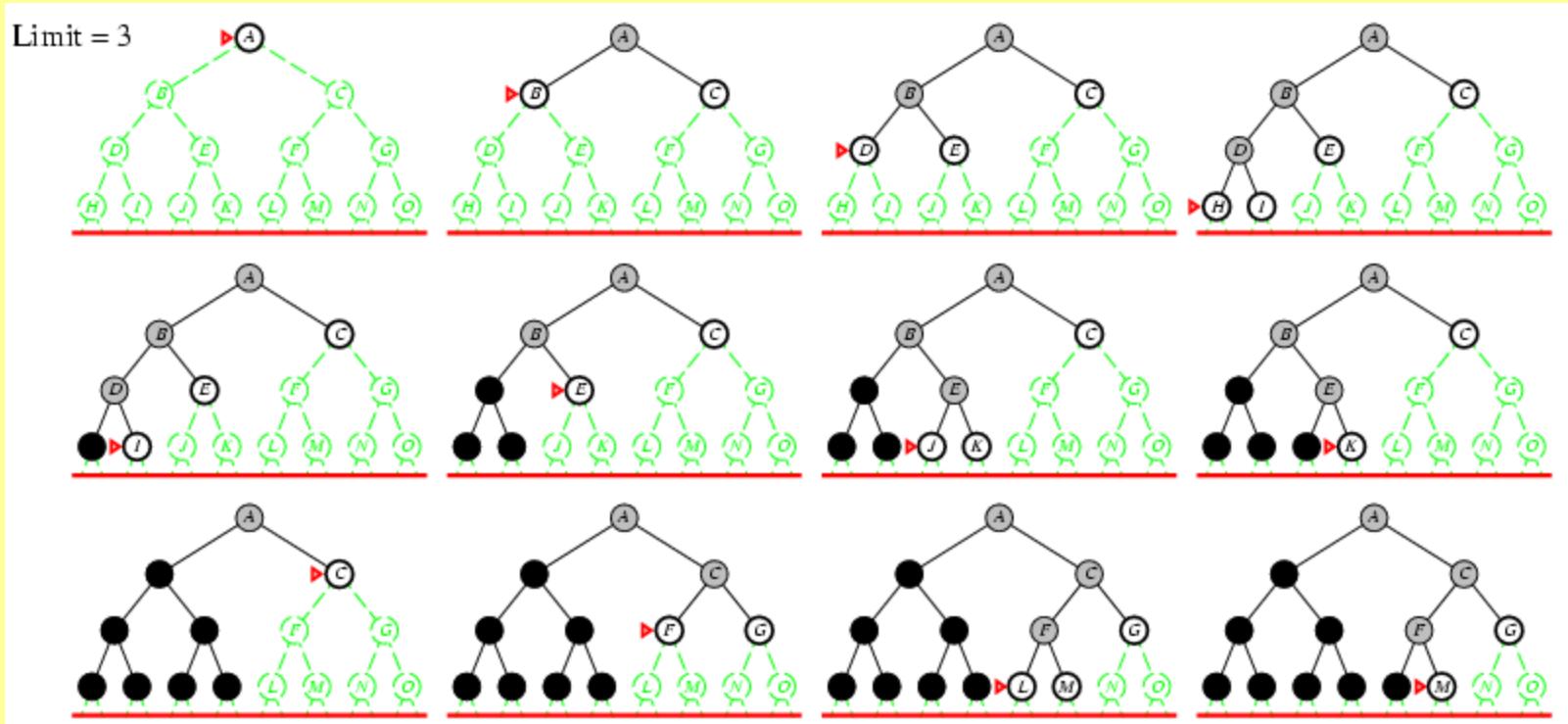
Iterative deepening search $L=1$

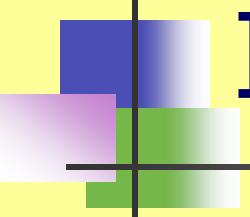


Iterative deepening search $L=2$



Iterative deepening search $L=3$





Iterative deepening search

- Number of nodes generated in a depth-limited search to depth d with branching factor b :

$$N_{DLS} = b^0 + b^1 + b^2 + \dots + b^{d-2} + b^{d-1} + b^d$$

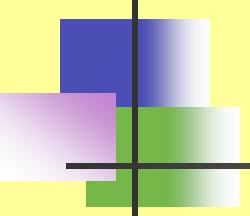
- Number of nodes generated in an iterative deepening search to depth d with branching factor b :

$$N_{IDS} = (d+1)b^0 + d b^1 + (d-1)b^2 + \dots + 3b^{d-2} + 2b^{d-1} + 1b^d = O(b^d) \neq O(b^{d+1})$$

- For $b = 10, d = 5$,

- $N_{DLS} = 1 + 10 + 100 + 1,000 + 10,000 + 100,000 = 111,111$
- $N_{IDS} = 6 + 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450$
- $N_{BFS} = \dots = 1,111,100$

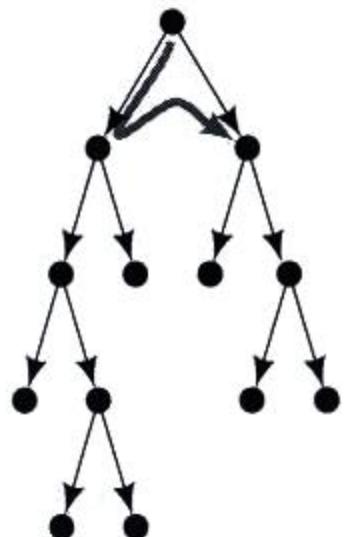
BFS



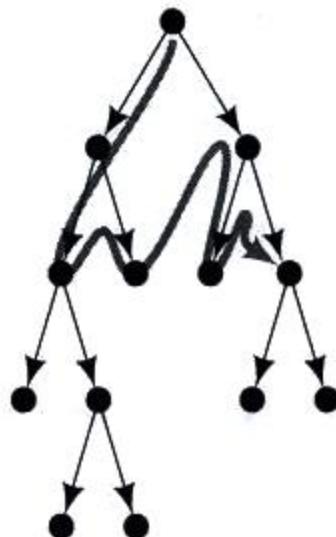
Properties of iterative deepening search

- Complete? Yes
- Time? $(d+1)b^0 + d b^1 + (d-1)b^2 + \dots + b^d = O(b^d)$
- Space? $O(bd)$
- Optimal? Yes, if step cost = 1 or increasing function of depth.

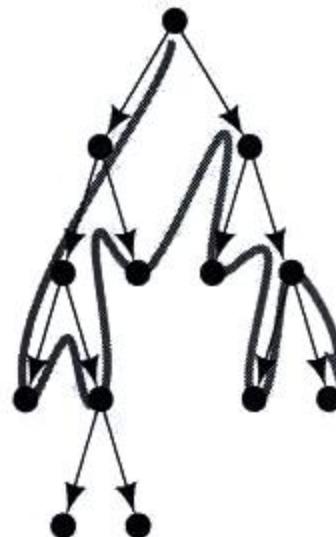
Example IDS



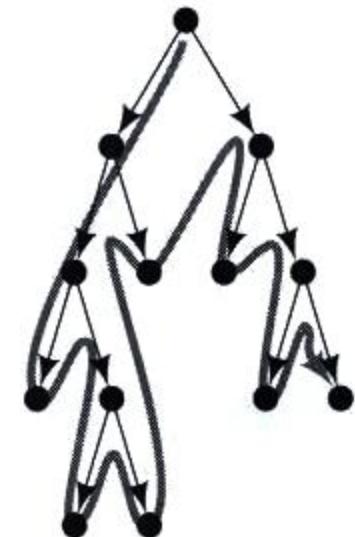
Depth bound = 1



Depth bound = 2

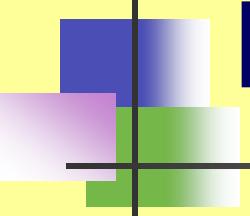


Depth bound = 3



Depth bound = 4

Stages in Iterative-Deepening Search



Bidirectional Search

- Idea
 - simultaneously search forward from S and backwards from G
 - stop when both “meet in the middle”
 - need to keep track of the intersection of 2 open sets of nodes
- What does searching backwards from G mean
 - need a way to specify the predecessors of G
 - this can be difficult,
 - e.g., predecessors of checkmate in chess?
 - what if there are multiple goal states?
 - what if there is only a goal test, no explicit list?

Bi-Directional Search

Complexity: time and space complexity are: $O(b^{d/2})$

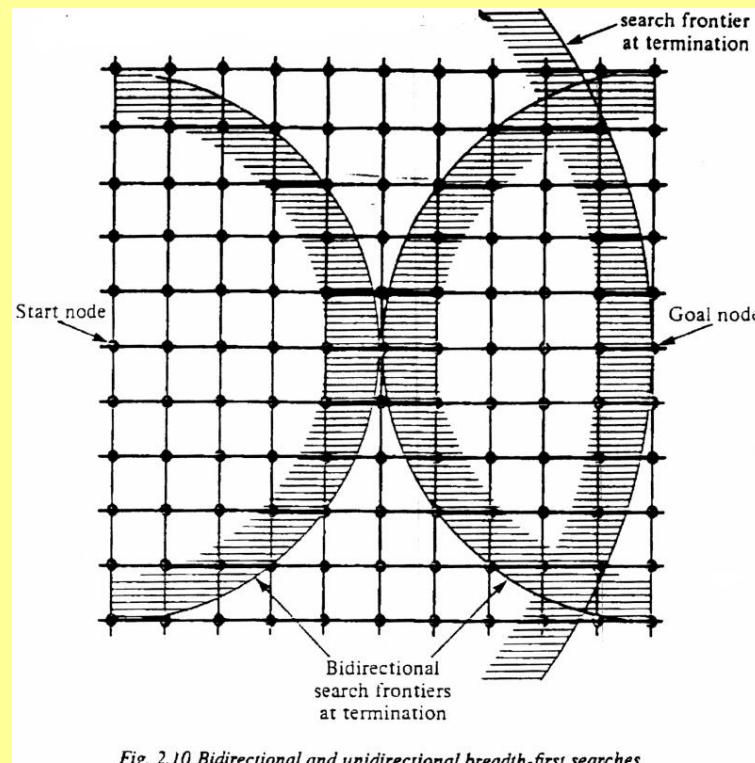
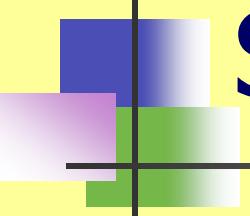


Fig. 2.10 Bidirectional and unidirectional breadth-first searches.

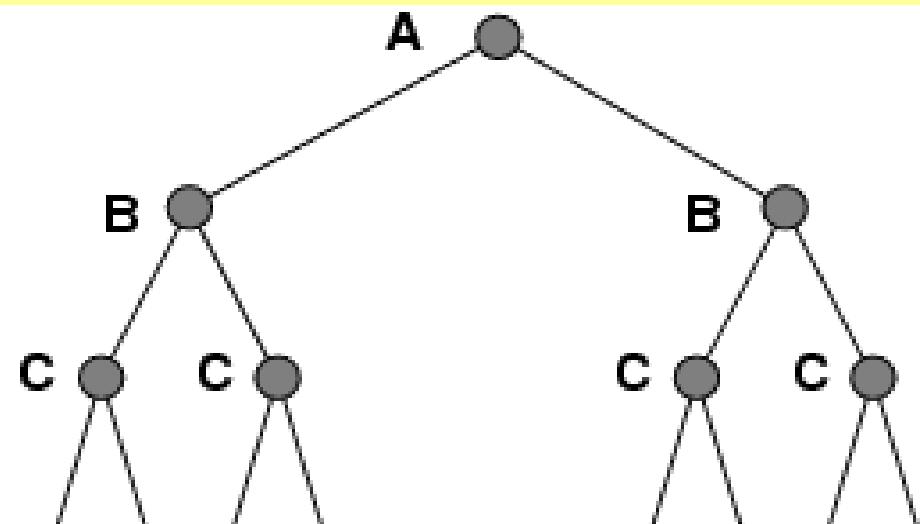
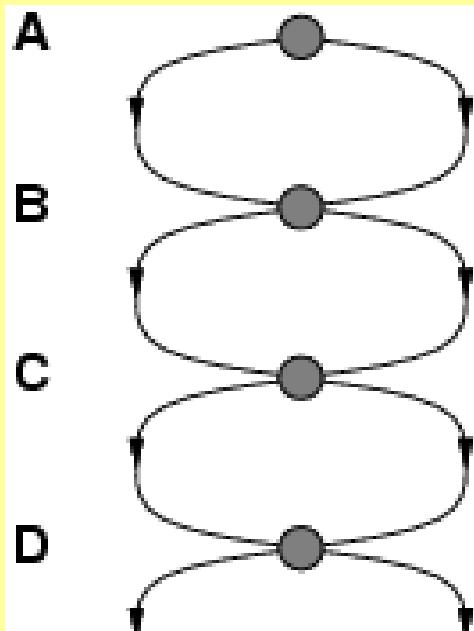


Summary of algorithms

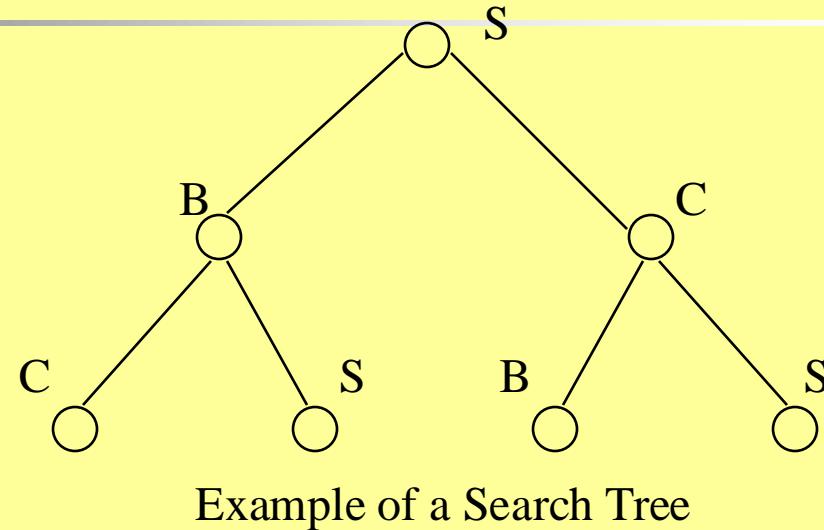
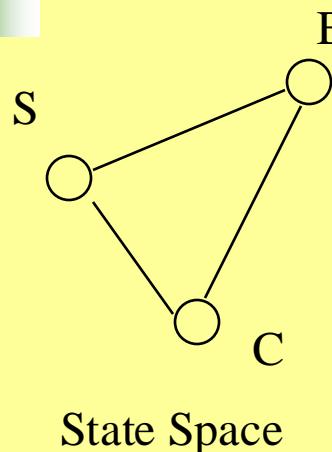
Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes	Yes	No	No	Yes
Time	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(b^m)$	$O(b^l)$	$O(b^d)$
Space	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(bm)$	$O(bl)$	$O(bd)$
Optimal?	Yes	Yes	No	No	Yes

Repeated states

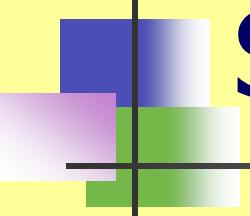
- Failure to detect repeated states can turn a linear problem into an exponential one!



Solutions to Repeated States



- Method 1 ← suboptimal but practical
 - do not create paths containing cycles (loops)
- Method 2 ← optimal but memory inefficient
 - never generate a state generated before
 - must keep track of all possible states (uses a lot of memory)
 - e.g., 8-puzzle problem, we have $9! = 362,880$ states



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

<http://www.cs.rmit.edu.au/AI-Search/Product/>

<http://aima.cs.berkeley.edu/demos.html> (for more demos)

Chapter 4

Informed/Heuristic Search

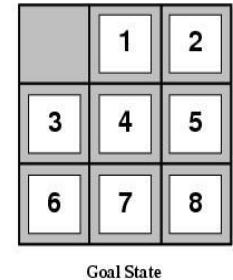
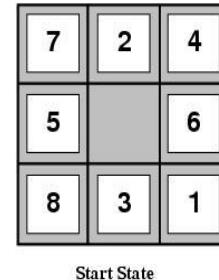
Outline

- Limitations of uninformed search methods
- Informed (or heuristic) search uses problem-specific heuristics to improve efficiency
 - Best-first
 - A*
 - RBFS
 - SMA*
 - Techniques for generating heuristics
- Can provide significant speed-ups in practice
 - e.g., on 8-puzzle
 - But can still have worst-case exponential time complexity
- Reading:
 - Chapter 4, Sections 4.1 and 4.2

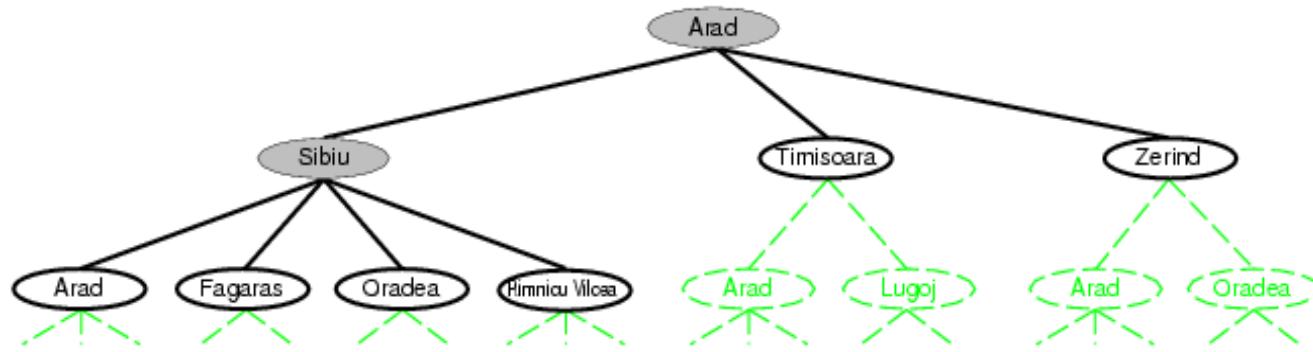
Limitations of uninformed search

- 8-puzzle
 - Avg. solution cost is about 22 steps
 - branching factor ~ 3
 - Exhaustive search to depth 22:
 - 3.1×10^{10} states
 - E.g., $d=12$, IDS expands 3.6 million states on average

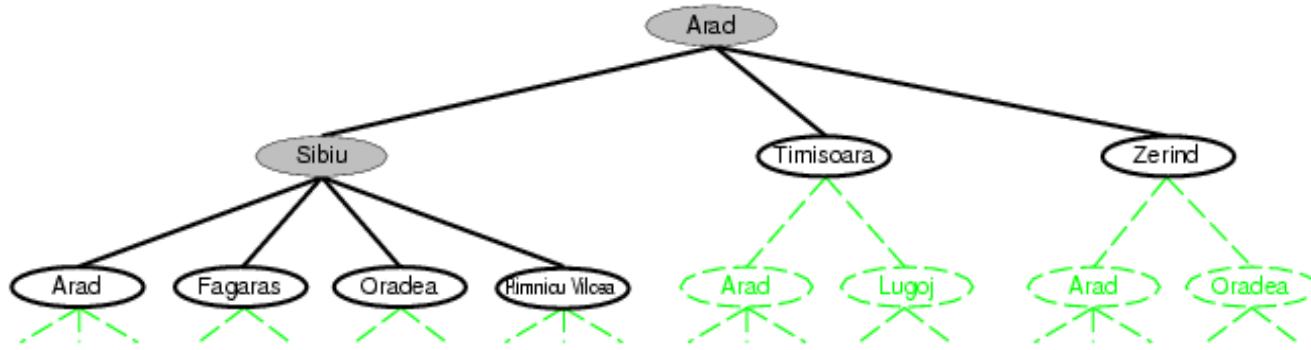
[24 puzzle has 10^{24} states (much worse)]



Recall tree search...



Recall tree search...



```
function TREE-SEARCH(problem, strategy) returns a solution
  initialize the search tree using the initial state of problem
  loop do
    if there are no candidates for expansion then return failure
    choose a leaf node for expansion according to strategy
    if the node contains a goal state then return the corresponding solution
    else expand the node and add the resulting nodes to the search tree
```

This “strategy” is what differentiates different search algorithms

Best-first search

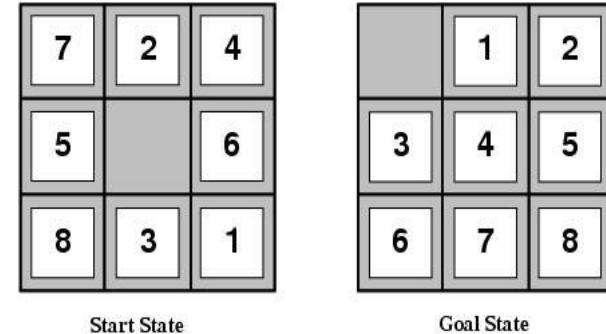
- Idea: use an **evaluation function** $f(n)$ for each node
 - estimate of "desirability"
 - Expand most desirable unexpanded node
- Implementation:
 - Order the nodes in fringe by $f(n)$ (by desirability, lowest $f(n)$ first)
- Special cases:
 - uniform cost search (from last lecture): $f(n) = g(n) = \text{path to } n$
 - greedy best-first search
 - A* search
- Note: evaluation function is an estimate of node quality
 - => More accurate name for "best first" search would be "seemingly best-first search"

Heuristic function

- Heuristic:
 - Definition: “using rules of thumb to find answers”
- Heuristic function $h(n)$
 - Estimate of (optimal) cost from n to goal
 - $h(n) = 0$ if n is a goal node
 - Example: straight line distance from n to Bucharest
 - Note that this is not the true state-space distance
 - It is an estimate – actual state-space distance can be higher
 - Provides problem-specific knowledge to the search algorithm

Heuristic functions for 8-puzzle

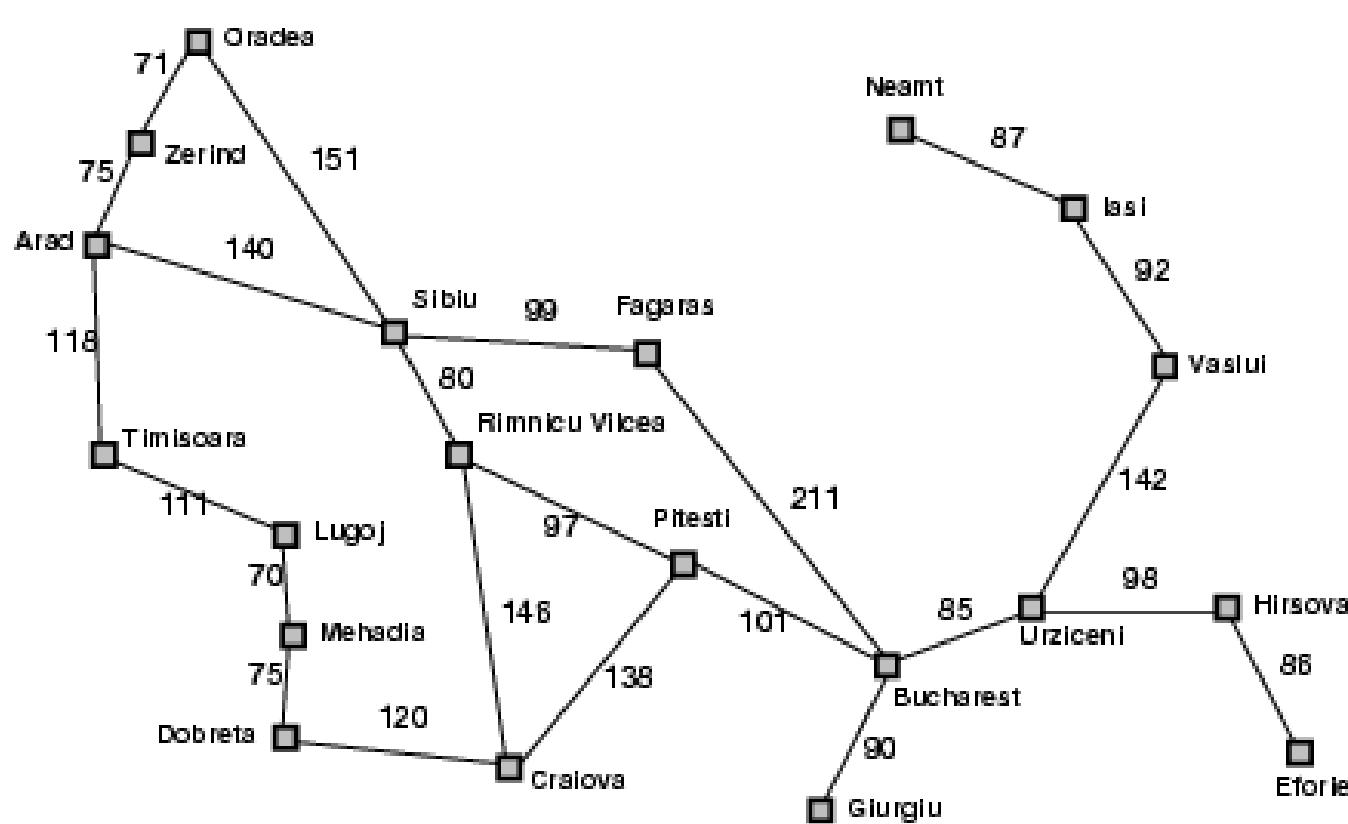
- 8-puzzle
 - Avg. solution cost is about 22 steps
 - branching factor ~ 3
 - Exhaustive search to depth 22:
 - 3.1×10^{10} states.
 - A good heuristic function can reduce the search process.
- Two commonly used heuristics
 - h_1 = the number of misplaced tiles
 - $h_1(s)=8$
 - h_2 = the sum of the distances of the tiles from their goal positions (Manhattan distance).
 - $h_2(s)=3+1+2+2+2+3+3+2=18$



Greedy best-first search

- Special case of best-first search
 - Uses $h(n)$ = heuristic function as its evaluation function
 - Expand the node that appears closest to goal

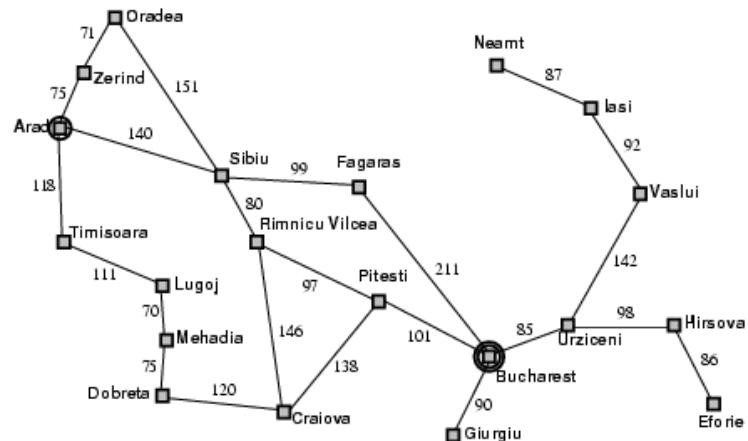
Romania with step costs in km



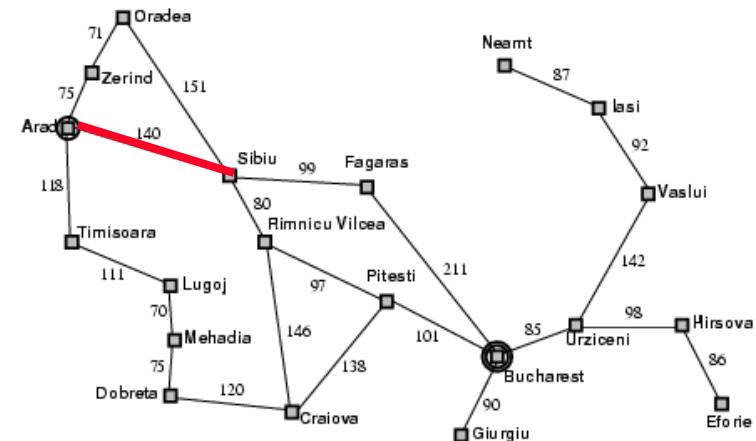
Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Greedy best-first search example

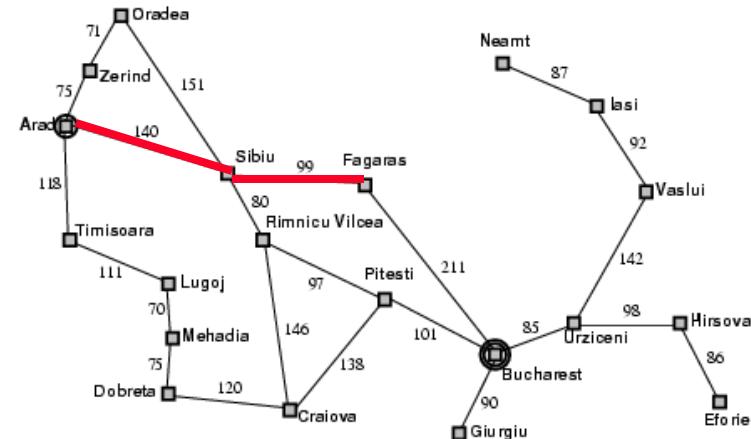
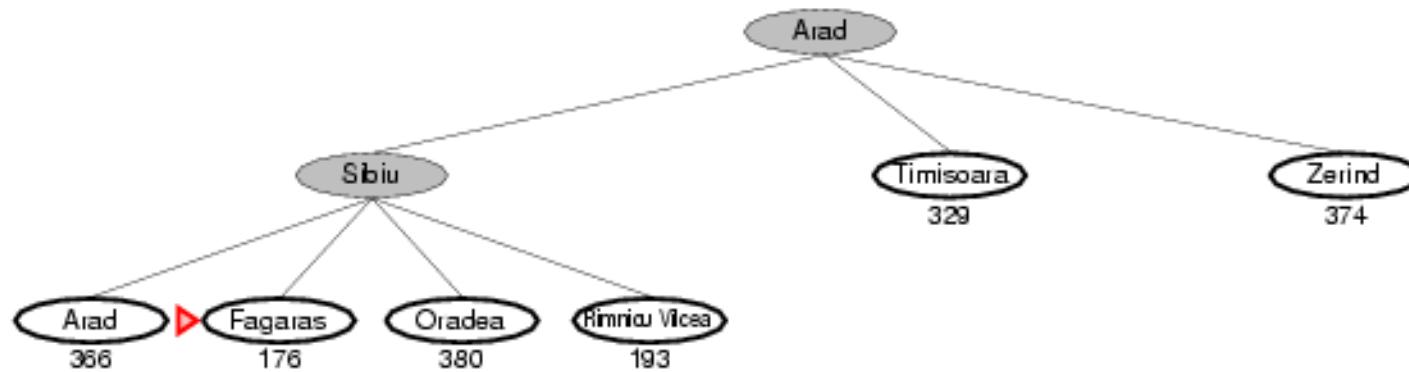
► Arad
366



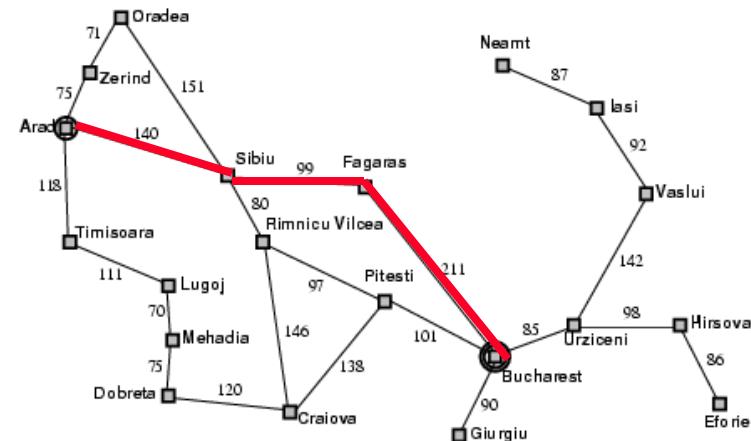
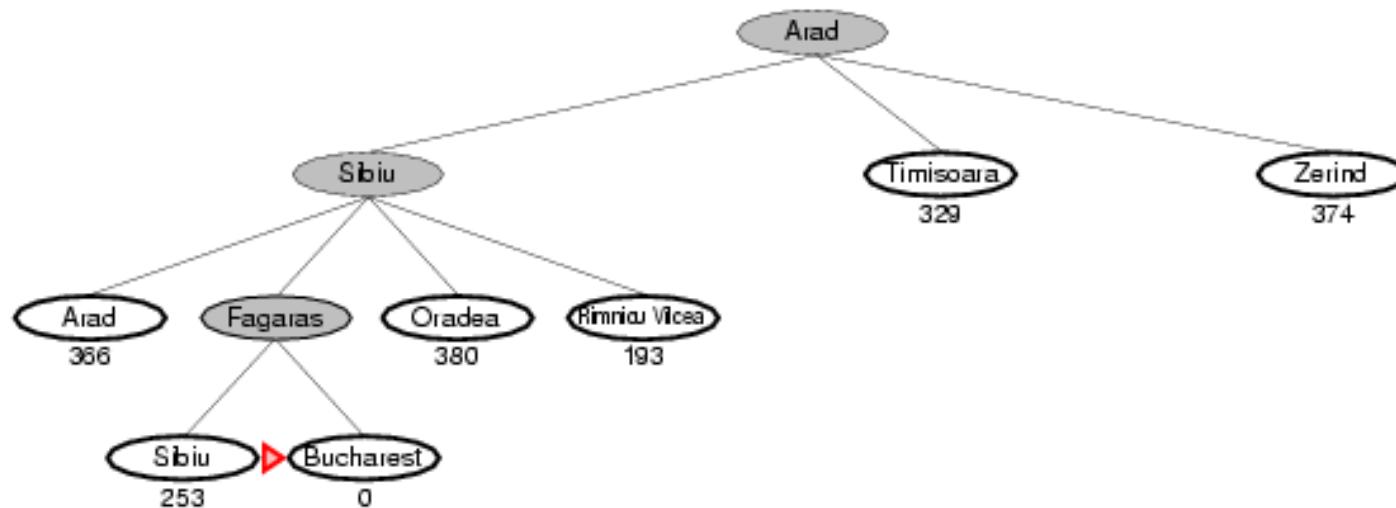
Greedy best-first search example



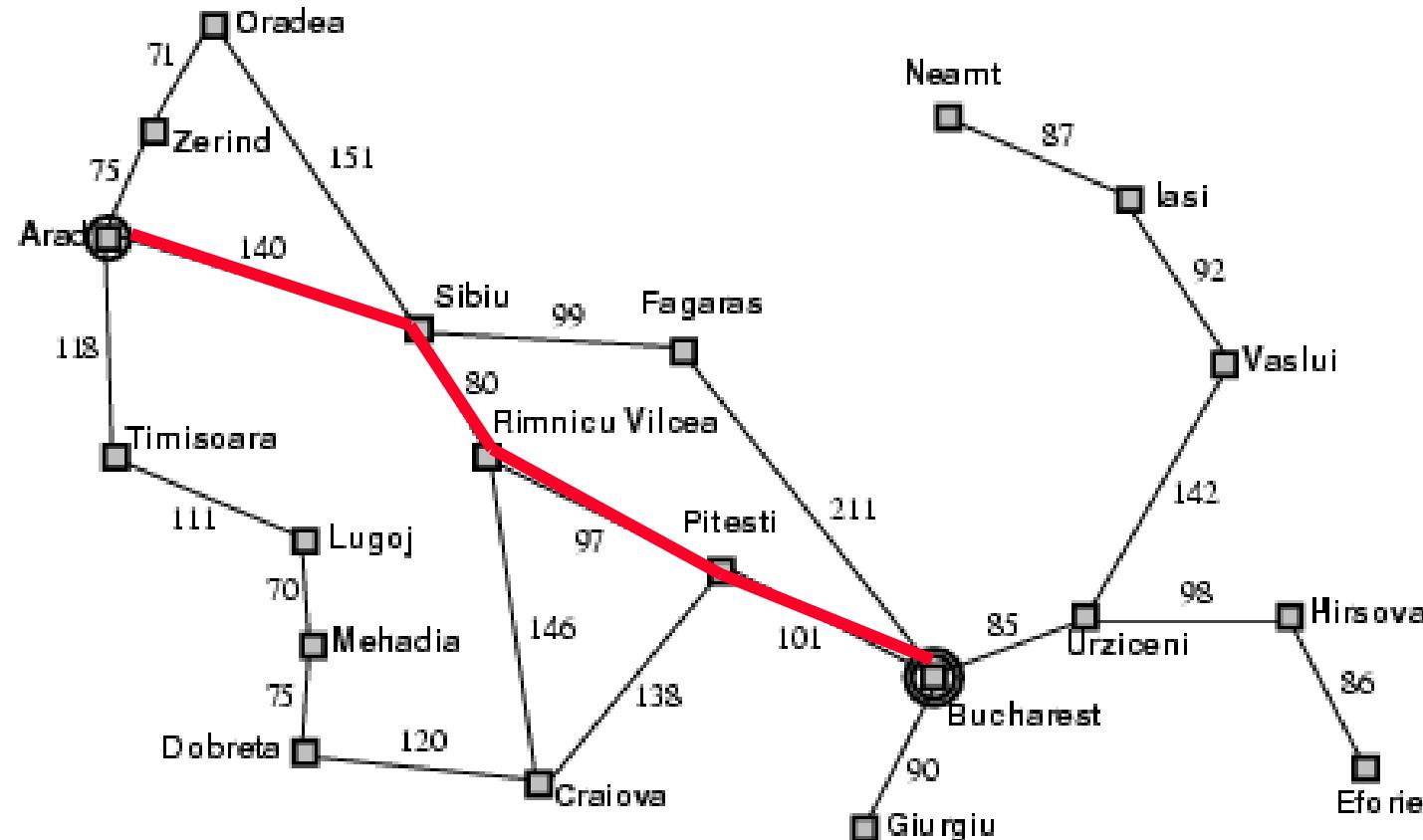
Greedy best-first search example



Greedy best-first search example



Optimal Path



Properties of greedy best-first search

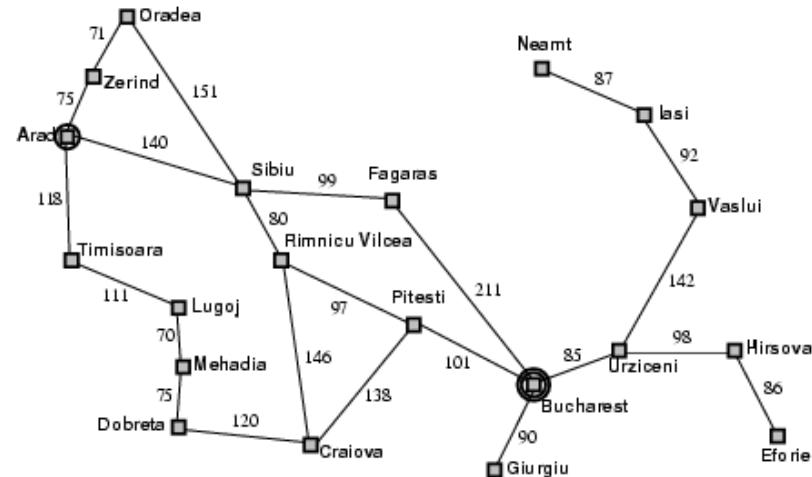
- Complete?
 - Not unless it keeps track of all states visited
 - Otherwise can get stuck in loops (just like DFS)
- Optimal?
 - No – we just saw a counter-example
- Time?
 - $O(b^m)$, can generate all nodes at depth m before finding solution
 - m = maximum depth of search space
- Space?
 - $O(b^m)$ – again, worst case, can generate all nodes at depth m before finding solution

A* Search

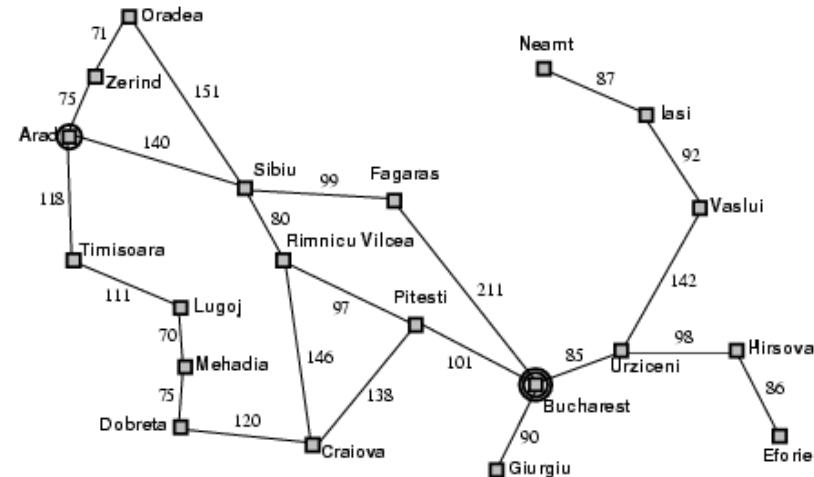
- Expand node based on estimate of total path cost through node
- Evaluation function $f(n) = g(n) + h(n)$
 - $g(n)$ = cost so far to reach n
 - $h(n)$ = estimated cost from n to goal
 - $f(n)$ = estimated total cost of path through n to goal
- Efficiency of search will depend on quality of heuristic $h(n)$

A* search example

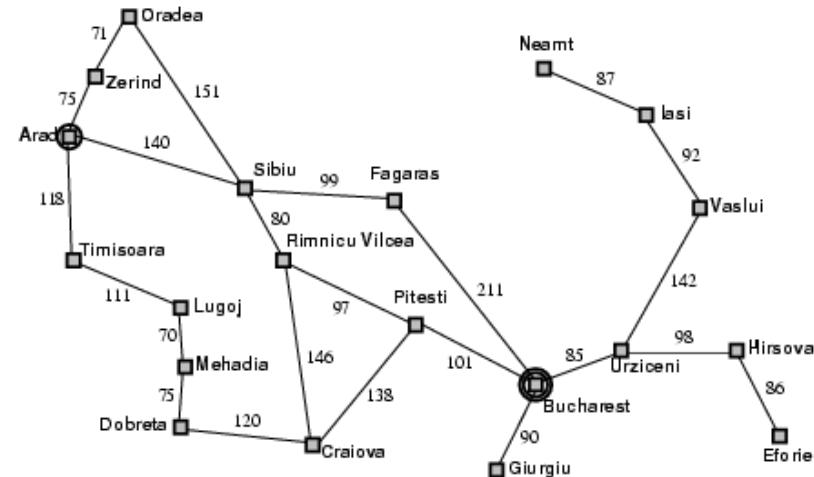
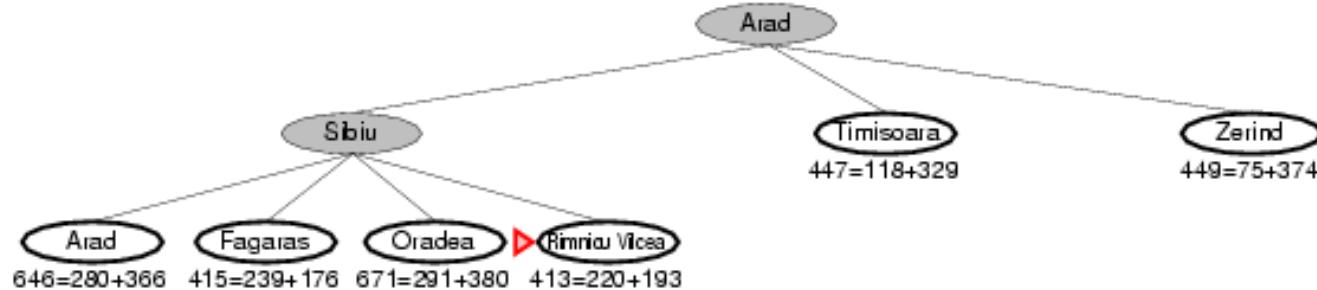
► Arad
366=0+366



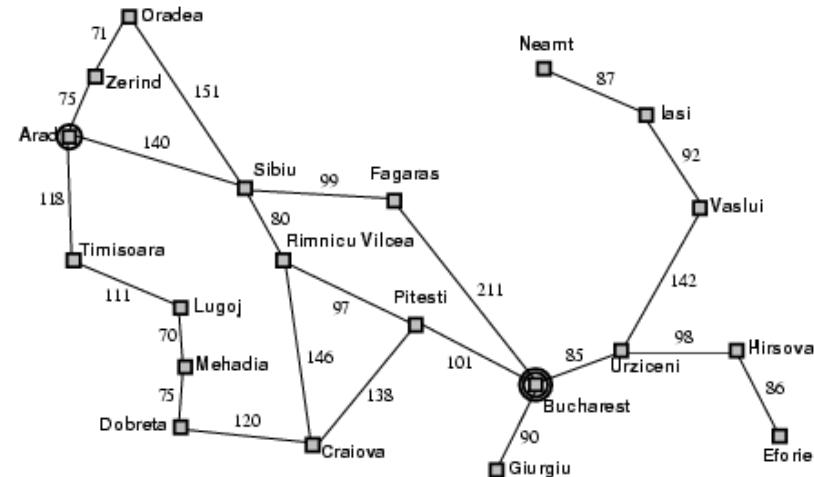
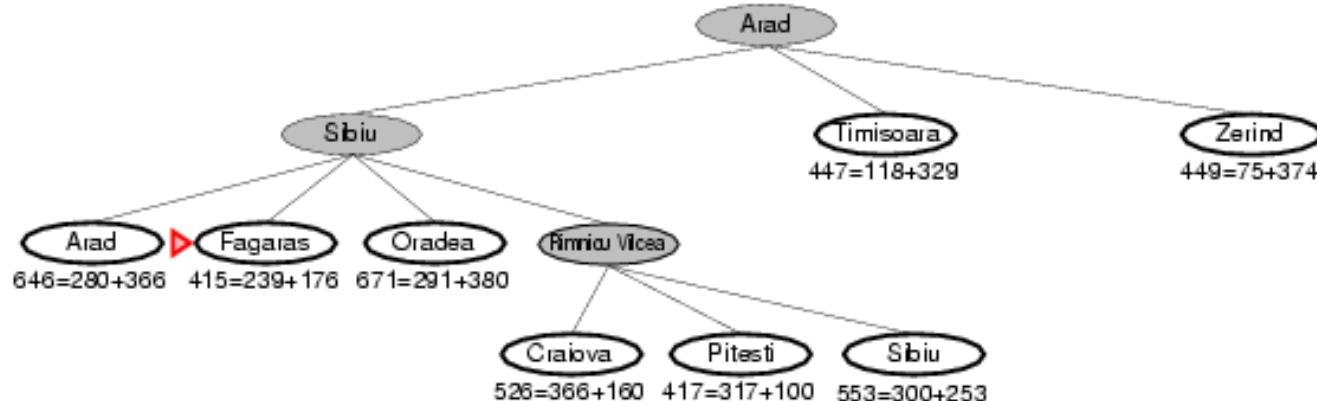
A* search example



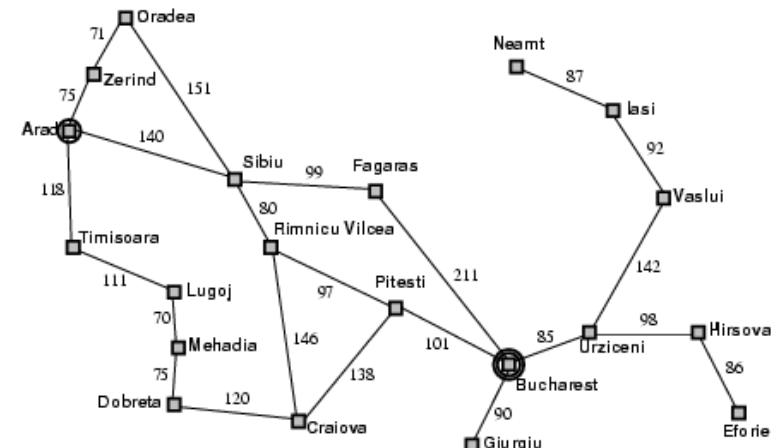
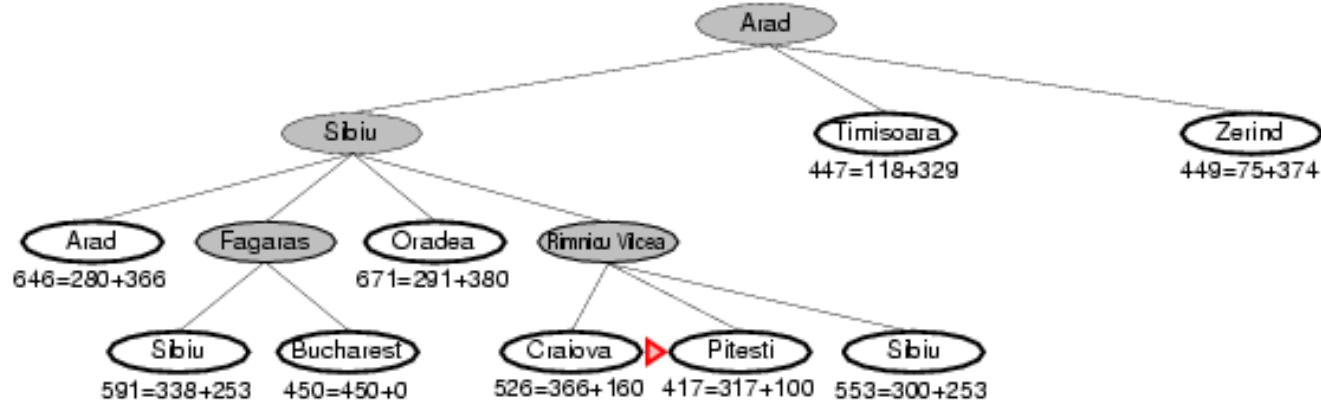
A* search example



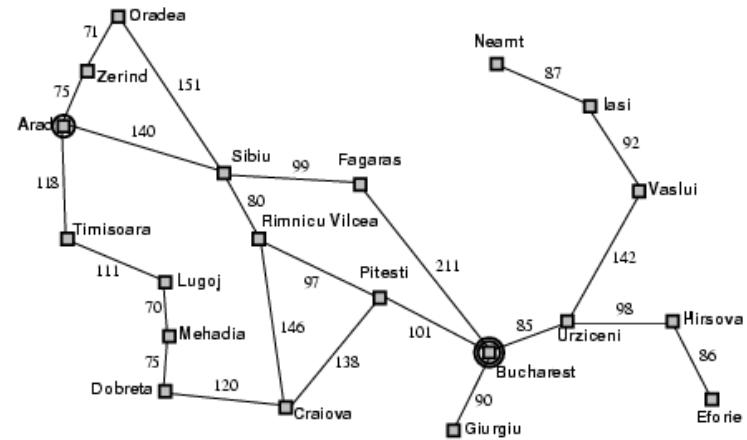
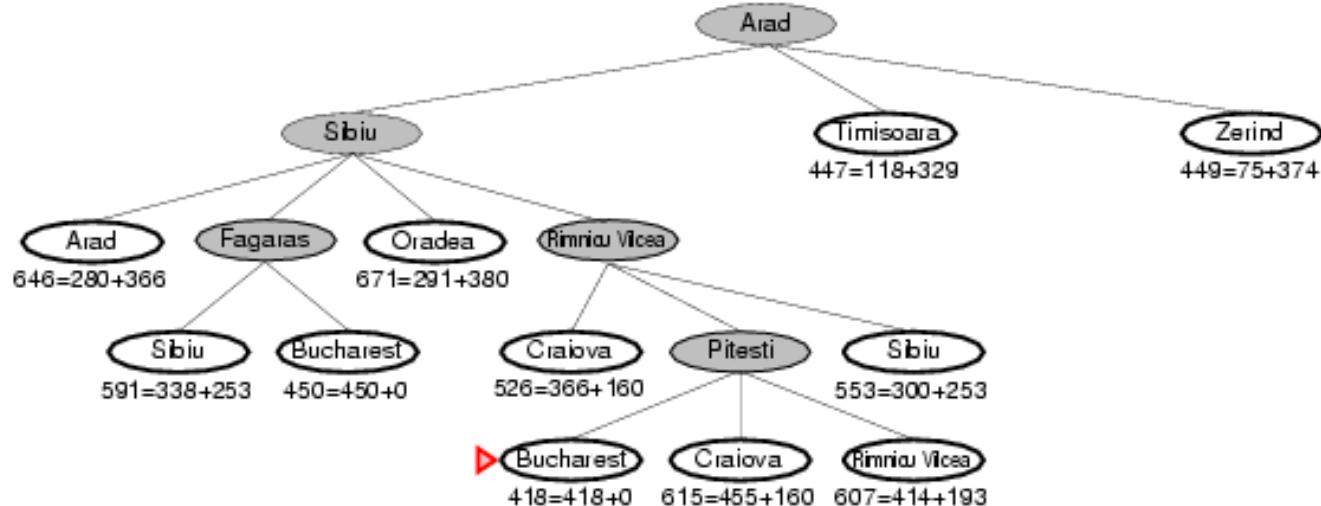
A* search example



A* search example



A* search example

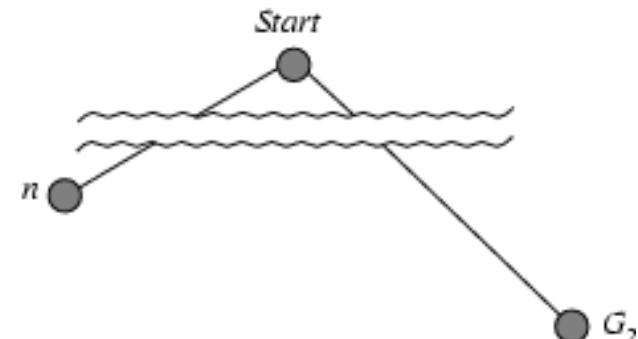


Admissible heuristics

- A heuristic $h(n)$ is **admissible** if for every node n , $h(n) \leq h^*(n)$, where $h^*(n)$ is the **true** cost to reach the goal state from n .
- An admissible heuristic **never overestimates** the cost to reach the goal, i.e., it is **optimistic**
- Example: $h_{SLD}(n)$ is **admissible**
 - never overestimates the actual road distance
- **Theorem:**
If $h(n)$ is admissible, A* using TREE-SEARCH is optimal

Optimality of A* (proof)

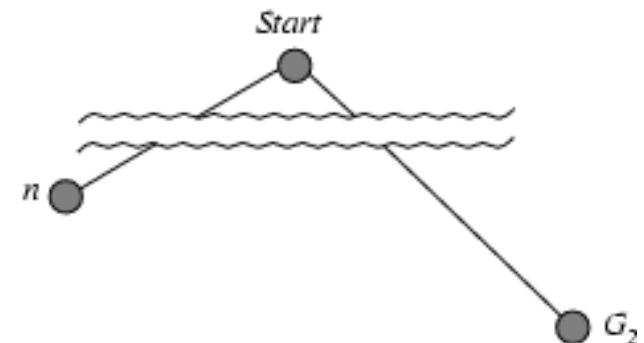
- Suppose some suboptimal goal G_2 has been generated and is in the fringe. Let n be an unexpanded node in the fringe such that n is on a shortest path to an optimal goal G .



- $f(G_2) = g(G_2)$ since $h(G_2) = 0$
- $g(G_2) > g(G)$ since G_2 is suboptimal
- $f(G) = g(G)$ since $h(G) = 0$
- $f(G_2) > f(G)$ from above

Optimality of A* (proof)

- Suppose some suboptimal goal G_2 has been generated and is in the fringe. Let n be an unexpanded node in the fringe such that n is on a shortest path to an optimal goal G .



- $f(G_2) > f(G)$ from above
- $h(n) \leq h^*(n)$ since h is admissible
- $g(n) + h(n) \leq g(n) + h^*(n)$
- $f(n) \leq f(G)$

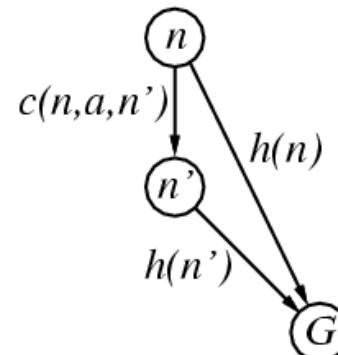
Hence $f(G_2) > f(n)$, and A* will never select G_2 for expansion

Optimality for graphs?

- Admissibility is not sufficient for graph search
 - In graph search, the optimal path to a repeated state could be discarded if it is not the first one generated
 - Can fix problem by requiring consistency property for $h(n)$
- A heuristic is **consistent** if for every successor n' of a node n generated by any action a ,

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

(aka “*monotonic*”)



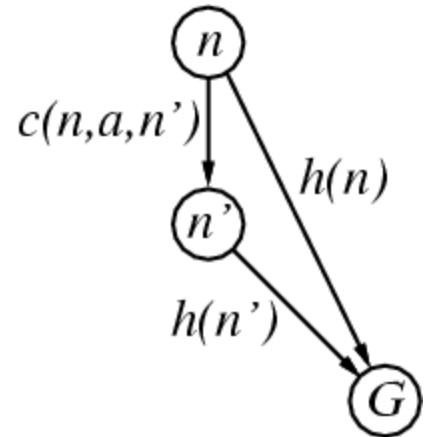
- admissible heuristics are generally consistent

A* is optimal with consistent heuristics

- If h is consistent, we have

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

i.e., $f(n)$ is non-decreasing along any path.

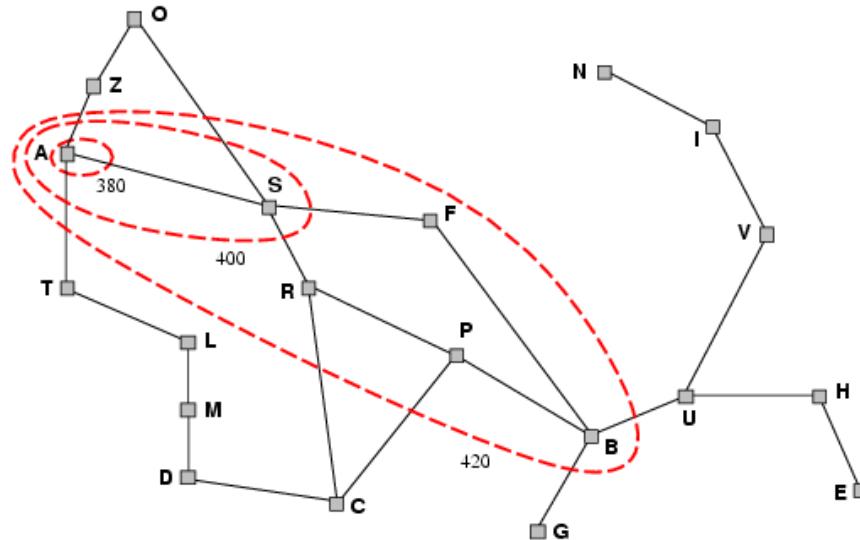


Thus, first goal-state selected for expansion must be optimal

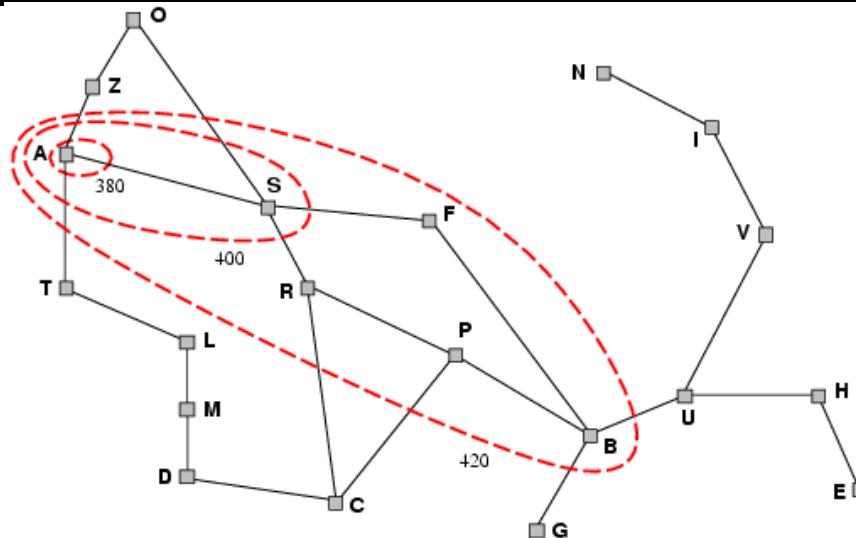
- Theorem:
 - If $h(n)$ is consistent, A* using GRAPH-SEARCH is optimal
 -

Contours of A* Search

- A* expands nodes in order of increasing f value
- Gradually adds "f-contours" of nodes
- Contour i has all nodes with $f=f_i$, where $f_i < f_{i+1}$



Contours of A* Search



- With uniform-cost ($h(n) = 0$), contours will be circular
- With good heuristics, contours will be focused around optimal path
- A* will expand all nodes with cost $f(n) < C^*$

Properties of A*

- Complete?
 - Yes (unless there are infinitely many nodes with $f \leq f(G)$)
- Optimal?
 - Yes
 - Also optimally efficient:
 - No other optimal algorithm will expand fewer nodes, for a given heuristic
- Time?
 - Exponential in worst case
- Space?
 - Exponential in worst case

Comments on A*

- A* expands all nodes with $f(n) < C^*$
 - This can still be exponentially large
- Exponential growth will occur unless error in $h(n)$ grows no faster than $\log(\text{true path cost})$
 - In practice, error is usually proportional to true path cost (not log)
 - So exponential growth is common

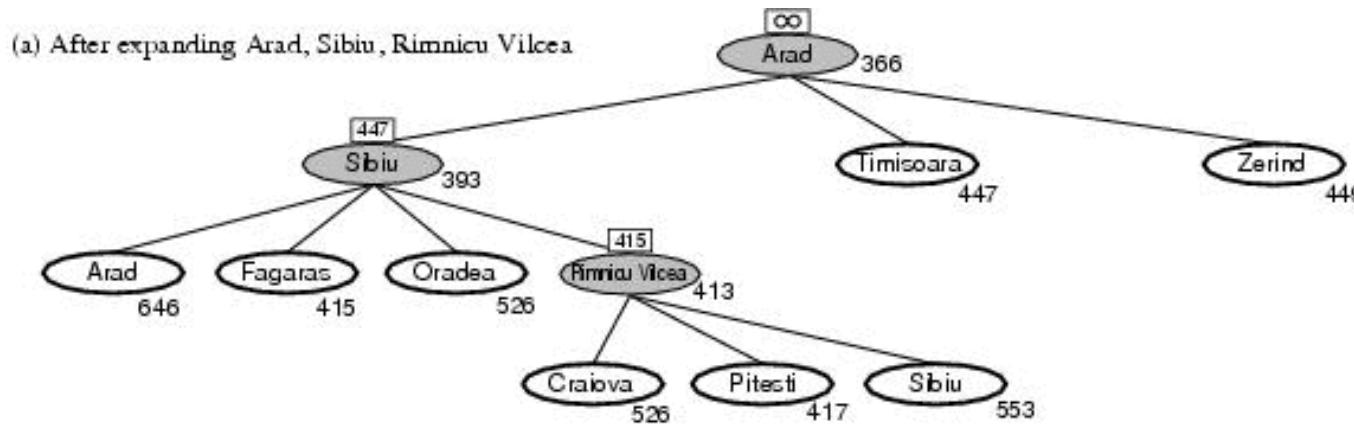
Memory-bounded heuristic search

- In practice A* runs out of memory before it runs out of time
 - How can we solve the memory problem for A* search?
- Idea: Try something like depth first search, but let's not forget everything about the branches we have partially explored.

Recursive Best-First Search (RBFS)

- Similar to DFS, but keeps track of the f-value of the best alternative path available from any ancestor of the current node
- If current node exceeds f-limit \rightarrow backtrack to alternative path
- As it backtracks, replace f-value of each node along the path with the best $f(n)$ value of its children
 - This allows it to return to this subtree, if it turns out to look better than alternatives

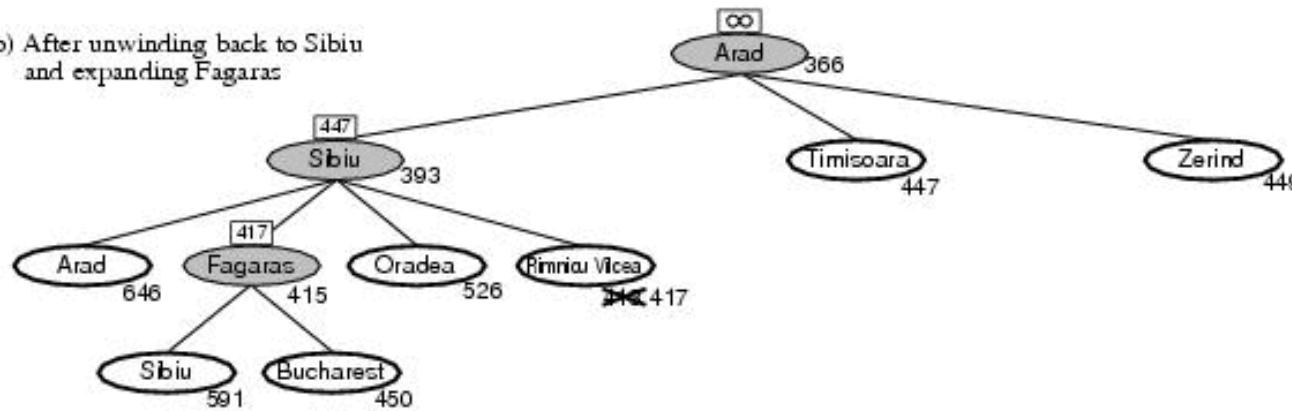
Recursive Best First Search: Example



- Path until Rumnicu Vilcea is already expanded
- Above node; f -limit for every recursive call is shown on top.
- Below node: $f(n)$
- The path is followed until Pitesti which has a f -value worse than the f -limit.

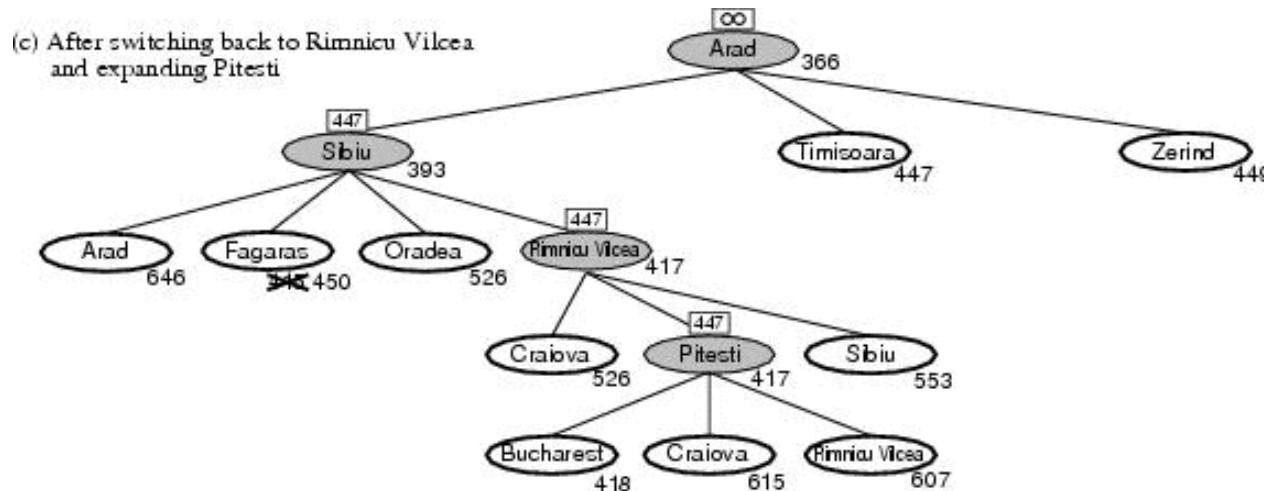
RBFS example

(b) After unwinding back to Sibiu and expanding Fagaras



- Unwind recursion and store best f -value for current best leaf Pitesti
$$\text{result, } f[\text{best}] \leftarrow \text{RBFS}(\text{problem, best, min}(f_limit, \text{alternative}))$$
- best is now Fagaras. Call RBFS for new best
 - best value is now 450

RBFS example



- Unwind recursion and store best f -value for current best leaf Fagaras
$$result, f[best] \leftarrow \text{RBFS}(\text{problem}, best, \min(f_limit, \text{alternative}))$$
- $best$ is now Rimnicu Vilcea (again). Call RBFS for new $best$
 - Subtree is again expanded.
 - Best $alternative$ subtree is now through Timisoara.
- Solution is found since because $447 > 418$.

RBFS properties

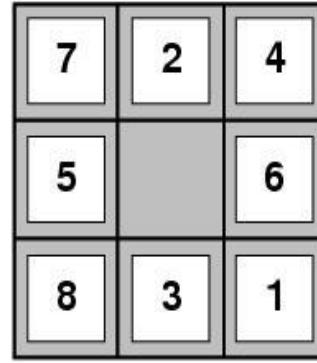
- Like A*, optimal if $h(n)$ is admissible
- Time complexity difficult to characterize
 - Depends on accuracy of $h(n)$ and how often best path changes.
 - Can end up “switching” back and forth
- Space complexity is $O(bd)$
 - Other extreme to A* - uses ***too little*** memory.

(Simplified) Memory-bounded A* (SMA*)

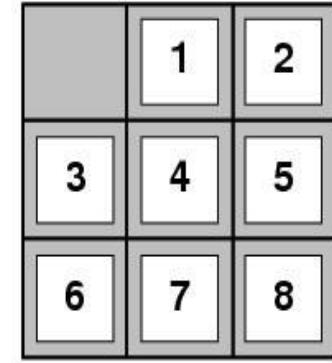
- This is like A*, but when memory is full we delete the worst node (largest f-value).
- Like RBFS, we remember the best descendant in the branch we delete.
- If there is a tie (equal f-values) we delete the oldest nodes first.
- simplified-MA* finds the optimal *reachable* solution given the memory constraint.
- Time can still be exponential.

Heuristic functions

- 8-puzzle
 - Avg. solution cost is about 22 steps
 - branching factor ~ 3
 - Exhaustive search to depth 22:
 - 3.1×10^{10} states.
 - A good heuristic function can reduce the search process.
- Two commonly used heuristics
 - h_1 = the number of misplaced tiles
 - $h_1(s)=8$
 - h_2 = the sum of the distances of the tiles from their goal positions (manhattan distance).
 - $h_2(s)=3+1+2+2+2+3+3+2=18$



Start State



Goal State

Notion of dominance

- If $h_2(n) \geq h_1(n)$ for all n (both admissible) then h_2 **dominates** h_1
 h_2 is better for search
- Typical search costs (average number of nodes expanded) for 8-puzzle problem

$d=12$ IDS = 3,644,035 nodes
 $A^*(h_1)$ = 227 nodes
 $A^*(h_2)$ = 73 nodes

$d=24$ IDS = too many nodes
 $A^*(h_1)$ = 39,135 nodes
 $A^*(h_2)$ = 1,641 nodes

Effective branching factor

- Effective branching factor b^*
 - Is the branching factor that a uniform tree of depth d would have in order to contain $N+1$ nodes.

$$N + 1 = 1 + b^* + (b^*)^2 + \dots + (b^*)^d$$

- Measure is fairly constant for sufficiently hard problems.
 - Can thus provide a good guide to the heuristic's overall usefulness.

Effectiveness of different heuristics

d	Search Cost			Effective Branching Factor		
	IDS	$A^*(h_1)$	$A^*(h_2)$	IDS	$A^*(h_1)$	$A^*(h_2)$
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	3644035	227	73	2.78	1.42	1.24
14	—	539	113	—	1.44	1.23
16	—	1301	211	—	1.45	1.25
18	—	3056	363	—	1.46	1.26
20	—	7276	676	—	1.47	1.27
22	—	18094	1219	—	1.48	1.28
24	—	39135	1641	—	1.48	1.26

- Results averaged over random instances of the 8-puzzle

Inventing heuristics via “relaxed problems”

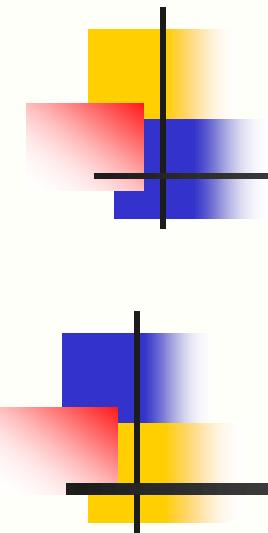
- A problem with fewer restrictions on the actions is called a **relaxed problem**
- The cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem
- If the rules of the 8-puzzle are relaxed so that a tile can move **anywhere**, then $h_1(n)$ gives the shortest solution
- If the rules are relaxed so that a tile can move to **any adjacent square**, then $h_2(n)$ gives the shortest solution
- Can be a useful way to generate heuristics
 - E.g., ABSOLVER (Prieditis, 1993) discovered the first useful heuristic for the Rubik’s cube puzzle

More on heuristics

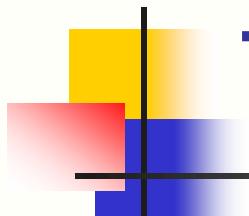
- $h(n) = \max\{ h_1(n), h_2(n), \dots, h_k(n) \}$
 - Assume all h functions are admissible
 - Always choose the least optimistic heuristic (most accurate) at each node
 - Could also learn a convex combination of features
 - Weighted sum of $h(n)$'s, where weights sum to 1
 - Weights learned via repeated puzzle-solving
- Could try to learn a heuristic function based on “features”
 - E.g., $x_1(n)$ = number of misplaced tiles
 - E.g., $x_2(n)$ = number of goal-adjacent-pairs that are currently adjacent
 - $h(n) = w_1 x_1(n) + w_2 x_2(n)$
 - Weights could be learned again via repeated puzzle-solving
 - Try to identify which features are predictive of path cost

Summary

- Uninformed search methods have their limits
- Informed (or heuristic) search uses problem-specific heuristics to improve efficiency
 - Best-first
 - A*
 - RBFS
 - SMA*
 - Techniques for generating heuristics
- Can provide significant speed-ups in practice
 - e.g., on 8-puzzle
 - But can still have worst-case exponential time complexity
- Next lecture: local search techniques
 - Hill-climbing, genetic algorithms, simulated annealing, etc



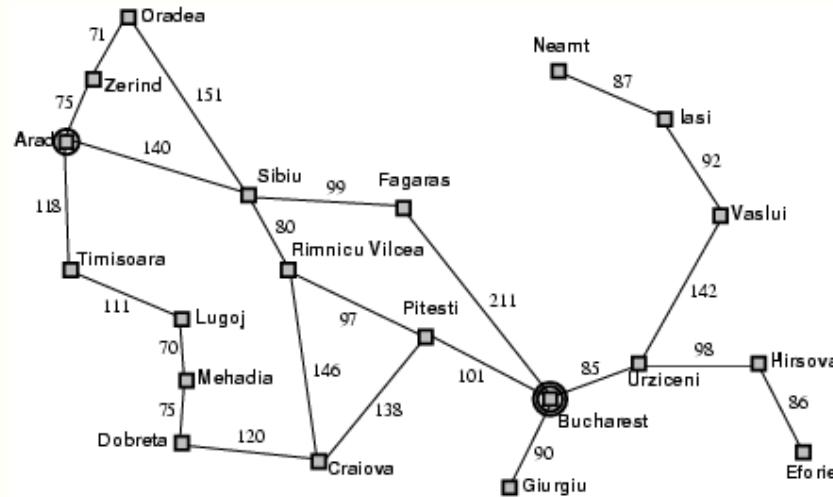
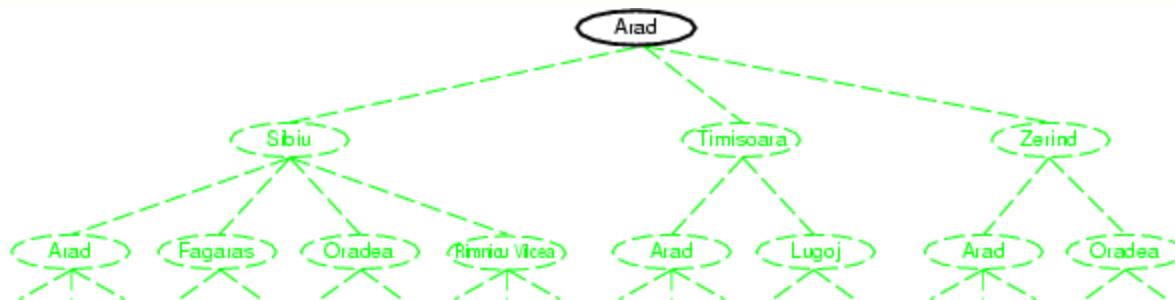
A* Search



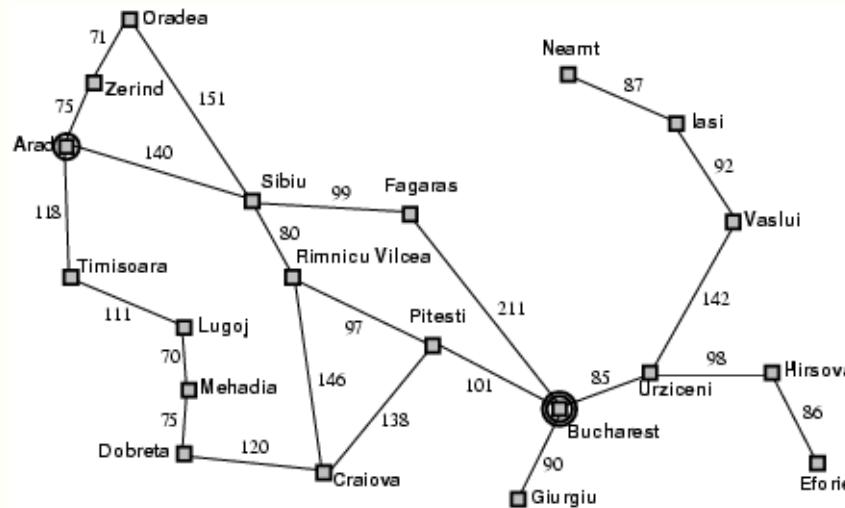
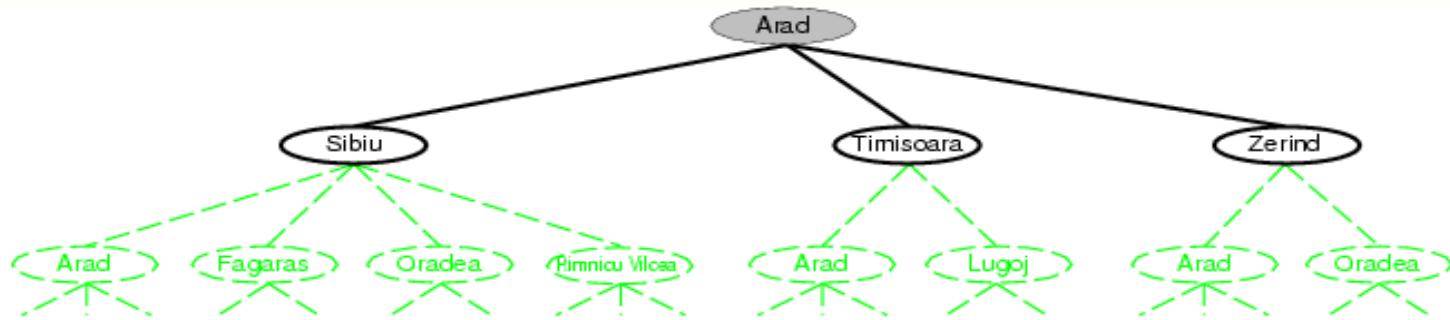
Tree search algorithms

- Basic idea:
 - Exploration of state space by generating successors of already-explored states (a.k.a.~**expanding** states).
 - Every states is evaluated: *is it a goal state?*

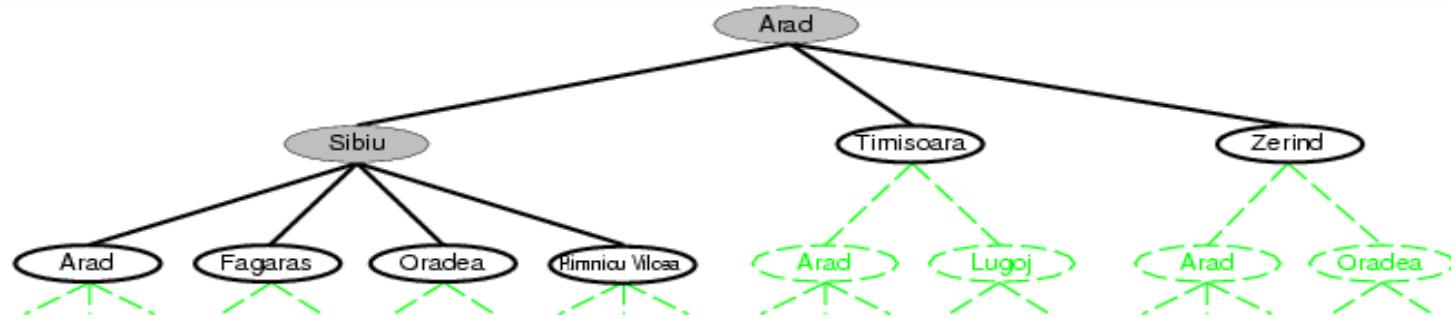
Tree search example



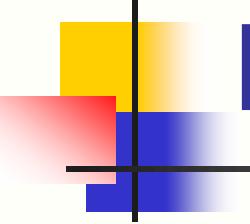
Tree search example



Tree search example



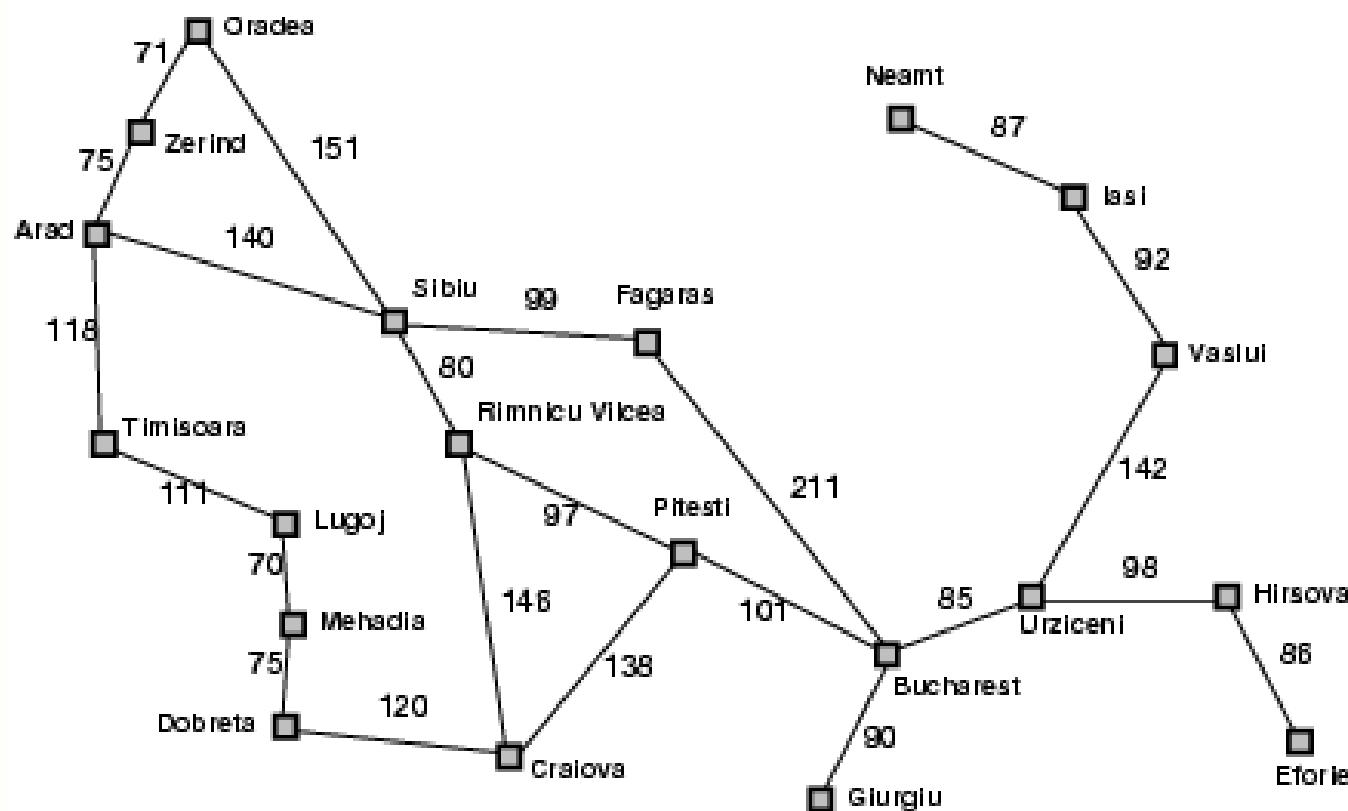
```
function TREE-SEARCH(problem, strategy) returns a solution, or failure
  initialize the search tree using the initial state of problem
  loop do
    if there are no candidates for expansion then return failure
    choose a leaf node for expansion according to strategy
    if the node contains a goal state then return the corresponding solution
    else expand the node and add the resulting nodes to the search tree
```

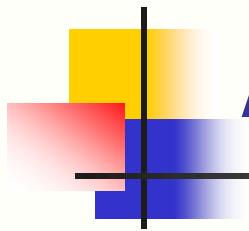


Best-first search

- Idea: use an **evaluation function** $f(n)$ for each node
 - $f(n)$ provides an estimate for the total cost.
 - Expand the node n with smallest $f(n)$.
- Implementation:
Order the nodes in fringe increasing order of cost.

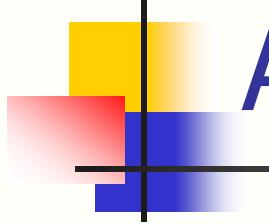
Romania with straight-line dist.





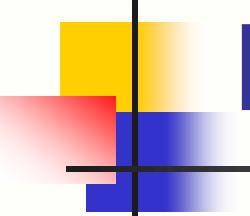
A* search

- Idea: avoid expanding paths that are already expensive
- Evaluation function $f(n) = g(n) + h(n)$
- $g(n)$ = cost so far to reach n
- $h(n)$ = estimated cost from n to goal
- $f(n)$ = estimated total cost of path through n to goal
- Best First search has $f(n)=h(n)$
- Uniform Cost search has $f(n)=g(n)$



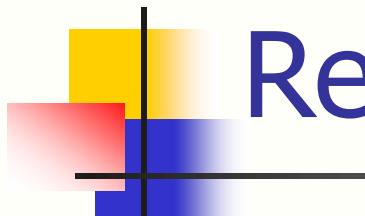
Admissible heuristics

- A heuristic $h(n)$ is **admissible** if for every node n , $h(n) \leq h^*(n)$, where $h^*(n)$ is the **true** cost to reach the goal state from n .
- An admissible heuristic **never overestimates** the cost to reach the goal, i.e., it is **optimistic**
- Example: $h_{SLD}(n)$ (never overestimates the actual road distance)
- **Theorem:** If $h(n)$ is admissible, A* using TREE-SEARCH is optimal



Dominance

- If $h_2(n) \geq h_1(n)$ for all n (both admissible)
- then h_2 **dominates** h_1
- h_2 is better for search: it is guaranteed to expand less or equal nr of nodes.
- Typical search costs (average number of nodes expanded):
- $d=12$ IDS = 3,644,035 nodes
 $A^*(h_1)$ = 227 nodes
 $A^*(h_2)$ = 73 nodes
- $d=24$ IDS = too many nodes
 $A^*(h_1)$ = 39,135 nodes
 $A^*(h_2)$ = 1,641 nodes



Relaxed problems

- A problem with fewer restrictions on the actions is called a **relaxed problem**
- The cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem
- 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

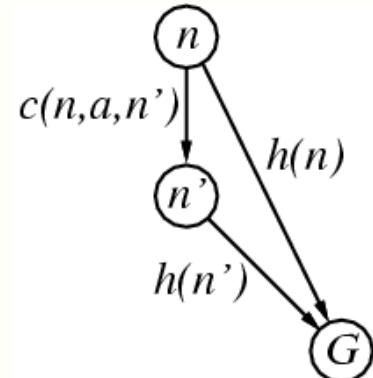
Consistent heuristics

- A heuristic is **consistent** if 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

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



It's the triangle inequality !

- i.e., $f(n)$ is non-decreasing along any path.

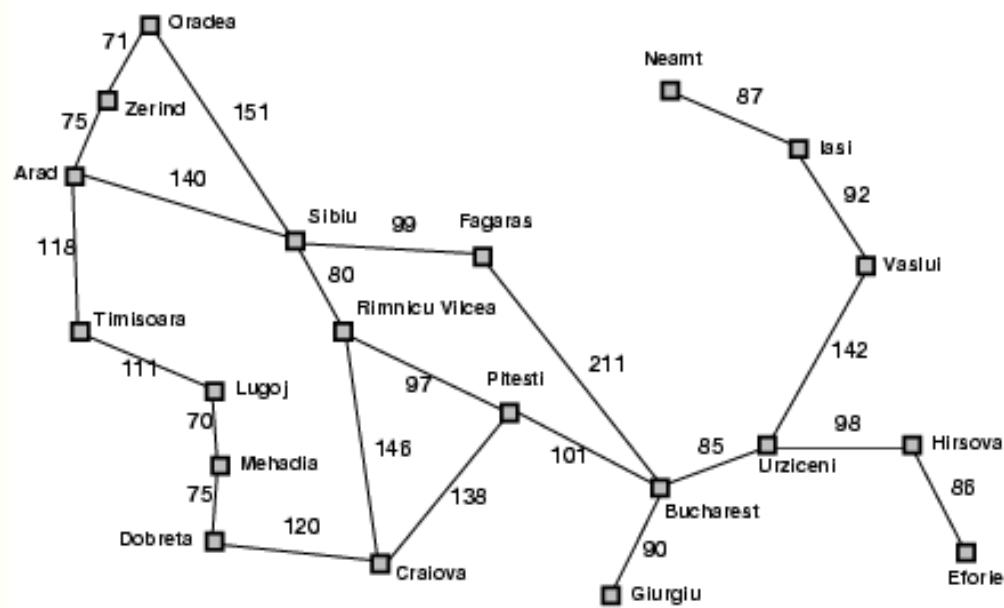
keeps all checked nodes
in memory to avoid repeated
states

- **Theorem:**

If $h(n)$ is consistent, A* using GRAPH-SEARCH is optimal

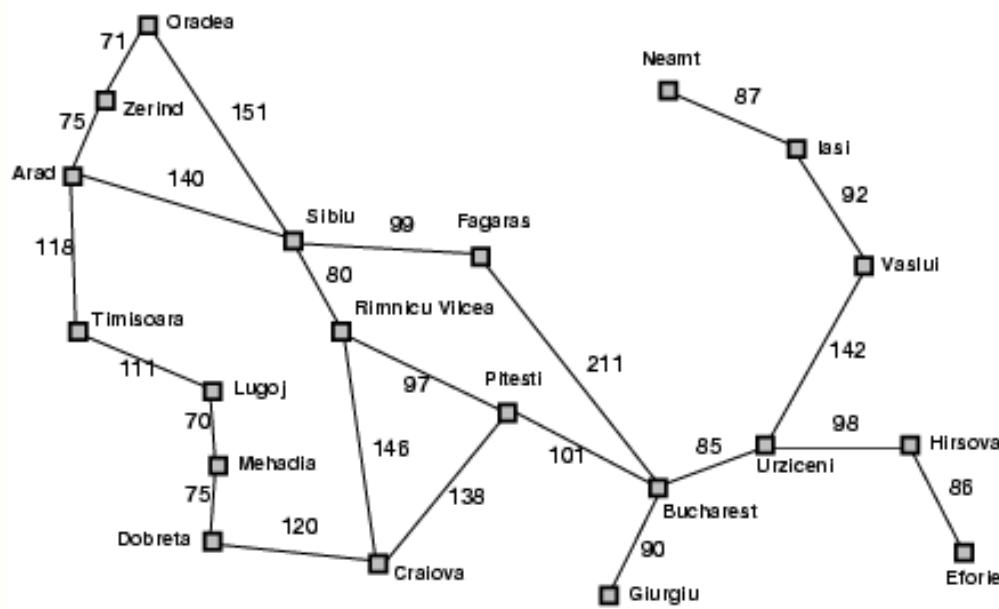
A* search example

► Arad
366=0+366

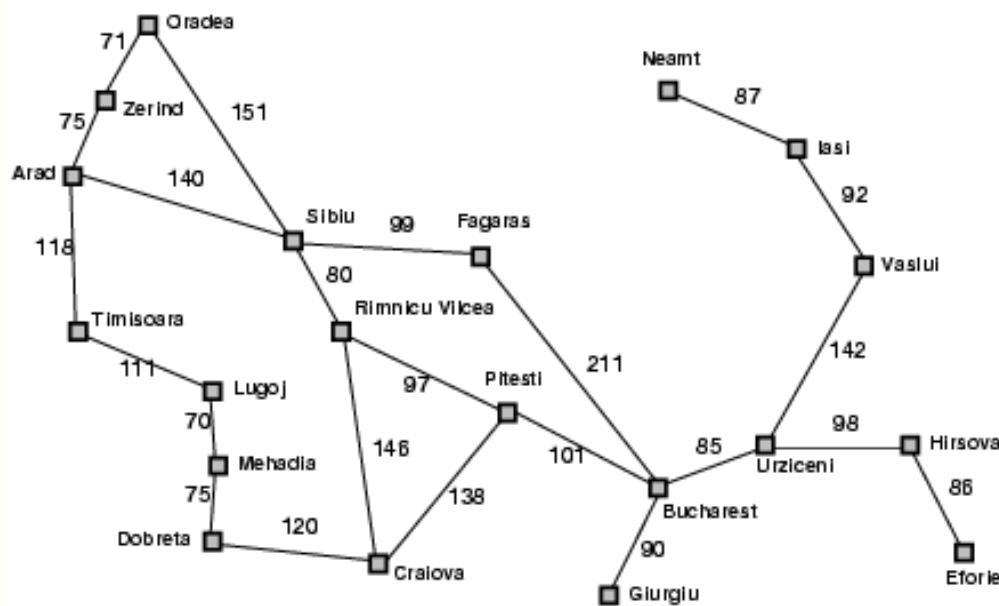
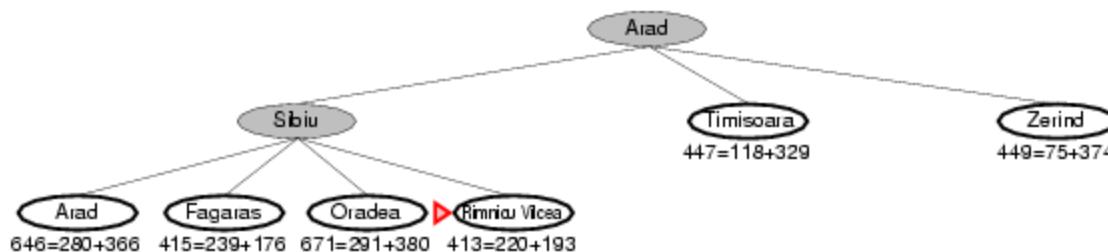


Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitești	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

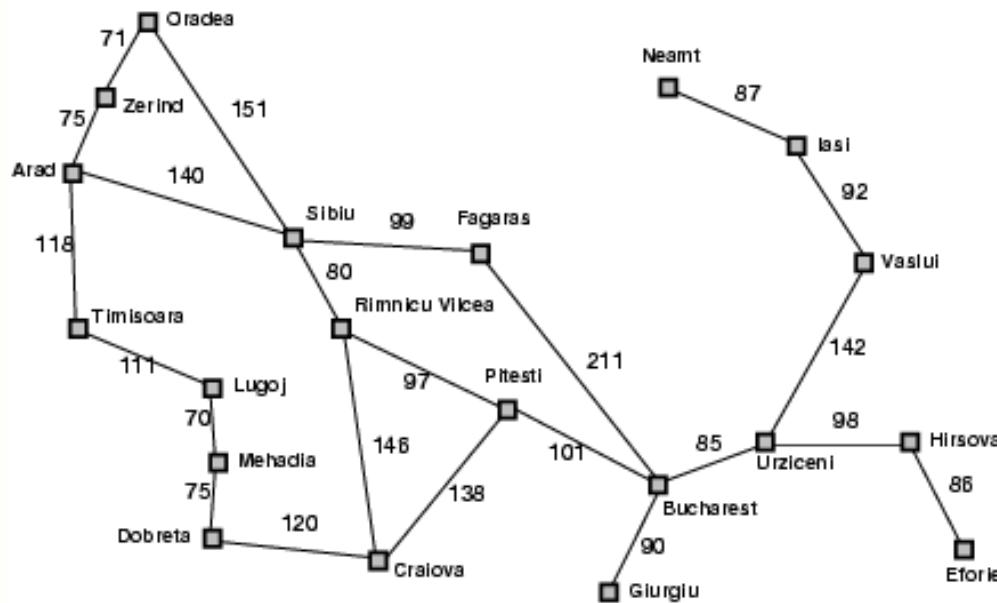
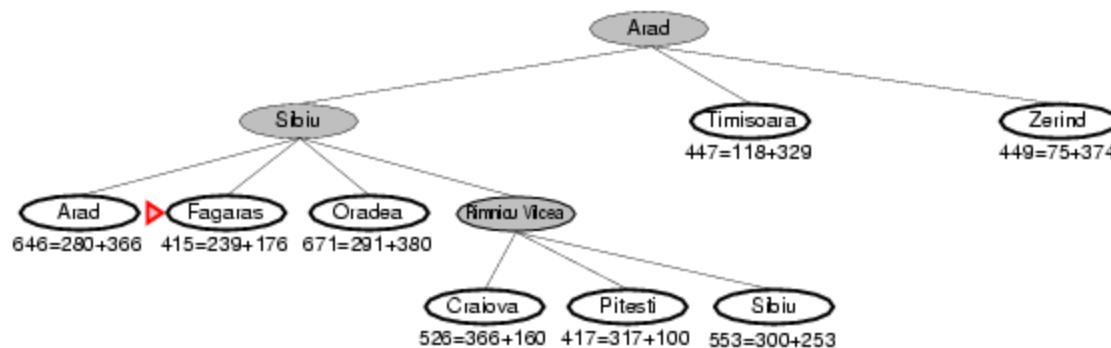
A* search example



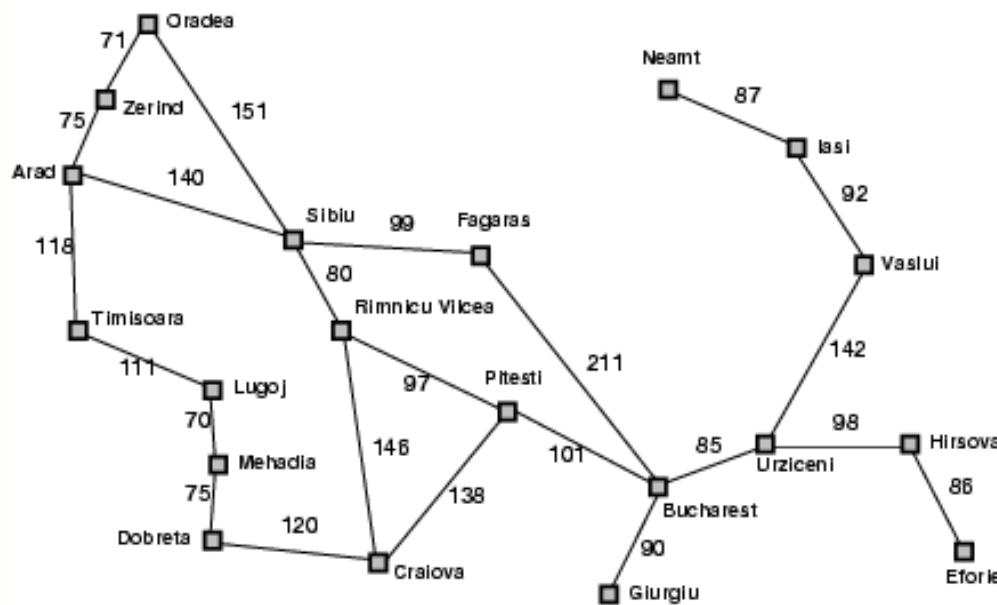
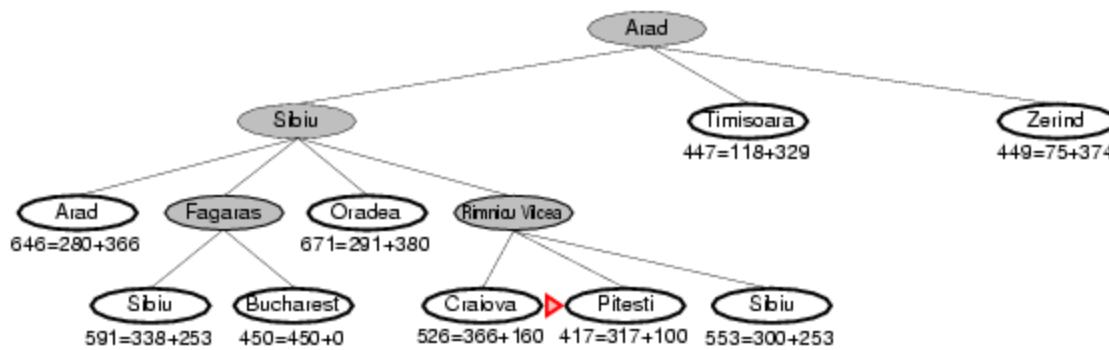
A* search example



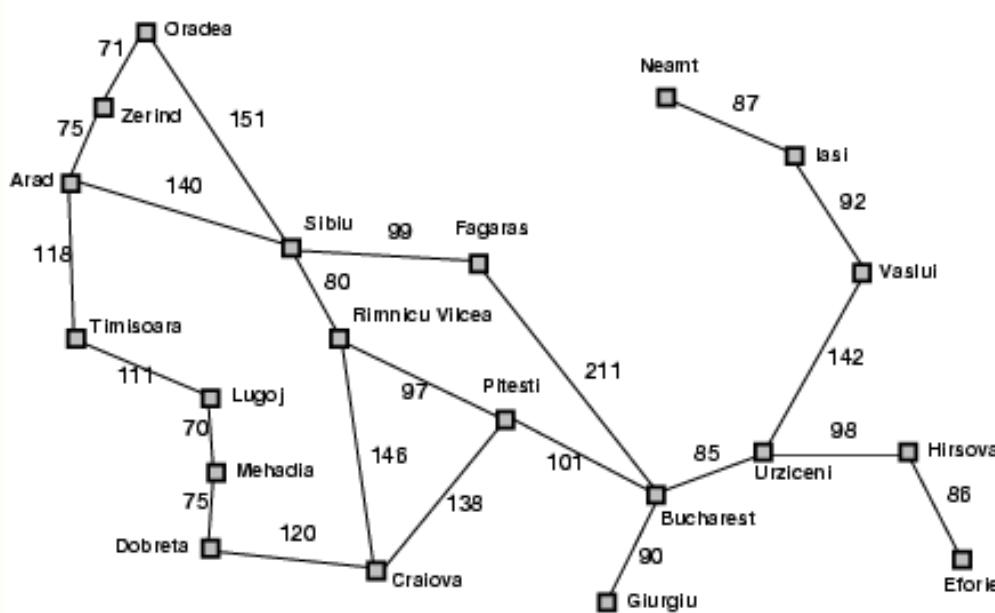
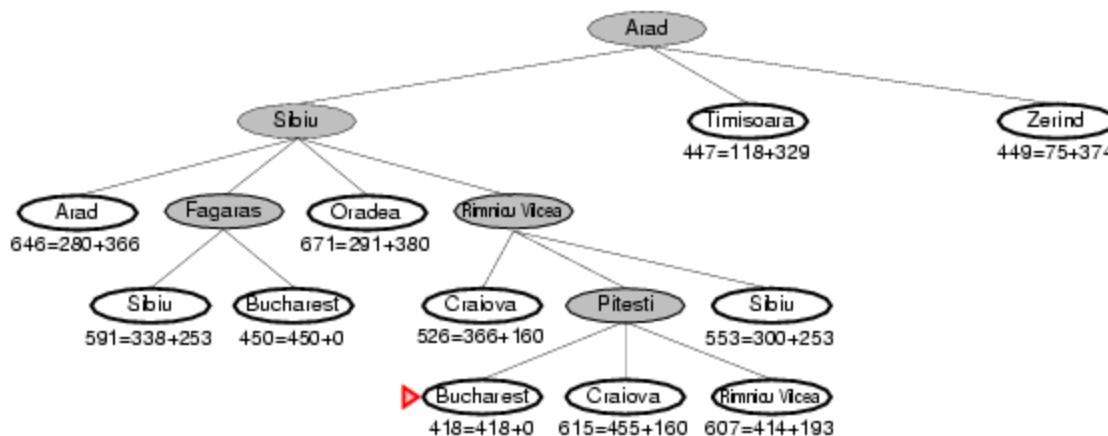
A* search example



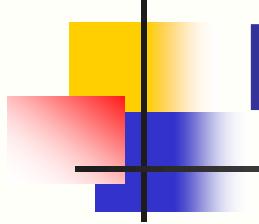
A* search example



A* search example

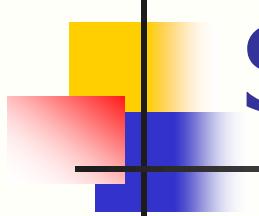


Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374



Properties of A*

- Complete? Yes (unless there are infinitely many nodes with $f \leq f(G)$, i.e. step-cost $> \varepsilon$)
- Time/Space? Exponential: b^d
except if: $|h(n) - h^*(n)| \leq O(\log h^*(n))$
- Optimal? Yes
- Optimally Efficient: Yes (no algorithm with the same heuristic is guaranteed to expand fewer nodes)



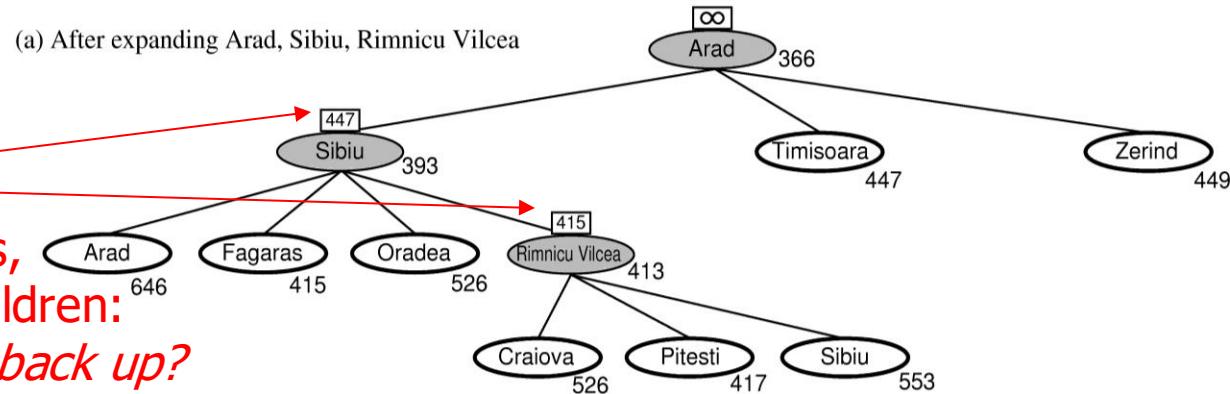
Memory Bounded Heuristic Search: Recursive BFS

- How can we solve the memory problem for A* search?
- Idea: Try something like depth first search, but let's not forget everything about the branches we have partially explored.
- *We remember the best f-value we have found so far in the branch we are deleting.*

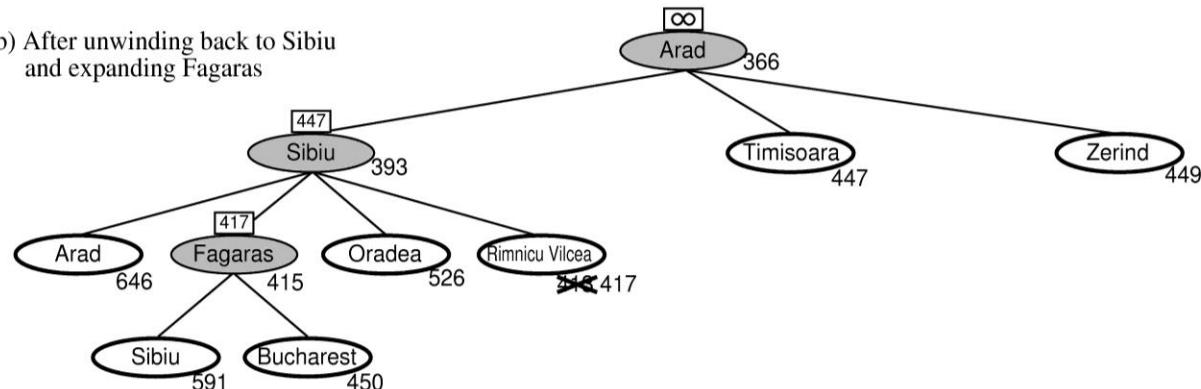
RBFS:

best alternative
over fringe nodes,
which are not children:
i.e. do I want to back up?

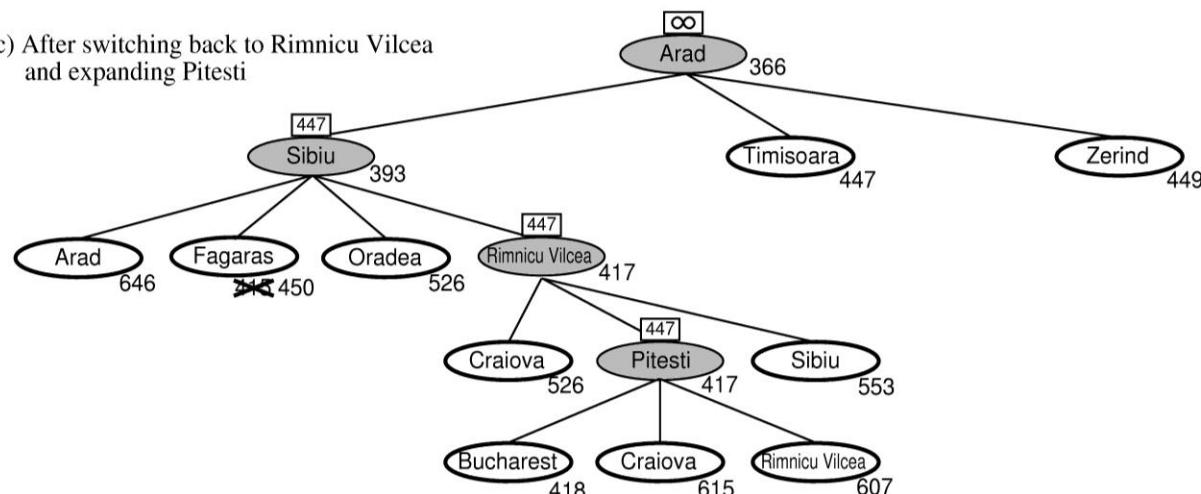
(a) After expanding Arad, Sibiu, Rimnicu Vilcea



(b) After unwinding back to Sibiu
and expanding Fagaras



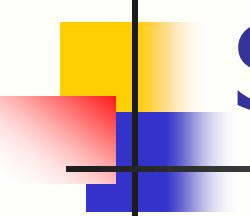
(c) After switching back to Rimnicu Vilcea
and expanding Pitesti



RBFS changes its mind
very often in practice.

This is because the
 $f = g + h$ become more
accurate (less optimistic)
as we approach the goal.
Hence, higher level nodes
have smaller f-values and
will be explored first.

Problem: We should keep
in memory whatever we can.



Simple Memory Bounded A*

- This is like A*, but when memory is full we delete the worst node (largest f-value).
- Like RBFS, we remember the best descendent in the branch we delete.
- If there is a tie (equal f-values) we delete the oldest nodes first.
- simple-MBA* finds the optimal *reachable* solution given the memory constraint.
- Time can still be exponential.

A Solution is not reachable if a single path from root to goal does not fit into memory

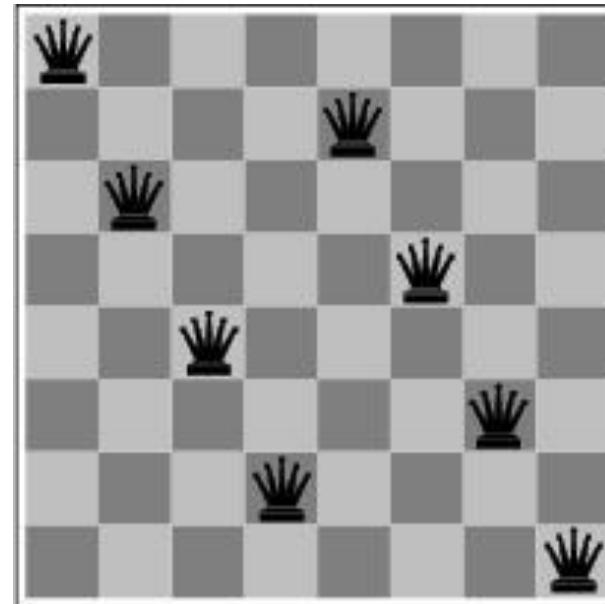
Chapter 4 (Section 4.3, ...) 2nd Edition
or
Chapter 4 (3rd Edition)
Local Search and Optimization

Outline

- Local search techniques and optimization
 - Hill-climbing
 - Gradient methods
 - Simulated annealing
 - Genetic algorithms
 - Issues with local search

Local search and optimization

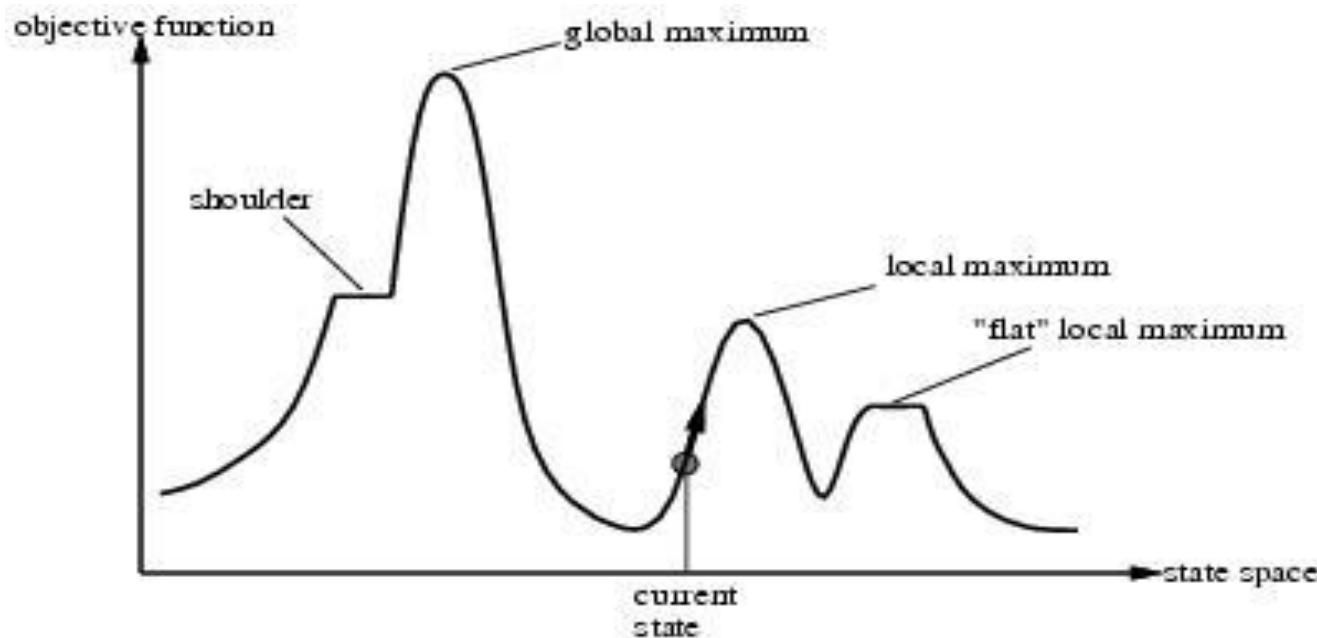
- Previously: systematic exploration of search space.
 - Path to goal is solution to problem
- YET, for some problems path is irrelevant.
 - E.g 8-queens
- Different algorithms can be used
 - Local search



Local search and optimization

- Local search
 - Keep track of single current state
 - Move only to neighboring states
 - Ignore paths
- Advantages:
 - Use very little memory
 - Can often find reasonable solutions in large or infinite (continuous) state spaces.
- “Pure optimization” problems
 - All states have an objective function
 - Goal is to find state with max (or min) objective value
 - Does not quite fit into path-cost/goal-state formulation
 - Local search can do quite well on these problems.

“Landscape” of search



Hill-climbing search

function HILL-CLIMBING(*problem*) **return** a state that is a local maximum

input: *problem*, a problem

local variables: *current*, a node.
neighbor, a node.

current \leftarrow MAKE-NODE(INITIAL-STATE[*problem*])

loop do

neighbor \leftarrow a highest valued successor of *current*

if VALUE [*neighbor*] \leq VALUE[*current*] **then return** STATE[*current*]

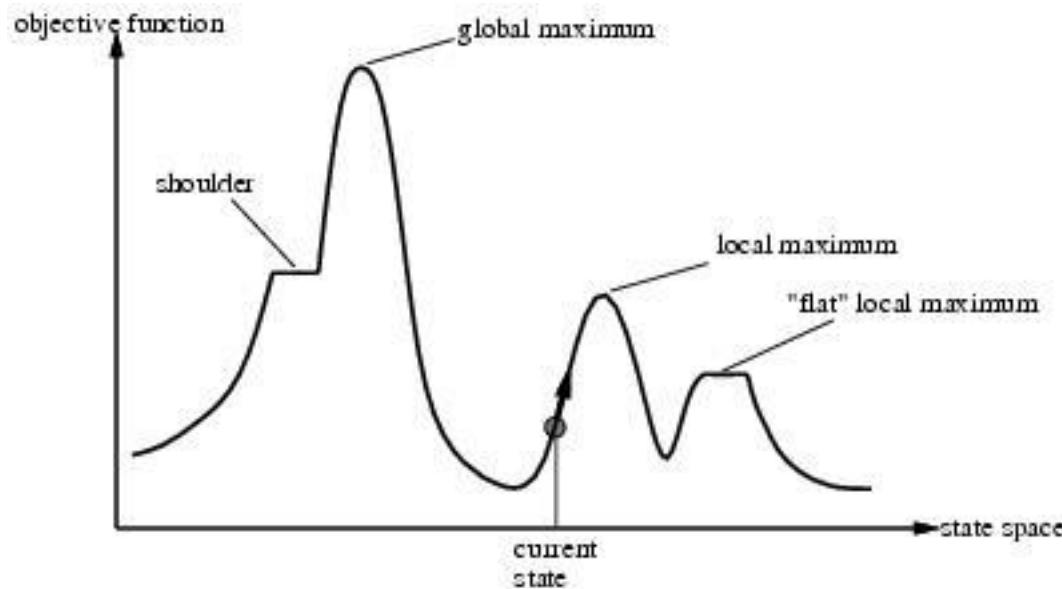
current \leftarrow *neighbor*

Hill-climbing search

- “a loop that continuously moves in the direction of increasing value”
 - terminates when a peak is reached
 - Aka greedy local search
- Value can be either
 - Objective function value
 - Heuristic function value (minimized)
- Hill climbing does not look ahead of the immediate neighbors of the current state.
- Can randomly choose among the set of best successors, if multiple have the best value
- Characterized as “trying to find the top of Mount Everest while in a thick fog”

Hill climbing and local maxima

- When local maxima exist, hill climbing is suboptimal
- Simple (often effective) solution
 - Multiple random restarts



Hill-climbing example

- 8-queens problem, complete-state formulation
 - All 8 queens on the board in some configuration
- Successor function:
 - move a single queen to another square in the same column.
- Example of a heuristic function $h(n)$:
 - the number of pairs of queens that are attacking each other (directly or indirectly)
 - (so we want to minimize this)

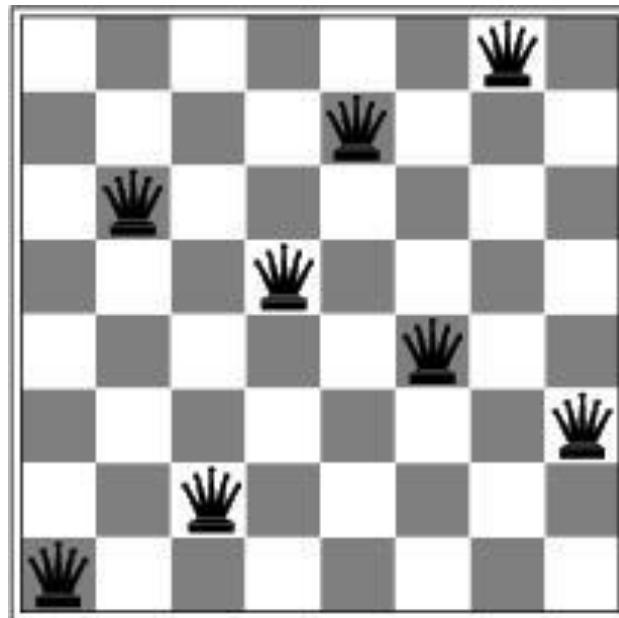
Hill-climbing example

18	12	14	13	13	12	14	14
14	16	13	15	12	14	12	16
14	12	18	13	15	12	14	14
15	14	14	14	15	13	16	13
15	14	14	14	15	13	16	16
14	14	17	15	15	14	16	16
17	15	16	18	15	15	15	16
18	14	15	15	15	14	15	16
14	14	13	17	12	14	12	18

Current state: $h=17$

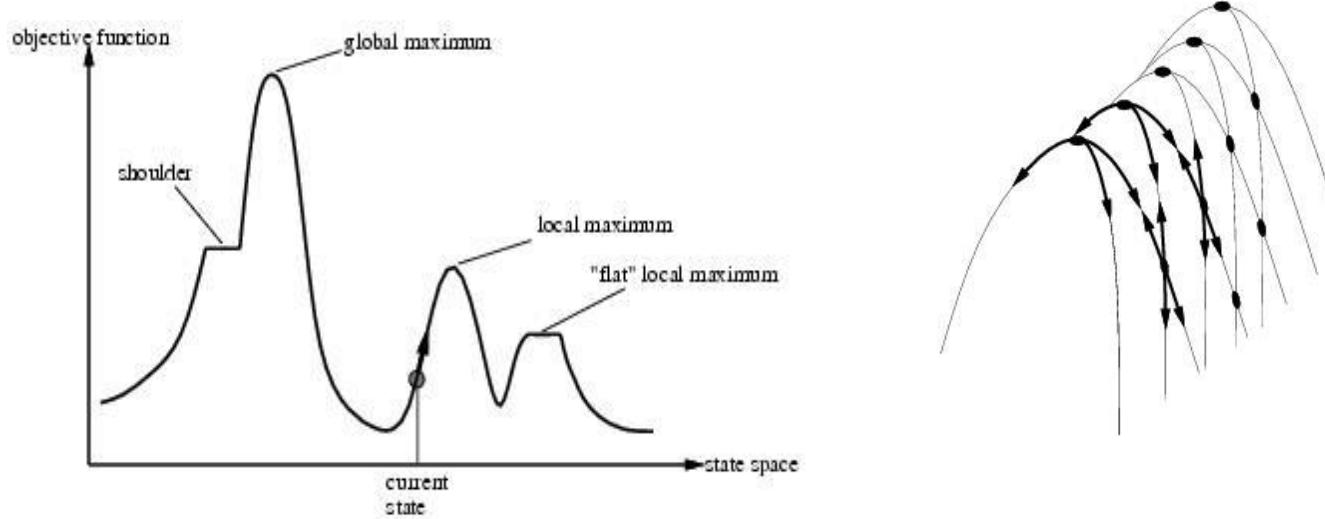
Shown is the h -value for each possible successor in each column

A local minimum for 8-queens



A local minimum in the 8-queens state space ($h=1$)

Other drawbacks



- Ridge = sequence of local maxima difficult for greedy algorithms to navigate
- Plateau = an area of the state space where the evaluation function is flat.

Performance of hill-climbing on 8-queens

- Randomly generated 8-queens starting states...
- 14% the time it solves the problem
- 86% of the time it get stuck at a local minimum
- However...
 - Takes only 4 steps on average when it succeeds
 - And 3 on average when it gets stuck
 - (for a state space with \sim 17 million states)

Possible solution...sideways moves

- If no downhill (uphill) moves, allow sideways moves in hope that algorithm can escape
 - Need to place a limit on the possible number of sideways moves to avoid infinite loops
- For 8-queens
 - Now allow sideways moves with a limit of 100
 - Raises percentage of problem instances solved from 14 to 94%
 - However....
 - 21 steps for every successful solution
 - 64 for each failure

Hill-climbing variations

- Stochastic hill-climbing
 - Random selection among the uphill moves.
 - The selection probability can vary with the steepness of the uphill move.
- First-choice hill-climbing
 - stochastic hill climbing by generating successors randomly until a better one is found
 - Useful when there are a very large number of successors
- Random-restart hill-climbing
 - Tries to avoid getting stuck in local maxima.

Hill-climbing with random restarts

- Different variations
 - For each restart: run until termination v. run for a fixed time
 - Run a fixed number of restarts or run indefinitely
- Analysis
 - Say each search has probability p of success
 - E.g., for 8-queens, $p = 0.14$ with no sideways moves
 - Expected number of restarts?
 - Expected number of steps taken?

Expected number of restarts

- Probability of Success = p
- Number of restarts = $1 / p$
- This means 1 successful iteration after $(1/p - 1)$ failed iterations
- Let avg. number of steps in a failure iteration = f
and avg. number of steps in a successful iteration = s

Therefore, expected number of steps in random-restart hill climbing = $1 * s + (1/p - 1) f$

So for 8-queens, $p = 14\%$, $s = 4$, $f = 3$,

Expected no of moves = $1 * 4 + (1/0.14 - 1) * 3 = 22$

With sideways moves, $p = 94\%$, $s = 21$, $f = 64$

Expected no of moves = $1 * 21 + (1/0.94 - 1) * 64 = 25$

Local beam search

- Keep track of k states instead of one
 - Initially: k randomly selected states
 - Next: determine all successors of k states
 - If any of successors is goal \rightarrow finished
 - Else select k best from successors and repeat.
- Major difference with random-restart search
 - Information is shared among k search threads.
- Can suffer from lack of diversity.
 - Stochastic beam search
 - choose k successors proportional to state quality.

Gradient Descent

Assume we have some cost-function: $C(x_1, \dots, x_n)$
and we want minimize over continuous variables X_1, X_2, \dots, X_n

1. Compute the *gradient*: $\frac{\partial}{\partial x_i} C(x_1, \dots, x_n) \quad \forall i$

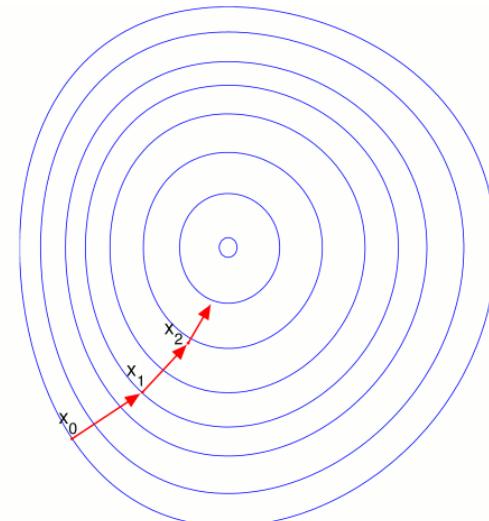
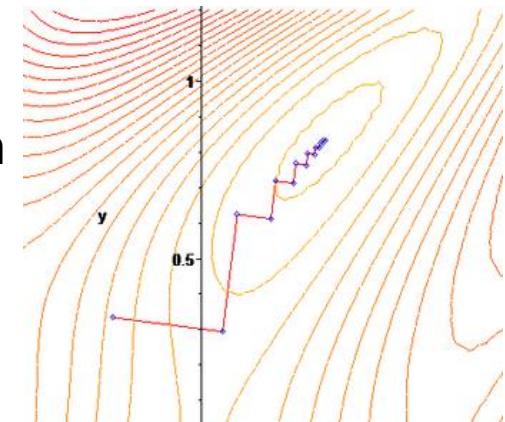
2. Take a small step downhill in the direction of the gradient:

$$x_i \rightarrow x'_i = x_i - \lambda \frac{\partial}{\partial x_i} C(x_1, \dots, x_n) \quad \forall i$$

3. Check if $C(x_1, \dots, x'_i, \dots, x_n) < C(x_1, \dots, x_i, \dots, x_n)$

4. If true then accept move, if not reject.

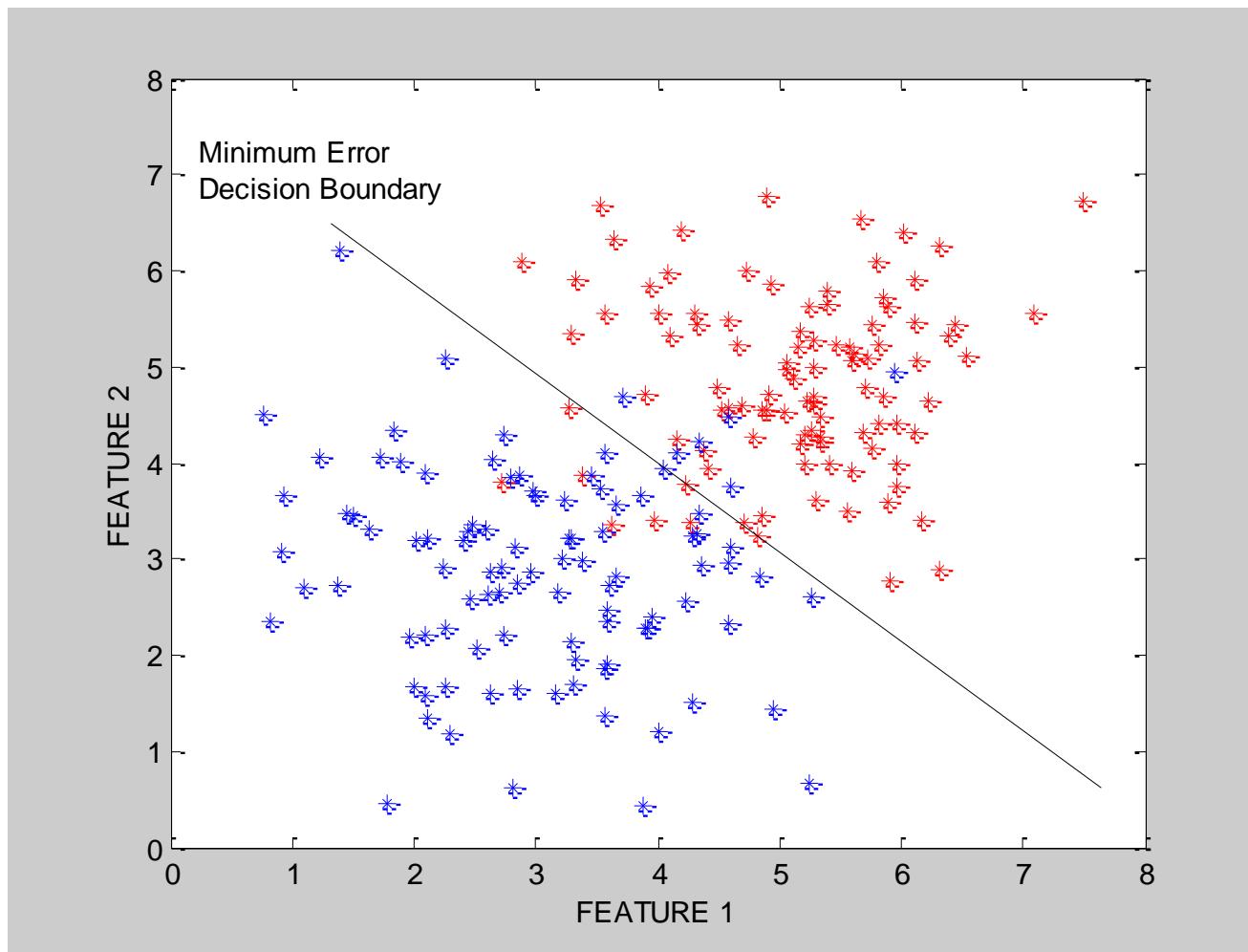
5. Repeat.



Learning as optimization

- Many machine learning problems can be cast as optimization
- Example:
 - Training data $D = \{(\underline{x}_1, c_1), \dots, (\underline{x}_n, c_n)\}$
where \underline{x}_i = feature or attribute vector
and c_i = class label (say binary-valued)
 - We have a model (a function or classifier) that maps from x to c
e.g., $\text{sign}(\underline{w} \cdot \underline{x}') = \{-1, +1\}$
 - We can measure the error $E(\underline{w})$ for any setting of the weights \underline{w} ,
and given a training data set D
 - Optimization problem: find the weight vector that minimizes $E(\underline{w})$
(general idea is “empirical error minimization”)

Learning a minimum error decision boundary



Search using Simulated Annealing

- Simulated Annealing = hill-climbing with non-deterministic search
- Basic ideas:
 - like hill-climbing identify the quality of the local improvements
 - instead of picking the best move, pick one randomly
 - say the change in objective function is δ
 - if δ is positive, then move to that state
 - otherwise:
 - move to this state with probability proportional to δ
 - thus: worse moves (very large negative δ) are executed less often
 - however, there is always a chance of escaping from local maxima
 - over time, make it less likely to accept locally bad moves
 - (Can also make the size of the move random as well, i.e., allow “large” steps in state space)

Physical Interpretation of Simulated Annealing

- A Physical Analogy:
 - imagine letting a ball roll downhill on the function surface
 - this is like hill-climbing (for minimization)
 - now imagine shaking the surface, while the ball rolls, gradually reducing the amount of shaking
 - this is like simulated annealing
- Annealing = physical process of cooling a liquid or metal until particles achieve a certain frozen crystal state
 - simulated annealing:
 - free variables are like particles
 - seek “low energy” (high quality) configuration
 - get this by slowly reducing temperature T , which particles move around randomly

Simulated annealing

```
function SIMULATED-ANNEALING( problem, schedule) return a solution state
  input: problem, a problem
          schedule, a mapping from time to temperature
  local variables: current, a node.
                      next, a node.
                      T, a "temperature" controlling the probability of downward steps

  current  $\leftarrow$  MAKE-NODE(INITIAL-STATE[problem])
  for t  $\leftarrow$  1 to  $\infty$  do
    T  $\leftarrow$  schedule[t]
    if T = 0 then return current
    next  $\leftarrow$  a randomly selected successor of current
     $\Delta E \leftarrow$  VALUE[next] - VALUE[current]
    if  $\Delta E > 0$  then current  $\leftarrow$  next
    else current  $\leftarrow$  next only with probability  $e^{\Delta E / T}$ 
```

More Details on Simulated Annealing

- Lets say there are 3 moves available, with changes in the objective function of $d_1 = -0.1$, $d_2 = 0.5$, $d_3 = -5$. (Let $T = 1$).
- pick a move randomly:
 - if d_2 is picked, move there.
 - if d_1 or d_3 are picked, probability of move = $\exp(d/T)$
 - move 1: $\text{prob1} = \exp(-0.1) = 0.9$,
 - i.e., 90% of the time we will accept this move
 - move 3: $\text{prob3} = \exp(-5) = 0.05$
 - i.e., 5% of the time we will accept this move
- T = “temperature” parameter
 - high T => probability of “locally bad” move is higher
 - low T => probability of “locally bad” move is lower
 - typically, T is decreased as the algorithm runs longer
 - i.e., there is a “temperature schedule”

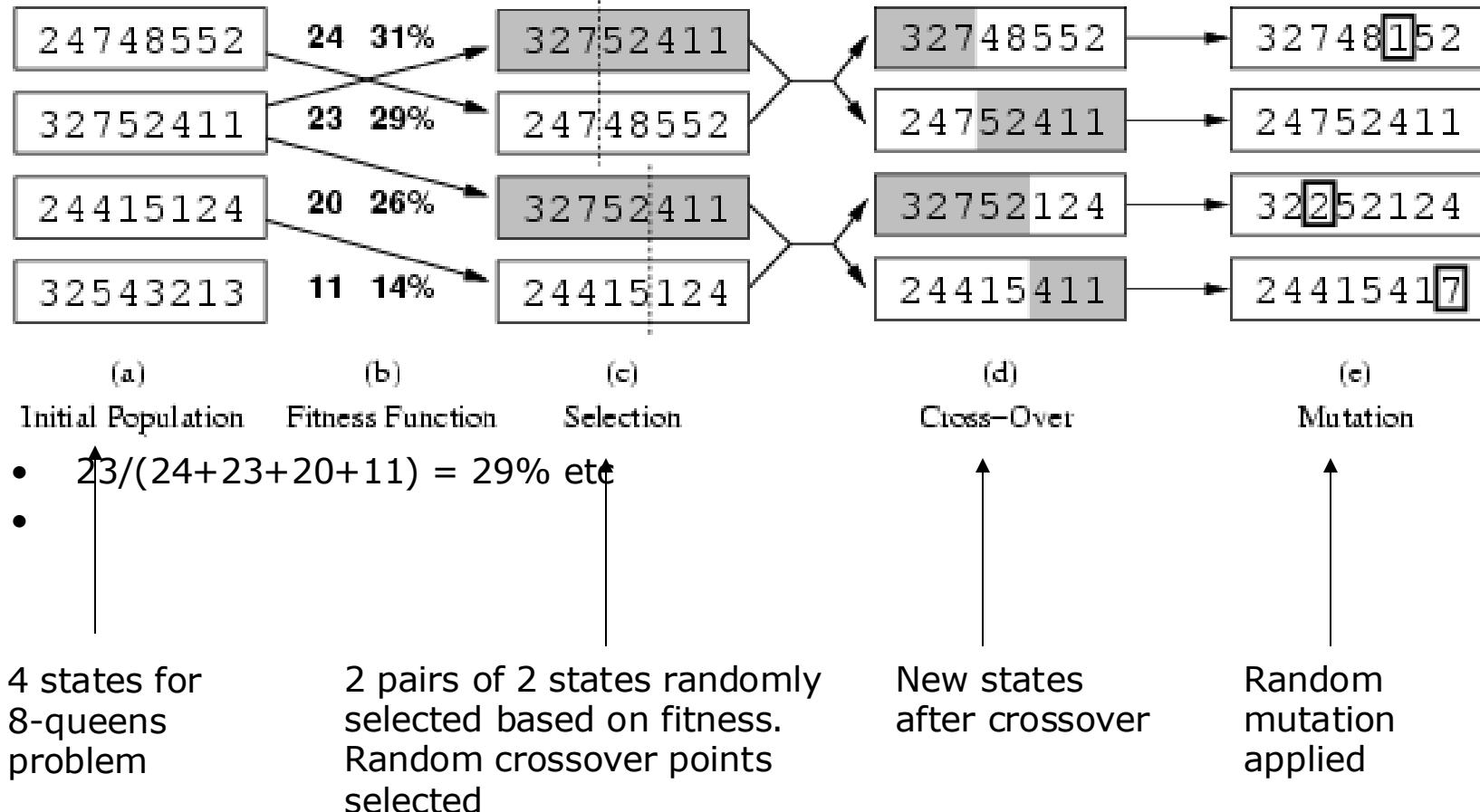
Simulated Annealing in Practice

- method proposed in 1983 by IBM researchers for solving VLSI layout problems (Kirkpatrick et al, *Science*, 220:671-680, 1983).
 - theoretically will always find the global optimum (the best solution)
- useful for some problems, but can be very slow
 - slowness comes about because T must be decreased very gradually to retain optimality
 - In practice how do we decide the rate at which to decrease T ? (this is a practical problem with this method)

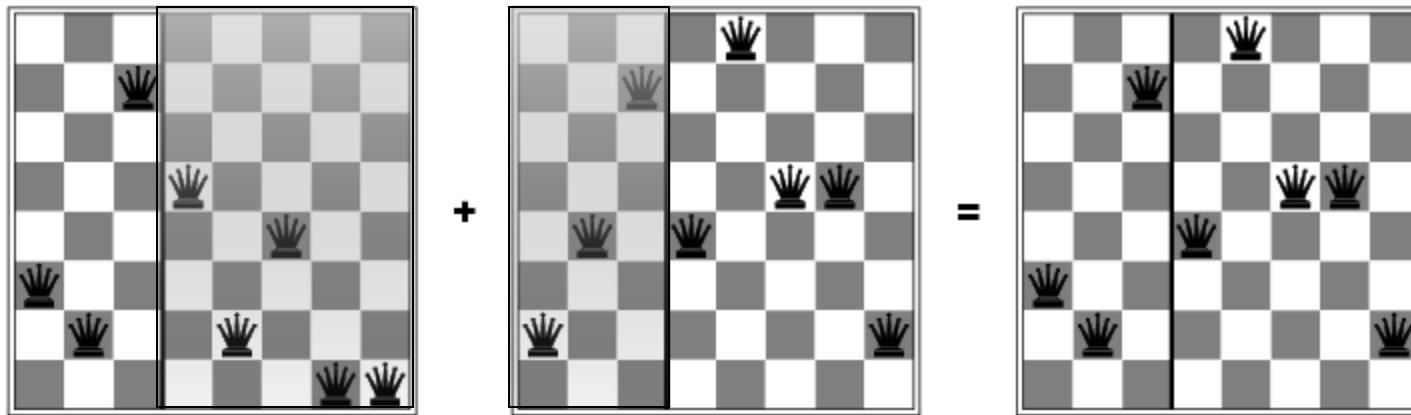
Genetic algorithms

- Different approach to other search algorithms
 - A successor state is generated by combining two parent states
 -
- A state is represented as a string over a finite alphabet (e.g. binary)
 - 8-queens
 - State = position of 8 queens each in a column
 - $= 8 \times \log(8)$ bits = 24 bits (for binary representation)
- Start with k randomly generated states (**population**)
-
- Evaluation function (**fitness function**).
 - Higher values for better states.
 -
 - Opposite to heuristic function, e.g., # non-attacking pairs in 8-queens
- Produce the next generation of states by “simulated evolution”
 - Random selection
 - Crossover
 - Random mutation
 -

Genetic algorithms



Genetic algorithms



Has the effect of “jumping” to a completely different new part of the search space (quite non-local)

Genetic algorithm pseudocode

```
function GENETIC_ALGORITHM( population, FITNESS-FN) return an individual
  input: population, a set of individuals
        FITNESS-FN, a function which determines the quality of the individual
  repeat
    new_population  $\leftarrow$  empty set
    loop for i from 1 to SIZE(population) do
      x  $\leftarrow$  RANDOM_SELECTION(population, FITNESS_FN)
      y  $\leftarrow$  RANDOM_SELECTION(population, FITNESS_FN)
      child  $\leftarrow$  REPRODUCE(x,y)
      if (small random probability) then child  $\leftarrow$  MUTATE(child )
      add child to new_population
    population  $\leftarrow$  new_population
  until some individual is fit enough or enough time has elapsed
  return the best individual
```

Comments on genetic algorithms

- Positive points
 - Random exploration can find solutions that local search can't
 - (via crossover primarily)
 - Appealing connection to human evolution
 - E.g., see related area of genetic programming
- Negative points
 - Large number of "tunable" parameters
 - Difficult to replicate performance from one problem to another
 - Lack of good empirical studies comparing to simpler methods
 - Useful on some (small?) set of problems but no convincing evidence that GAs are better than hill-climbing w/random restarts in general

Summary

- Local search techniques and optimization
 - Hill-climbing
 - Gradient methods
 - Simulated annealing
 - Genetic algorithms
 - Issues with local search

Neural Network: Algorithms and Methods

February 22, 2010

Chapter 1

Alternative Learning Algorithms

An important issue in training neural networks is that the networks should not become paralyzed or stuck into poor local minima, which are far from the global minimum. We introduced several learning algorithms in the previous chapter to train neural works. The majority of those algorithms, specifically gradient based algorithms, have a tendency to trap into local minima. To alleviate this problem, we may apply a number of tricks such as restarting training with a new random set of weights, training with noisy exemplars, and perturbing the weights. All these tricks can be applied when we find that a network appears to prematurely converge. While these tricks may lead to improved solutions, there is no guarantee that such solutions will not also be only locally optimal. Further, the same suboptimal solution may be rediscovered, leading to fruitless oscillatory training behavior.

All learning algorithms introduced in the previous chapter use some sort of gradient information during training to update the weights of a neural network. This requirement enforces that an activation function of the neural network must be differentiable, although some activation functions may not be differentiable but suitable for practical purposes. For instance, a threshold logic activation function has only two states: on and off, which make them easily implementable in hardware. However, it would not be possible to learn appropriate weights for neural networks with the threshold function because most classical learning algorithms requires a differentiable

activation function. We thus require learning algorithms that have abilities in escaping from local optima and can work with both differentiable or non-differentiable activation functions. In this chapter, we shall introduce three learning algorithms that neither suffer from entrapment in local minima nor depend on the differentiability of an activation function. These learning algorithms are developed based on metaheuristics and can be used independently or in conjunction with classical learning algorithms to train neural networks. Porto *et al.* (1995) termed the metaheuristic based learning algorithms as *alternative* learning algorithms.

A metaheuristic is a heuristic that combines user-specified black-box procedures to solve a very general class of computational problems for which there is no satisfactory problem-specific algorithm or heuristic; or when it is not practical to implement such a method. The name metaheuristics combines the Greek prefix “meta” (“beyond”, here in the sense of “higher level”) and “heuristic” (“to find”). The recent success of metaheuristics is clearly recognizable in an increasing number of books [3, 207], collections of articles [204, 194], special issues of journals [97, 202, 149] and a new conference series, Metaheuristics International Conference, that concentrates solely on this subject. Metaheuristics based training can avoid local minima either by accepting worse solutions or by generating good starting solutions to guide a training process towards better solutions. In the latter case, often the experience accumulated during training is used to guide the search in subsequent iterations. Furthermore metaheuristics based learning algorithms do not place restrictions on the network topology and activation function. A number of metaheuristic have been proposed in the literature such as simulated annealing, genetic algorithm, evolutionary programming, evolution strategies, ant colony optimization and tabu search. In this chapter, we shall briefly describe the first three metaheuristic methods and their application to training neural networks. For more detail of these methods, interested readers can look the books [10].

1.1 Simulated Annealing

Simulated annealing is a robust and general search technique that has attracted significant attention due to its suitability for optimization problems, especially for some problems where a desired global solution is hidden in

many poorer local solutions. This technique has been applied to the famous traveling salesman problem, integrated circuit design, image restoration, graph partitioning, routing and many other problems. While this list is far from exhaustive, it indicates that simulated annealing has a great scientific and industrial importance in solving various problems. Simulated annealing plays a special role within search for two reasons. Firstly, it appears to be quite successful when applied to a broad range of practical problems. Secondly, it uses a stochastic component that facilitates a theoretical analysis of its asymptotic convergence. The later feature makes simulated annealing very popular to mathematicians. At the heart of simulated annealing is an analogy to the statistical mechanics of annealing. We shall first describe the physics analogy of simulated annealing, then the method of simulated annealing and finally the use of simulated annealing to train neural networks.

1.1.1 The Physics Analogy

In metallurgy, annealing is a process of heating up a solid in a heat bath and then cooling slowly until it crystallizes. Simulated annealing uses an analogous set of “controlled cooling” operations for nonphysical optimization problems, in effect transforming a poor and unordered solution into a highly optimized and desirable solution. The annealing process consists of the following two steps.

- Increase temperature of the heat bath to a maximum value at which the solid melts.
- Decrease temperature carefully until the atoms of the melted solid rearrange themselves in a ground state.

The atoms have high energies at very high temperatures. This causes the atoms to become unstuck from their initial positions and gives them a great deal of freedom to move randomly through high energy states. As temperature reduces, energy decreases and the atoms lose their mobility. However, the slow reduction of temperature gives the atoms more chances to rearrange and form crystal, a highly regular atomic structure. This structure is the state of a minimum energy for the system. The amazing fact is that, for a slowly cooled system, nature is able to find such a minimum energy state. In contrast, if we reduce temperature too quickly, the crystal is not

formed; rather it ends up in a polycrystalline or amorphous state having somewhat higher energy. The process of quick reduction of temperature is called *quenching*.

We can model the physical annealing process by computer simulation based on Monte Carlo techniques that rely on repeated random samplings to simulate physical and mathematical systems. In a standard Monte Carlo technique, the state of a system is randomly changed and all such changes are accepted regardless the feasibility of the new states. This acceptance criterion may cause the system to fall into unreasonable states for much of the time. Thus computer simulation would have to run for a long time to sample properly all feasible states of the system. An introductory overview of the use of Monte Carlo techniques in statistical physics is given by Binder (1978). We here discuss one of the early techniques known as the *Metropolis algorithm*. In 1953, Nicholas Metropolis and coworkers proposed a new algorithm based on the Monte Carlo technique for simulating the evolution of solids in a heat bath to thermal equilibrium. The modification made by Metropolis and coworkers is the use of a probabilistic acceptance criterion to accept the new states produced by repeated random samplings. The probabilistic acceptance criterion is known as the *Metropolis criterion*.

Let the current state of the solid is i and its energy E_i . The Metropolis algorithm generates the next state j by perturbing the current state i , for instance, by displacing a single atom. This small change makes the new state near to the old one. Let E_j be the energy of the new state. If the energy difference, i.e., $E_i - E_j$, is less than or equal to 0, the Metropolis algorithm accepts j as a current state. Otherwise, the algorithm accepts j as the current state with some probability. The probability of accepting j as the new state is defined as

$$P \{ \text{accept } j \} = \begin{cases} 1 & \text{if } E_j \leq E_i \\ \exp \left(\frac{E_i - E_j}{k_B T} \right) & \text{if } E_j > E_i \end{cases} \quad (1.1)$$

where k_B is a physical constant known as the Boltzmann constant and T denotes the temperature of the heat bath. A characteristic feature of the acceptance criterion is that it not only accepts improvement (energy reduction), but also accepts deterioration (energy increase) to some extent. This criterion is quite different from the one used in a greedy approach, e.g. gra-

dient descent learning algorithm, that accepts only improvement. Accepting deterioration in contrast to iterative improvement facilitates simulated annealing to escape from local optima.

Ref K. Binder, Monte Carlo Methods in statistical physics, Springer, Berlin.

1.1.2 The Method

S. Kirkpatrick and coworkers showed how the Metropolis algorithm provides a natural tool for bringing the techniques of statistical mechanics to bear on optimizations. By analogy the generalization of such an algorithm to an optimization problem is very straightforward. The current state of a thermodynamic system is analogous to the current solution of the optimization problem, the energy equation to an objective function, and the ground state to the global optimum (minimum or maximum). When simulated annealing was first introduced, it was used for designing complex integrated circuits and solving the traveling salesman problem.

Notice that the two applications cited above are both examples of combinatorial problems. The aim of these problems is to minimize an objective function. The space over which that function is defined is the N -dimensional space of N discrete parameters, like the set of possible orders of cities for the traveling salesman problem, or the set of possible allocations of element to construct integrated circuits. While combinatorial optimization problems have wide applicability, there is a large variety of optimization problems involving continuous parameters but not discrete. Assume we like to fit a function to a sum of exponentials. The choice of the decay constant is an optimization problem. Continuous parameter optimization problems also frequently arise in many other contexts such as engineering design, econometrics, and data analysis. In this chapter, the concept of simulated annealing will be employed for training neural networks, a kind of continuous parameter optimization problem. It is natural to describe simulated annealing in the context of continuous parameter optimization.

Consider a function $f(x_1, x_2, \dots, x_n) = f(\mathbf{x})$ is defined over an n -dimensional space of continuous parameters, We like to minimize f with respect of \mathbf{x} . The function f may be the energy of a physical system, the error in a fitting problem or any other “objective function”. It is clear from

the preceding section that the Metropolis algorithm has elements. In the context of our minimization problem, these elements are as follows. The function, f , is the objective function. The system state is \mathbf{x} . A generator of random variations to change the system states, that is, a procedure for taking a random step from \mathbf{x} to $\mathbf{x} + \Delta\mathbf{x}$. Finally, a control parameter, something like a temperature, with an annealing schedule that describes how the temperature is to be lowered from high to low values. The pseudo code of the method of simulated annealing is presented in *Algorithm 3*

The method of simulated annealing proceeds by choosing an initial state $\mathbf{x}^{(0)}$ and making random steps $\Delta\mathbf{x}$. A natural way to chose $\Delta\mathbf{x}$ is to call a random number generator n times to produce $\Delta x_1, \dots, \Delta x_n$. At each step, the method evaluates the change

$$\Delta f = f(\mathbf{x} + \Delta\mathbf{x}) - f(\mathbf{x}) \quad (1.2)$$

in the objective function value. If Δf is negative (improved) in terms of the objective function value, the method of simulated accepts the new state $\mathbf{x}^{(1)} = \mathbf{x}^{(0)} + \Delta\mathbf{x}$ as a current state. In contrast, if Δf is positive (deteriorated) in terms of the objective function value, the new state is accepted with a probability p , which is calculated based on the new fitness value, $f(\mathbf{x}^{(0)} + \Delta\mathbf{x})$, relative to the previous fitness value, $f(\mathbf{x}^{(0)})$. That is,

$$\begin{aligned} p &= \frac{\exp(-f(\mathbf{x}^{(0)} + \Delta\mathbf{x})/T)}{\exp(-f(\mathbf{x}^{(0)} + \Delta\mathbf{x})/T) + \exp(-f(\mathbf{x}^{(0)})/T)} \\ &= \frac{1}{1 + \exp(\Delta f/T)} \\ &\approx \exp(-\Delta f/T) \end{aligned} \quad (1.3)$$

where Δf is the increase in f and T is a control parameter, which by analogy with a thermodynamic system is “temperature” irrespective of the objective function being involved in the optimization process. The probability of accepting deterioration with respect to the value of f depends on the choice of T . The higher the value of T , the higher the probability of accepting deterioration. The method of simulated annealing decreases T during its execution. It uses a procedure called “cooling schedule” for decrementing T . Initially, at large values of T , large deteriorations will be accepted; as

T decreases, only smaller deteriorations will be accepted and finally, as the value of T approaches 0, no deteriorations will be accepted at all.

When T is reduced slowly enough, a system will avoid getting trapped into local minima because the probability of accepting new states expressed by (1.3) allows the deterioration f in contrast to iterative improvement to get over a barrier into a new local (or global) minimum. We can generate many new states and apply the aforementioned acceptance criterion many times successively for getting a series of accepted states in the parameter space. These states generate a random walk that explores the parameter space, and which at long times is governed by the probability distribution function

$$P(\mathbf{x}) = \frac{1}{Z} \exp\left(\frac{-\Delta f}{T}\right) \quad (1.4)$$

where $P(\mathbf{x})d^n x$ is the probability that the walk will be in the volume $d^n x$ on any given step at long times, and the normalization constant or “partition function” Z is given by

$$Z = \int d^n x \exp\left(\frac{-\Delta f}{T}\right). \quad (1.5)$$

We have now understood that the most important components in the method of simulated annealing are a generator of random changes and a cooling schedule. The choice of these two components greatly influence the solution quality and convergence of simulated annealing. We describe these components at length in the following sections.

1.1.3 Generating Function

The aim of a generating function in simulated annealing is to facilitate the search traversal from one state to another, for instance, \mathbf{x} to $\mathbf{x} + \Delta\mathbf{x}$, over an n -dimensional parameter space. We can generate $\Delta\mathbf{x}$ in a deterministic way or random fashion. The method of simulated annealing generates $\Delta\mathbf{x}$ randomly. The many applications of randomness have led to many different techniques for generating random numbers. These techniques may vary as to how unpredictable or statistically random they are, and how quickly they

can generate random numbers. The earliest methods for generating random numbers – dice, coin flipping, roulette wheels – are used mainly in games and gambling. These techniques can not be employed for generating Δx because they generate discrete numbers, thereby not suitable for continuous parameter optimization.

We like to generate random variables taking values in a continuum, such as \mathbb{R} and the interval $[a, b]$. How we can calculate the probability of events associated with a random variable X that takes values on the continuum. In probability theory and statistics, the *cumulative distribution function*, also called *probability distribution function* or just *distribution function*, describes completely the probability distribution of the real-valued random variable X . The distribution of X for every real number x is given by

$$x \mapsto F_X(x) = P(X \leq x) \quad (1.6)$$

where the right-hand side represents the probability that the random variable X takes on a value less than or equal to x . The probability that X lies in the interval $(a, b]$ is therefore $F_X(b) - F_X(a)$ if $a < b$. A function $F : \mathbb{R} \rightarrow \mathbb{R}$ is the distribution function of some random variable if and only if it has the following properties:

- i) F is non-decreasing (i.e., $F(x) \geq F(y)$ whenever $y > x$).
- ii) F is right-continuous i.e., $F(x + \epsilon) \geq F(y)$
- iii) F is normalized i.e., $\lim_{x \rightarrow \infty} F(x) = 0$, $\lim_{x \rightarrow -\infty} F(x) = 1$

One such function that has the aforementioned properties is the *normal distribution*, also called the *Gaussian distribution*. This is an important family of continuous probability distributions applicable in many fields. Each member of the family can be defined by two parameters, location and scale: the mean (“average”, μ) and variance (“standard deviation squared”, σ^2), respectively. Carl Friedrich Gauss (1777-1855) became associated with this set of distributions when he analyzed astronomical data and defined the equation of its probability density function.

The continuous probability distribution is often described in terms of another function called the *probability density function*. The one-dimensional probability density function of the Gaussian distribution is

$$\varphi_{\mu, \sigma^2}(x) = \varphi(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{-(x-\mu)^2}{2\sigma^2}}, \quad x \in \mathbb{R} \quad (1.7)$$

If we choose the value of μ to be zero, the probability density function becomes

$$\varphi(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{-x^2}{2\sigma^2}}, \quad x \in \mathbb{R} \quad (1.8)$$

The variance σ^2 controls the shape of the probability density function. To visualize the effect of σ^2 , we plot the probability density function for different σ s in the same scale (Fig. 2). This figure shows that, for a large σ , the probability density function distribution exhibits a central maximum with a long flat tail. This phenomena indicates that the probability density function involving a large σ^2 is likely to generate a large x . This feature is beneficial for exploring a search space. Similarly, a large central maximum with a small flat tail, which is obtained for a small σ^2 , is suitable for exploitation (fine tuning). As mentioned before, simulated annealing uses a large T at the beginning and gradually decreases T as the annealing process progresses. This is equivalent to exploration at the beginning and exploitation (fine tuning) at the later stage. We have now understood that σ^2 can be thought equivalent to T . If we replace σ^2 and Δx of Eq.(1.8) by T and x , respectively, we obtain

$$\varphi(\Delta x) = \frac{1}{\sqrt{2\pi T}} e^{\frac{-\Delta x^2}{2T}}, \quad \Delta x \in \mathbb{R} \quad (1.9)$$

The form of (1.9) is called the Gaussian form of the *Boltzmann probability density function*. For n -parameters, this density function can be written as

$$\varphi(\Delta \mathbf{x}) = \frac{1}{\sqrt[n]{2\pi T}} e^{\frac{-\Delta \mathbf{x}^2}{2T}}, \quad \Delta x \in \mathbb{R} \quad (1.10)$$

Another commonly used continuous probability distribution is the *Cauchy distribution*, which is also known as the *Lorentz distribution* among physicists. The one-dimensional Cauchy density function centered at the origin is defined by

$$\varphi(x) = \frac{1}{\pi} \frac{c}{c^2 + x^2} \quad x \in \mathbb{R} \quad (1.11)$$

where $c > 0$ is a scale parameter. To visualize the effect of c , we plot this density function for different c s in the same scale (Fig. 2). The shape of the Cauchy density function resembles that of the Gaussian density function, but the former one approaches the axis so slowly that an expectation does not exist. As a result, the variance of the Cauchy distribution is infinite. The Cauchy distribution is more likely to generate large variations due to its long flat tails than the Gaussian distribution. Any system with this feature have a higher probability of escaping from a local optimum or moving away from a plateau. If we replace c by T and x by Δx , the Cauchy density function for n -parameters can be written as

$$\varphi(\Delta x) = \frac{1}{\pi} \frac{T}{[T^2 + \Delta x^2]^{(n+1)/2}} \quad \Delta x \in \mathbb{R} \quad (1.12)$$

Amin will write here about a generating function based on the Gaussian and Cauchy distribution

1.1.4 Annealing Schedule

We have already seen that most features of simulated annealing, i.e., the state space, the new state generation and the objective function, are fixed by definition. The control parameter T is the only feature that simulated annealing modifies (decrements) during its execution using a annealing schedule through Eq.(1.3). The choice of this schedule is critical to the success of simulated annealing because it has affect on the acceptance criterion and the generating function as well. The principle underlying the choice of a suitable annealing schedule can easily be stated—the initial temperature should be high enough to “melt” a system completely and should be reduced as the search progresses towards the system’s “freezing point”— but choosing an annealing schedule for practical purposes is not easy. A number of annealing scheduling have been proposed in the literature. We here describe some of them.

An annealing schedule has usually four components: an initial temperature T_0 , a stopping criterion (something a final temperature), a rule for

temperature decrement, and **a finite number of state transitions at each temperature k** . The T_0 must be high enough so that search can traverse from the current state to its any neighbor. This may cause searching (at least in the early stages) into random walks. On the other hand, if T_0 is chosen not high enough, the final state would be very close to the starting state. It is, therefore, necessary to find an appropriate T_0 . At present, there is no known method to find a suitable T_0 for all problems. According to Kirkpatrick *et al.* (1984), a suitable T_0 is one that results in an average probability χ_0 of a solution that increases f being accepted of about 0.8. The value of T_0 is clearly dependent on the scaling of f and, hence, be problem-specific. We can estimate such a by conducting initial search in which all increases in f are accepted and calculating $\bar{\delta}f$, an average increases of f . The initial temperature then can be defined as

$$T_0 = \frac{\bar{\delta}f}{\ln(\chi_0)} \quad (1.13)$$

Rayward and Smith (1996) suggested to start with a very high temperature and to cool rapidly until about 60% of worst solutions are being accepted. This forms a real T_0 that can now be cooled more slowly. A very similar idea was also suggested by Dowsland (1995). This kind of annealing schedule is analogous to physical annealing in which a metal is heated until it becomes liquid and then cooling begins. That is, once the metal is transformed into liquid it is pointless carrying on heating it.

An obvious stopping criterion is that we should stop executing simulated annealing when T_0 becomes zero. This choice may require a long time for obtaining a optimal or a near optimal solution of a given problem. However, in practice, it is not necessary to let T_0 becomes zero because as it approaches zero, the chances of accepting worse solutions are almost same as T_0 being equal to zero. That is, we can stop executing simulated annealing when T_0 becomes low. The magnitude of the low value can be difficult to determine and is problem-dependent. In practice, the stopping criterion is determined by fixing the number of temperature values to be used for solving a given problem, or the total number of solutions to be generated by simulated annealing. Alternatively, we can stop executing simulated annealing when no better solution is found in all trials of any temperature.

Once we have T_0 and the stopping criterion we can devise a temperature decrementing procedure. The most important consideration in such procedure is that temperature should be decremented very slowly. This consideration leads to the following temperature decrement procedure

$$T(t) = \left(\frac{T_1}{T_0}\right)^t T_0 = \alpha^t T_0 \quad (1.14)$$

where t , “time” is the step count and α is a constant close to, but smaller than, 1. This temperature decrement procedure was first proposed by Kirkpatrick *et al.* (1982) with $\alpha = 0.90$. A linear procedure can also be used in decrementing temperature very slowly. This can be expressed as

$$T(t) = T_0 - \beta t \quad (1.15)$$

where β is a constant and can be any value greater than zero. Of special theoretical importance is a logarithmic temperature introduced by Geman and Geman [7],

$$T(t) = \frac{T_0}{\log(d+t)} \quad (1.16)$$

where d is usually set equal to one. It has been proven that for T_0 being greater than or equal to the largest energy barrier of a given problem, the logarithmic temperature decrement procedure will lead a system to the global minimum state in the limit of infinite time. However, due to its asymptotically extremely slow temperature decrease, this procedure is impractical for solving many real-world problems. Szu and Hartley (1987) proposed a temperature decrement procedure

$$T(t) = \frac{T_0}{(1+t)} \quad (1.17)$$

that is inversely linear in time.

Algorithm 1 Simulated Annealing(\mathbf{x}_0, T_0, n, m)

```

 $T \leftarrow T_0$ 
 $\mathbf{x} \leftarrow \mathbf{x}_0$ 
 $\mathbf{x}^* \leftarrow \mathbf{x}$ 
for  $k = 1$  to  $n$  do
  5:   for  $j = 1$  to  $m$  do
     $\tilde{\mathbf{x}} \leftarrow \text{Neighbor}(\mathbf{x})$ 
     $\Delta E \leftarrow E(\tilde{\mathbf{x}}) - E(\mathbf{x})$ 
     $p \leftarrow \text{Min}\{\exp(-\Delta E/T), 1\}$ 
     $u \leftarrow \text{Uniform}(0, 1)$ 
    10:  if  $u < p$  then
       $\mathbf{x} \leftarrow \tilde{\mathbf{x}}$ 
      end if
      if  $E(\mathbf{x}) < E(\mathbf{x}^*)$  then
         $\mathbf{x}^* \leftarrow \mathbf{x}$ 
    15:  end if
    end for
     $T \leftarrow T_0 / (k + 1)$ 
  end for
  return  $\mathbf{x}^*$ 

```

1.1.5 Simulated annealing for training neural network

As already described in the previous chapter, the learning objective of the neural networks is to find a suitable connection weight values so as to the error function gets minimized. However, such objective poses a challenge due to the irregular surface characteristics, i. e., lots of local minima, of the error function. Classical algorithms like the gradient descent and the conjugate gradient do not guarantee of finding a global minimum. On the other hand, an alternative learning algorithm like the simulated annealing guarantees for a global minimum under some conditions. We are now going to discuss how a neural network can be trained with the simulated annealing algorithm.

Recall the analogy between function optimization and the physical crystallization process by heating at high temperature followed by slow cooling process. In the physical system, one is interested to drive the system to a microscopic configuration of the atoms for which the system has globally minimum energy. In fact, such condition makes the atoms very organized and orderly. A neural network resembles such a physical system with its connection weights as a microscopic configuration and the error function as an energy function. Our goal is to find an optimal connection weight so that the network gives minimum error.

We consider here a feed forward three layer neural network - one input layer, a hidden layer, and an output layer. At the beginning, the network will be at a random state (analogous to a random microscopic configuration). Generally it is realized by initializing the connection weights randomly within a small range. It will be easy to handle the connection weights if all the weights are arranged in a vector. If the network has a total of n weights including the bias connections simulated annealing treats it as a vector in the R^n space. Once we arrange the connection weights in a vector we have to keep track of which element of the vector denotes which connection. MATLAB provides a pair of commands, `getx` and `setx`, to switch between the two representations - a vector and a structure of connection weights. After the initialization we simply follow the simulated algorithm to train the neural network. A pseudocode for simulated algorithm is presented in Figure x.

The procedure starts with an initial temperature and an initial weight configuration. Since the temperature controls the amount of random per-

turbation of a configuration and also the acceptance probability of a bad configuration with higher energy, initial temperature should be high enough to let the network cross the barriers of local minima.

The algorithm then progresses by slowly decreasing the temperature and at each temperature stage by taking a random walk over the space R_n (i. e., the space of all configurations). A random walk consists of a sequence of steps, where each step results in a neighbor state. This random walk is also known as sampling. Simulated annealing controls the neighborhood with a distribution function having a scaling parameter proportional to the current temperature. A step is always accepted whenever it leads to a lower energy than that of the current state. An important feature of this algorithm is that it also accepts a bad step resulting higher energy with some probability. This acceptance probability is given by when a step causes higher energy, i. e., when E is positive. Acceptance probability decreases with the decrease of temperature. As the temperature T approaches to zero acceptance probability also approaches to zero. As a result, simulated annealing accepts more energy rising steps at its earlier stages than the later stages. Similar to the crystallization of solid, simulated annealing algorithm can find a global solution by the aforementioned cooling and sampling mechanism.

We can do both the sampling and cooling in different ways. Depending on the scheme, we have different simulated annealing algorithms. In the earlier section, we explained three major variants - classical, fast and very fast simulated annealing algorithms. To give an idea of how simulated annealing can be programmed for training a neural network, a complete MATLAB code of fast simulated annealing is given in Fig. X.

1.2 Genetic Algorithms

Mimicking biological evolution and harnessing its power for adaptation are problems that have intrigued computer scientists for several decades. Genetic algorithms are a family of computational models based on the underlying genetic process in biological organisms. These algorithms process a population of simple *chromosomes*-like data structures with three operators: *selection*, *crossover*, and *mutation*. The current framework of genetic algorithms was first proposed by John Holland in the late 1960s and early 1970s, although some of the ideas appeared as early as 1957 in the context

of simulating genetic systems (Fraser, 1957).

In his book *Adaptation in Natural and Artificial Systems* (Holland, 1992), Holland presented genetic algorithms in a general theoretical framework for adaptation in nature. Holland attempted to understand and link diverse natural phenomena, but he also proposed potential engineering applications of such phenomena. Since the publication of Holland's book, genetic algorithms have gained recognition as a general problem solving techniques in many applications including function optimization, image processing, system control and many other areas. In this section, we first describe genetic algorithms and then their application to training neural networks.

1.2.1 Basic Description

We have already seen that the training of neural networks can be considered as a function optimization problem. We here describe genetic algorithms in the context of function optimization where we like to optimize a set of variables either to maximize some profit or to minimize some cost. In more traditional terms, we wish to maximize (or minimize) some function

$$\text{maximize } f(\mathbf{x}) \quad (1.18)$$

$$\text{subject to } \mathbf{x} \in \Omega \quad (1.19)$$

Given a specific problem to be solved, the first step of genetic algorithms is to choose randomly a set of potential solutions to that problem, encoded them in some fashion on simple chromosomes, which represent search space solutions and evolve over time through a process of *selection* and controlled variations introduced by two evolutionary operators: *crossover* and *mutation*. The number of chromosomes n in the population is usually chosen at random and is kept fixed throughout the solution process. We call the initial set of chromosomes as the *initial population* and denote it by $P(0)$. We then evaluate each chromosome of $P(0)$ by a predefined evaluation function, sometimes called *fitness function*. Some chromosomes of $P(0)$ may get higher fitness scores and can be regarded as good chromosomes, while others can be regarded as bad chromosomes. Of course, we like to proceed towards the solution of the problem with the good chromosomes, but not with the bad chromosomes. Genetic algorithms, therefore, creates an *mating pool* or *intermediate population* $M(0)$ by probabilistically selecting the

chromosomes of $P(0)$ based on their fitness scores. Since $M(0)$ must have the same number of chromosomes as $P(0)$, the probabilistic selection may choose some good chromosomes of $P(0)$ multiple times and omit some bad chromosomes.

We finally create a new population $P(1)$ from $M(0)$ by applying crossover and mutation operators. We first apply crossover with some probability p_c on the chromosomes of $M(0)$, and then mutation with some probability p_m . The purpose of the crossover and mutation is to introduce some sort variations in the chromosomes of $M(0)$. We repeat the above procedure iteratively, generating populations $P(2), P(3), \dots$, until a desired solution is found or an appropriate stopping criterion is reached. The process of generating $P(k)$ from $P(0)$ is called an *evolutionary process*. We call one iteration of the evolutionary process as one *generation* and denote it by g . The pseudo code of genetic algorithms is given in *Algorithm*. In short, we can think genetic algorithms as a population-based model that uses selection, crossover and mutation operators to generate new sample points in a search space. The five basic components of genetic algorithms are chromosome and encoding, fitness function, selection and mating pool, crossover, and mutation. We discuss these components at some length in the following subsections.

1.2.2 Chromosome and Encoding

In genetic algorithms, the term chromosome typically refers to a candidate solution of a given problem, often encoded as a string of symbols. This is equivalent to mapping points in the space of Ω on to a string of symbols. The string length L for all chromosomes in $P(g)$ is same and is kept throughout the whole evolutionary process. A chromosome consists of elements from a chosen set of symbols, called the *alphabet*. As for example, a common alphabet is the set $\{0, 1\}$ in which case the chromosome is simply a binary string. The choice of chromosome length, alphabet, and encoding is called the *representation scheme* for the problem. Some terminologies and their synonyms are used widely in genetic algorithms and evolutionary community in general. On the side of the original problem context, candidate solution, phenotype, and individual are used to denote points in the space of Ω . This space is commonly called the *phenotype space*. On the

side of genetic algorithms, genotype, chromosome, and again individual are used for points in the encoded space where evolutionary search will actually take place through crossover and mutation operations. The encoded space is commonly called the *genotype space* (**Give a phenotype and genotype conversion figure here**).

Assume our $f(\mathbf{x})$ is consisted of two variables X_1 and X_2 , and the value of X_i lies between 0 and 31. If we use a binary string to represent a chromosome, five bits are necessary to encode each of X_i into the chromosome. The length of the chromosome, L , would be 10 bits. It is obvious that the representation of the chromosome will be crucial to the success and efficiency of genetic algorithms. A good representation will make a problem easier to solve, while a poor one will do the opposite. For instance, let we use six bits to represent X_i though five bits are sufficient. The chromosome length will increase in such a representation, eventually the search space size. Let M is the number of alphabets, L is the length of a chromosome, and N is the number of chromosomes in $P(g)$. The search space size involving $P(g)$ would be NM^L . Let X_i lies in the continuum \mathbb{R} but we encode it as an integer variable. This representation is also problematic in the sense that genetic algorithms would not able to find an appropriate value for X_i . In general, an important issue faced by genetic algorithms is the same that faced by many artificial intelligence approaches, that is, representation.

Genetic algorithms traditionally use the binary representation, but they can also use non-binary representations that are gaining popularity in recent years. When X_i lies in the continuum \mathbb{R} , it would seem natural to represent the variable into a chromosome directly as a real number. A vector of real numbers then constitute the chromosome. Since $f(\mathbf{x})$ is consisted of two variables, a real-value vector (30.5, 22.2) could be an example of the chromosome. It is clear that the goal of a representation scheme is to map the phenotype space into the genotype space. Identification of an appropriate representation scheme is the first step for solving problems using genetic algorithms. Once a suitable representation scheme has been found, the next phase is to create the initial population $P(0)$. The chromosomes of $P(0)$ are usually initialized in an entirely random fashion because we may not have any idea as to what constitutes a good initial set of potential solutions. If users have some idea about the potential solutions of a given problem, they can use it in creating $P(0)$.

1.2.3 Fitness Function

A fitness function is one of important components of genetic algorithms as it determines whether a given chromosome will contribute to future generations through reproduction. This function assigns a quality measure for each genotype of a population $P(g)$. An ideal fitness function correlates closely with an algorithm's goal, and yet may be computed quickly. The *Speed of execution* is very important, because genetic algorithms must iterate many, many times to produce good solutions for non-trivial problems.

Typically, a fitness function is composed from a quality measure in the phenotype space although the function measures the quality of a point (chromosome) in the genotype space. Consider we like to maximize $f(\mathbf{x}) = X_1^3 + X_2^3$, where X_1 and X_2 are integer variables whose value lies between 0 and 31. We can use five bits to represent each of $X_i, i \in 1, 2$. Let one chromosome of $P(k)$ is 00101 11000. The fitness score of this chromosome can be defined as the cube of its corresponding phenotype, that is, $5^3 + 24^3 = 13,949$.

1.2.4 Selection and Mating Pool

Selection is a operator in some sense a procedure that genetic algorithms use to construct a mating pool $M(g)$ by selecting copies of chromosomes from $P(g)$. To emulate natural selection, that is, survival of the fittest, it is obvious to select chromosomes based on their fitness scores. The chromosome with a higher fitness score usually have a greater chance of contributing more copies in $M(k)$, while the one with a lower fitness score has a less chance to contribute. The motivation is that highly fit chromosomes of a population possess good properties that, recombined in the right way, could lead to even better chromosomes.

Genetic algorithms traditionally use a fitness proportionate selection scheme in constructing $M(k)$. This scheme consists of two steps: the calculation of selection probability and the application of a simpling procedure. Assume our population $P(k)$ is consisted of N chromosomes, C_1, \dots, C_N and the fitness of a chromosome C_i is f_i . The average fitness, \bar{f} , over all chromosomes in $P(k)$ is

$$\bar{f} = \frac{1}{N} \sum_i^N f_i \quad (1.20)$$

In fitness proportionate selection, the probability of selecting a chromosome to be a member of $M(k)$ is proportional to the relative fitness score. That is, the probability of C_i , $p_s(C_i)$, to be in $M(k)$ is

$$p_s(C_i) = \frac{f_i}{\sum_i^N f_i} = \frac{f_i}{\bar{f}N} \quad (1.21)$$

It can be seen from Eq.(1.21) that chromosomes with a above-average fitness tend to have a better chance than those with a below-average fitness to be chosen for $M(g)$. Nevertheless, chromosomes with a below-average fitness are given a small, but a positive chance. Otherwise the search process could become too greedy resulting in stuck at local optima.

After computing the selection probability for all chromosomes in $P(g)$, we apply a sampling procedure to select chromosomes based on such probabilities. A widely used classical sampling procedure is stochastic sampling with replacement (Holland, 1975; Goldberg, 1989a). This sampling procedure maps chromosomes of $P(g)$ to contiguous segments of a line, one segment for each chromosome. The segment size of the chromosome is equal to its selection probability. We generate a random number and select a chromosome whose segment spans the random number. In the same fashion, we can select N chromosomes by generating N random numbers to form $M(g)$.

Example 1.1. Consider $P(k)$ is consisted of six chromosomes. We encode each chromosome by five bits, and the fitness of the chromosome is the decimal value of its binary representation. Table 1.2 shows chromosomes' representation, their fitness scores and selection probability. The mapping of these chromosomes to contiguous segments of a line is shown in **Fig.** We allocate the segment sizes to the chromosomes according their selection probabilities. It can be seen from **Fig.** that the chromosomes 1 and 6 got the smallest and largest segments, respectively.

To construct $M(k)$, we generate six numbers uniformly at random between 0.0 and 1.0. We call the generation of a random number as a trail. Assume six random numbers generated by six trails are 0.81, 0.32, 0.96,

Table 1.1: A typical population with six chromosomes, their fitness scores and selection probabilities.

Chromosome number	1	2	3	4	5	6
Chromosome representation	00011	01101	01000	00110	01010	10000
Fitness score	3	13	8	6	10	16
Selection probability	0.05	0.23	0.14	0.11	0.18	0.29

Table 1.2: A typical mating pool constructed from Table 1.2 and the fitness scores of its chromosomes.

Chromosome number	1	2	3	4	5	6
Chromosome representation	00011	01101	01000	00110	01010	10000
Fitness score	3	13	8	6	10	16
Selection probability	0.05	0.23	0.14	0.11	0.18	0.29

0.01, 0.65, and 0.42. **Figure** shows the position of these numbers in the line segments by their corresponding trail numbers. We have now understood that the chromosome 2 will have two copies in $M(k)$, and chromosomes 2, 3, 4 and 5 will have one copy of each in $M(k)$. Table 1.3 shows chromosomes in $M(k)$ and their corresponding fitness scores.

Table 1.3: Chromosomes of a population consists, their fitness and selection probability

Chromosome No.	1	2	3	4	5	6
Chromosome	0011	1101	1000	0110	0100	0001

The aforementioned sampling procedure is stochastic in nature and analogous to a game of chance popular in casinos and gambling, named after the French word *roulette* means “small wheel”. The analogy to a roulette can be envisaged by imagining a small wheel in which each chromosome represents a pocket on the wheel; the size of the pocket is proportional to the chromosome’s selection probability. Figure shows a roulette-wheel with pockets sized according to the probability of selection given in Table 1.2. By

spinning the roulette wheel N times, we can get N copies of chromosomes to construct $M(k)$. This is analogous to select N copies of chromosomes by generating N random numbers. This is why the stochastic sampling procedure with replacement is sometimes called the *roulette-wheel selection*. Although this sampling procedure provides a zero bias in selecting chromosomes, but does not guarantee a minimum spread.

One sampling procedure that provides zero bias and guarantees a minimum spread is the *stochastic universal sampling*. In stochastic universal sampling, the expected number of copies, $N(C_i)$ of a chromosome C_i to be in $M(k)$ is

$$N(C_i) = p_s(C_i).N = \frac{f_i}{\bar{f}} \quad (1.22)$$

The stochastic universal sampling procedure guarantees that the minimum and maximum number of copies of any chromosome are bounded by the floor and by ceiling of f_i/\bar{f} , respectively. As a result, the chromosomes with a above-average fitness ($f_i > \bar{f}$) tend to have more than one copies in $M(k)$, while they with a below-average fitness $f_i < \bar{f}$ tend to have no copies.

The stochastic universal sampling procedure maps the chromosomes of $P(g)$ to contiguous segments of a line like the stochastic sampling procedure. However, we here place N pointers that are equally spaced for selecting N chromosomes to construct $M(k)$. Since all pointers are equally spaced, the distance between any two pointers is $1/N$. This implies that we can place all pointers over the line if we know the position of the first pointer. In stochastic universal sampling, the position of the first pointer is a random number generated between 0 and $1/N$. The value of $N(C_i)$ is equal to the number of pointers that fall into the C_i 's segment of the line. The graphical representation of the stochastic universal sampling is shown in **Fig.** We can think the stochastic universal sampling as a N -armed hand that is spun just once for selecting N chromosomes instead of single-armed hand of the stochastic sampling that is spun N times. This is the basic and only difference between these two sampling procedures.

1.2.5 Crossover

Crossover is a genetic operator that intends to simulate “sexual reproduction” of biological organisms in a simple way. This operator introduces variation in $M(k)$ by exchanging information between its chromosomes. The chromosomes that exchange information are called the *parents* and the new chromosomes that are produced due to exchange of information are called the *children* or *offspring*. The essence of crossover is the inheritance of information from parents to offspring, with the possibility that good parents may produce better offspring. Generally, crossover exchanges information between two parents, but multiple parents may be useful in some cases and produces two offspring, but, again, other numbers might be appropriate in some cases.

We first choose a pair of chromosomes from $M(K)$ randomly. These algorithms then generate a random number between 0 and 1. If this number is greater than or equal to the *crossover probability or crossover rate*, p_c , the chosen chromosomes exchange information in a specific way and produce two offspring chromosomes. We apply crossover to each pair of chromosomes in $M(g)$ with probability p_c . After the crossover operation, we replace the parent (chosen) chromosomes in $M(g)$ by their offspring offspring. The $M(g)$ has therefore been modified, but still maintained the same number of chromosomes. Since crossover plays a central role in genetic algorithms, P_c is set high, usually in the range of 0.6 to 0.95 (De Jong 1975; Schraudolph and Belew, 1992). In fact, crossover is considered to be one of genetic algorithms’ defining characteristics, and it is one of the components to be borne in mind to improve the algorithms’ behavior (Liepins et al., 1992).

There are many types of possible crossover operations that we can apply on two chosen chromosomes, say X and Y . The simplest one is the *one-point crossover* when we encode the chromosomes as a string of bits. Let X and Y can produce offspring according to p_c . What is the way to produce offspring by exchanging information between X and Y ? In one-point crossover, we first generate a number, say r , between 1 and $L - 1$ uniformly at random, where L is the length of X and Y . We refer to this number as the *crossing site*. We then produce two offspring by exchanging information between X and Y according to r and L . The first offspring consists of the first r bits of X and the last $L - r$ bits of the Y . The second offspring consists of the first

r bits of Y and the last $L - r$ bits of the X . We can generalize single-point crossover to k -point crossover, where $k > 0$. Given X and Y of length L . We generate k numbers, r_1, r_2, \dots, r_k , between 1 and $L - 1$ uniformly at random without repetition. We then produce offspring by exchanging information between X and Y through the crossing sites r_1, r_2, \dots, r_k .

De. Jong, (1975). *An analysis of the behavior of a class of genetic adaptive systems.* PhD thesis, University of Michigan, Ann Arbor
N. N. Schraudolph and R. K. Belew (1992). “Dynamic parameter encoding for Genetic Algorithms,” *Machine Learning*, Vol. 9, pp. 9-21.

Example 1.1. Suppose the length $L = 8$. Consider the pair of parents X and Y is 00110000 and 11111111. Suppose the crossing site r is 4. Then, the crossover operation applied to the above parent chromosomes yields the two offspring 00111111 and 00111111. This operation is represented graphically in **Fig.**

Example 1.2. Suppose we have now two crossing sites at 3 and 7. Then, the crossover operation applied to the parents X and Y used in the previous example yields the two offspring 00011110 and 111000011. This operation is graphically represented in **Fig.**

We usually apply k -point crossover on the binary coded chromosomes. However, we apply a different crossover when chromosomes are encoded as real-valued vectors and use the term “recombination” instead of crossover. A number of recombination operators have been proposed for genetic algorithms. Most recombination operators produce the genes of offspring via some form of combination of the parents’ genes. Let us assume that $\mathbf{x} = (x_1, x_2, \dots, x_n)$ and $\mathbf{y} = (y_1, y_2, \dots, y_n)$ are two real-coded chromosomes selected for recombination, where $x_i, y_i \in (a, b) \subset \mathbb{R}$. We can produce an offspring $\mathbf{z} = (z_1, z_2, \dots, z_n)$ as follows

$$z_i = x_i \cdot a_i + y_i \cdot (1 - a_i) \quad i \in 1, 2, \dots, n \quad (1.23)$$

where a_i is a scaling factor chosen uniformly at random over an interval $[-d, 1 + d]$ for each z_i anew. The parameter d is specified by the user and defines the range of z_i . When the value of d is set to 0, z_i has the same range as the range of x_i and y_i . One problem with this value of d is that the range

for z_i may shrink over the generations because z_i may not be generated on the border of its range. This effect can be prevented by choosing a larger value for d . A value of $d = 0.25$ may ensure that the range of z_i is same as x_i and y_i . This kind of recombination is called sometimes *intermediate recombination*.

Example 1.3. Suppose we have chosen two real-coded chromosomes $\mathbf{x} = 123.5, 10.0, 22.5$ and $\mathbf{y} = 11.2, 27.0, 54.6$ for recombination. The value of d is set to 0.25. Since \mathbf{x} and \mathbf{y} are consisted of three variables, the offspring \mathbf{z} will be consisted of three variables. Thus we need to generate three random numbers uniformly at random over an interval $[-0.25, 1.25]$ for the scaling factor a used in Eq.(1.23). Let these number are 0.2, -0.3, and 0.7. By Eq.(1.23), we get $z_1 =$, $z_2 =$ and $z_3 =$. The offspring Z is.

1.2.6 Mutation Operator

Mutation is also a variation operator applied to a single chromosome not to multiple chromosomes like crossover. Genetic algorithms use mutation after crossover on the chromosomes of $M(g)$. Two parameters involved in mutation: one is the *mutation probability* or *mutation rate*, p_m and one is the mutation size or the step size, δ . Mutation modifies randomly the alphabets (variables) of a chromosome based on the p_m .

Example 1.3. Let a chromosome $\mathbf{x} = (x_1, x_2, x_3, x_4, x_5)$, and $p_m = 0.02$. We like to produce an offspring \mathbf{x}' from \mathbf{x} by mutation. The most common way of implementing mutation involves generating one random number for each variable of \mathbf{x} . This number tells whether mutation will modify a particular variable. Since \mathbf{x} is consisted of five variables, we generate five random numbers uniformly at random between 0 and 1. Let these numbers are 0.22, 0.01, 0.33, 0.51, and 0.40. Since the second number is within the range of p_m , mutation will modify x_2 . Let mutation modifies x_2 to x'_2 , Thus $\mathbf{x}' = (x_1, x_2', x_3, x_4, x_5)$ is the offspring.

We use a different mutation for chromosomes with a different encoding. Mutation for the binary-valued chromosome is a lot simpler. We call such mutation as *bit mutation*. It is usually a bit-flipping operation; that is, we replace each bit of a chromosome from 0 to 1 or vice versa with probability p_m . We can also implement bit mutation without using the flipping operation. For instance, we can replace each bit of a binary-coded chromosome by

0 or 1 with equal probability, that is, with $p_m = 0.5$. The mutation size, δ , of the bit mutation is always one irrespective of the way of implementation. However, for a real-valued chromosome, there is no way to use the flipping or replacing operation in mutation and δ may not be one. What we can do is that we can add some real number based on δ , say Δx_i , with p_m to each variable x_i of a real-coded chromosome, \mathbf{x} . The most common way of getting Δx_i is the use of a distribution function, for instance, the Gaussian or Cauchy distribution function. That is,

$$\Delta x_i = \delta_i r \quad (1.24)$$

where r is a random number generated by the Gaussian or Cauchy distribution function.

Genetic algorithms use a very small p_m , usually in the range from 0.001 to 0.01 (De Jong 1975; Schraudolph and Belew, 1992) or $1/L$. Thus only a few chromosomes will undergo a change due to mutation, and of those that are mutated, only a few of the symbols are modified. The small p_m implies that mutation plays a minor role in the evolutionary process of genetic algorithms. The purpose of mutation in genetic algorithms is to allow the algorithms to avoid local minima by preventing the population of chromosomes from becoming too similar to each other that slows or even stops evolution. Consider we have a population of binary-coded chromosomes with $L = 10$. Assume the first bit of each chromosome in the population becomes one after several generations of an evolutionary process. However, we need a solution that requires a zero in the first bit of any chromosome. Can we get such a solution by applying only crossover again and again to the chromosomes of the population? The answer is not affirmative. Since mutation can change randomly any bit of a chromosome, we may get a zero in the first bit position by applying mutation again and again. Holland (1975) argues that mutation is what prevents the loss of variability at a given bit position.

1.2.7 Training network using genetic algorithms

So far we have concentrated on the basic ideas and components of genetic algorithms. It is time to look the employment of such algorithms into neural

networks. We here consider how genetic algorithms can be employed for training neural networks to learn suitable mappings from a given data set. As in previous training methods, learning will be based on an error function, which we like to minimize by iteratively adjusting the weights and biases of a neural network.

The first step of using genetic algorithms, as we have already seen, is the creation of a population of chromosomes. The chromosomes constitute a search space on which genetic algorithms work to find a optimal or near optimal solution of a given problem. Since the objective of a neural network training is to find a suitable set of weights, a chromosome must contain information (the value) of such weights. A straightforward way of constructing the chromosome is simply the concatenation of all the network's weights in a string. Do we need to concatenate the weights in a specific way or random fashion? As an illustration of the concatenation issue, consider a three layered feed-forward neural network with three input, two hidden and two output neurons. Let we construct two chromosomes, say X and Y , by randomly concatenating the weights of this network. Since X and Y encode all the weights of the same network, we use a different value to encode a same weight into X and Y . Let a_{ij}^l and b_{ij}^l denote the value of the same weight W_{ij}^l encoded into X and Y , respectively. **Figure** shows the graphical representation of the aforementioned network and two chromosomes.

What happen if we apply the single point crossover on X and Y . Let the crossing site is 7 and crossover produces two offspring, say X' and Y' . The offspring X' consists of the first seven weights of X and the last weights of Y , while Y' consists of the first seven weights of Y and the last weight of X . The graphical representations of such offspring are shown in **Fig. Y**. It is clear that the parent networks, that is, the networks with the weights encoded in X and Y , will have a very different functionality compared to the offspring networks, that is, the networks with the weights encoded in X' and Y' , although crossover exchanges only one weight. As for example, a_{22}^2 is the value of the weight variable W_{22}^2 connecting the hidden neuron 2 to the output neuron 2. Since the network shown in **Fig.** can be used for two-class problems, a_{22}^2 will help to turn on and off the output neuron 2 for a training example belongs to class 2 and 1, respectively. However, the crossover exchanges u_{22}^2 with v_{11}^2 , which have a opposite functionality. We have now understood that the crossover may change the functionality

of a network drastically when we form the chromosomes of a population by randomly concatenating the weights of a neural network. Besides the disruptive effect of crossover, there is another prominent problem is associated with crossover. This problem is called the *permutation* or *competing convention* problem, which we shall described in the next subsection.

We can avoid the disruptive effect of crossover by concatenating the network's weights in a specific way for all chromosomes of a population. One such way is concatenating the weights based on the processing units (hidden neurons) of the network. Let we form a chromosome in such a way that all incoming and outgoing weights of a hidden neuron are placed together closely in the coding representation of the chromosome. Specifically, for a particular hidden neuron, we first concatenate its all incoming weights then its all outgoing weights. If a network is consisted of n hidden neurons, we concatenate the weights of the first hidden neuron at beginning, then the second hidden neuron and so on. Let p and q are two chromosomes that we form as described above for our network shown in **Fig.**. The graphical representation of p and q is shown in **Fig.**. We again here consider the same crossing sit 7 for producing offspring. In this case, crossover produces two offspring, say p' and q' , by exchanging a_{22}^2 of p with b_{22}^2 of q . We can expect that when crossover exchanges one weight, that is, a_{22}^2 to b_{22}^2 (or vice versa), the parent and offspring networks will have a similar functionality. This expectation is reasonable in the sense that both a_{22}^2 and b_{22}^2 represent the value of the same weight variable W_{22}^2 connecting hidden neuron 2 to output neuron 2. So we can say that since the single point crossover operator is more likely to disrupt genes that are far apart on a chromosome than to disrupt genes located close to each other, it is useful to place functional units of a neural network tightly together on a chromosome.

After deciding the way of forming chromosomes, we need to decide a coding scheme to encode the weights. The decision is basically the choice between the binary-valued and real-valued encodings. While standard genetic algorithms use the former encoding scheme, the later one seems to be beneficial here because the weights of a neural network are real numbers in general. The binary encoding results a large string (a large sized chromosome), specifically for a neural network with many weights. Let we use c and d bits to encode the integer part and decimal part of a weight, respectively. If the network has k weights, the size of a chromosome will be

$k(c + d)$ bits. The value of k is dependent on the number of neurons and their connectivities in the network. We shall describe various methods in chapters 3-5 to determine automatically a near optimal number of neurons, specifically hidden neurons, for a given problem. The value of c and d is dependent on the weights' size that the network requires to solve the problem. However, there is no way to find the weights' size in advance although we need to do so for training the network using standard genetic algorithms. A large value for c and d increases the chromosome's size, eventually the search space size. Thus genetic algorithms will take more time in finding a suitable set of weights. On the other hand, a small value for c and d reduces the search space size. Let we choose the value of c as 5 and d as 2. The weight's value in such a setting can lie in the range of **-15** to **+15**. Genetic algorithms do not able to find optimal weights if they do not lie in the aforementioned range. We have now understood that the binary encoding not only increases the search space size but also restrict it. This bottle neck make an evolutionary process inefficient when standard genetic algorithms are applied for training neural networks. So we can directly represent the weights in the chromosome as real numbers. The chromosome in this case will be a vector of real numbers.

Once we have chosen the way to form a chromosome and the way to represent the weights in the chromosome, we can create an initial population $P(0)$ consisting of n chromosomes. We then need a fitness function to measure the quality of each chromosome in the population. Since the chromosome contains all the network's weights, we can measure the chromosome's quality in terms of the network error for the training data. As mentioned before, the goal of training the network is to minimize its training error. The training error, therefore, does not directly translate into a fitness score which have to be the larger the better the chromosome. We can easily transform the error by using $1/e$ or $1/(1 + e)$ as fitness score. We have already seen that crossover and mutation do not require any gradient information to modify the chromosomes, here the network's weights because the chromosomes encode such weights. We, therefore, can also use non-differentiable functions like the percentage of correct answers (classification accuracy) on the training set as a fitness function. However it is advantageous if the performance measure is continuous not just discrete because this allows genetic algorithms to better discriminate between the

performance of different chromosomes. Finally, we have to decide the type of selection, crossover and mutation to be used for evolution. Although we can use any selection scheme, the choice of crossover and mutation is dependent on the encoding scheme of chromosomes. As mentioned earlier, the real-coded chromosomes are suitable for training a neural network. So we can use intermediate crossover and Gaussian or Cauchy mutation.

1.2.8 Permutation Problem

The Permutation problem may arise when genetic algorithms use two representations: one for evolutionary search, specifically for crossover and one for fitness evaluation. The source of this problem is the mapping of many genotypes (chromosomes) to one phenotype (function). An encoding scheme that allows different genotypes for one phenotype is responsible for the permutation problem. One such encoding is the binary encoding. To understand this problem clearly, consider two neural networks that have the same number of input, and output neurons. We also consider the same number of hidden neurons for these neurons but the hidden neurons are shuffled. Let us use five bit to encode each weight of the networks. **Figure 2** shows the networks and their genetic representations.

Figure 2 essentially depicts a many-to-one mapping from genotype to phenotype as two networks have a different genotype representation but they are same in terms of functionality. In fact, all permutations over the set of hidden node indices, $(H1, H2, H3)$, are equivalent in terms of the network functionality. This is because, in a purely feed-forward neural network, rearranging the order of the hidden neurons has no effect on the network functionality. That is, the same network have many different genotype representations. Given a fully connected feed-forward neural network with N hidden neurons, there will be $N!$ symmetries and up to $N!$ equivalent solutions. Genetic algorithms apply search operators, crossover and mutation, on the genotype space (representation). The redundancy in the genotype space will make searching inefficient.

The permutation problem not only creates redundancy in the search space but also creates search basis towards *frequent phenotypes* and ineffectiveness in the crossover operation. Let us assume that G and S denote the genotype and phenotype space, respectively. **Igel** () defines $r(s)$, the

redundancy of a phenotype $s \in S$, as the number of genotypes that map on to s . The average redundancy, \bar{r} , can be defined as

$$\bar{r} = \frac{1}{|S|} \sum_{s \in S} r(s) = \frac{|G|}{|S|} \quad (1.25)$$

A phenotype can be considered as the *frequent phenotype* or the *rare phenotype* if and only if $r(s) > \bar{r}$ or $r(s) < \bar{r}$. We can now define fraction of frequent phenotype as the ratio of the number of frequent phenotypes to the total number of phenotypes in the phenotype space. Fraction of frequent genotype can be defined as the ratio of the number of genotypes that map into a frequent phenotype to the total number of genotypes in the genotype space. Frequent phenotypes arise only when an encoding scheme suffers from permutation problem where different genotypes map into one phenotype introduces the redundancy of phenotypes to appear. A network with a increasing number of hidden neuron nodes the fraction of frequent genotypes increases and fraction of frequent phenotypes decreases. Therefore, an unbiased search operator in the genotype space is heavily biased in the phenotype space and the bias is specifically towards frequent phenotypes.

Effectiveness genetic algorithms greatly depend on the efficacy of crossover. We usually apply crossover between fitter parents to produce offspring by exchanging values of their lined up genes. This lining up is particularly of the foremost importance for genetic algorithms because without identifying and lining up the building blocks (genes) crossover will be meaningless and may result weaker offspring. Thus, the genetic process may eventually turn into a random walk. A proper lining up of the genes can be verified by applying crossover on two parents that have different genotypes but have a identical phenotype. Consider two such parents which are shown in (Fig.). What happen if we apply crossover in the dotted line position of the parents (Fig.). The offspring duplicate some elements of the parents and omit others.

two identical phenotypes. If the combination of encoding and recombination operator can identify which genes line up with which, the operation should result into the same individual. In case of different parent phenotypes, this effective lining up will result in values exchanges of comparable building blocks.

Due to permutation problem, this lining up of phenotypes becomes a strenuous task. In many cases, it is probably not possible. Schemes that cannot assist such lining up will result into futile recombination.

Example 1.4.

1.2.9 Evolutionary Programming

Evolutionary programming is also a population based computation model. It was first proposed by Lawrence J. Fogel in the mid 1960 as one way to achieve artificial intelligence through simulated evolution. In his work, Fogel applied evolutionary programming to the evolution of finite state machines. Mutation operators were used to create new solutions (finite state machines) that were being evolved for specific tasks. Evolutionary programming was dormant for many years. Later, in the late 80's, David B. Fogel reintroduced it to solve more general tasks. The pseudo code of evolutionary programming is given in *Algorithm*.

There are a couple of conceptual ideas associated with evolutionary programming. First, evolutionary programming works on the phenotype space unlike genetic algorithms that work on the genotype space. A philosophical tenant of evolutionary programming is that a search operator should act as directly as possible on the phenotype space to change the behavior of a system. Thus evolutionary programming uses a representation scheme typically tailored to the problem domain for encoding information in a chromosome. Second, evolutionary programming does not support exchanging information between chromosomes of a population. This computation model, therefore, does not apply crossover or recombination on the chromosomes of a population to produce offspring, but rather applies only mutation. This is the basic difference between evolutionary programming and genetic algorithms. Crossover is considered unnecessary in evolutionary programming and evolutionary computation in general!

Any kind of mutation can be used.

Let $P(k)$ is consisted of N chromosomes. Evolutionary programming applies mutation on each chromosome of $P(k)$ and produces N offspring. We can use bit **or** mutation for the binary valued chromosomes, while Gaussian or Cauchy mutation for the real valued chromosomes. As seen from **Fig. ,** the choice of standard deviation is very important in Gaussian distribution.

Evolutionary programming, therefore, puts emphasis on the optimal value of standard deviation when it uses the Gaussian mutation. This emphasis is reasonable because mutation is the primary and only search operator in evolutionary programming. Do we use the same optimal standard deviation for all problems? Unfortunately, the optimal standard deviation is problem dependent as well as dimension dependent. Even for the same problem the optimal standard deviation is not same in the whole evolutionary process. A large standard deviation is suitable at the beginning of an evolutionary process because it allows a large jump that is beneficial for exploring a search space. On the other hand, a small standard deviation is suitable at the end of the process to facilitate exploitation (fine tuning). How can we get an optimal standard deviation at different stages of the evolutionary process? One way to achieve it is to evolve the standard deviation along with the chromosomes of $P(k)$. Evolutionary programming does the same thing exactly. It includes the standard deviation as part of an chromosome so that it can evolve automatically. Let us like to minimize $f(\mathbf{x})$, $\mathbf{x} \in \mathbb{R}^d$. The representation of a population of chromosomes to solve $f(\mathbf{x})$ is (\mathbf{x}_i, σ_i) , $\forall i \in \{1, \dots, N\}$. In many implementations of evolutionary programming, σ_i is mutated first and then \mathbf{x}_i using the new σ'_i . That is, for each parent (\mathbf{x}_i, σ_i) , $i = 1, \dots, N$, the Gaussian mutation produces a single offspring $(\mathbf{x}'_i, \sigma'_i)$ by: for $j = 1, \dots, d$,

$$\sigma'_i(j) = \sigma_i(j) \exp(\tau' N(0, 1) + \tau N_j(0, 1)) \quad (1.26)$$

$$x'_i(j) = x_i(j) + \sigma'_i(j) N_j(0, 1) \quad (1.27)$$

where $x_i(j)$, $x'_i(j)$, $\sigma_i(j)$ and $\sigma'_i(j)$ denote the j -th component of the vectors \mathbf{x}_i , \mathbf{x}_i , σ_i and σ_i , respectively. $N(0, 1)$ denotes a normally distributed one-dimensional random number with mean 0 and standard deviation 1. $N_j(0, 1)$ indicates that the random number is generated anew for each value of j . The factors τ and τ' have commonly set to $(\sqrt{2\sqrt{d}})^{-1}$ and $(\sqrt{2d})^{-1}$ (Bck and Schwefel, 1993; Fogel, 1994).

Third, evolutionary programming traditionally uses the tournament selection to construct $P(k+1)$ from all parents and offspring. Of course, from an algorithmic point of view, there is no reason why evolutionary programming can not use other selection schemes. However, the aim of the selection

scheme in evolutionary programming is different from that in genetic algorithms, which uses the selection scheme to form a mating pool for crossover. In the tournament selection scheme, as the name suggests, we arrange a pairwise competition over the union of parents and offspring. For each chromosome, we choose s opponents from the union uniformly at random. The chromosome receives a “win” if its fitness is better than an opponent. Since the competition is done in pairwise, the chromosome can receive at most s wins. We select N chromosomes out of N parent chromosomes and N offspring chromosomes that have the most wins to form $P(k+1)$. The most important aspect tournament selection is the parallelization as each tournament is independent. In the best-case scenario if we have the same number of processors as twice the population size all tournaments can be run simultaneously. However, the tournament size s introduces a sampling bias into the selection process. A larger s correspond to higher probability of the most fit chromosome being selected relative to the other chromosomes. Sokolov and Whitley (2005) review analytic results and present empirical evidence that shows this bias has a significant impact on search performance.

1.3 *Hint Evolutionary Programming*

The representations used in evolutionary programming are typically tailored to the problem domain [SJBF93]. One representation commonly used is a fixed-length real-valued vector. The primary difference between evolutionary programming and the other approaches (GA, GP, and ES) is that no exchange of material between individuals in the population is made. Thus, only mutation operators are used. For real-valued vector representations, evolutionary programming is very similar to evolutionary strategies without recombination.

they differ mainly in their representations of potential solutions and their operators used to modify the solutions.

There are a couple of conceptual ideas that are closely associated with evolutionary programming. First, evolutionary programming does not use recombination and there is a general philosophical stance that recombination is unnecessary in evolutionary programming and in evolutionary computation in general! The second idea is related to the first. Evolutionary programming is viewed as working in the phenotype space whereas genetic

algorithms are seen as working in the genotype space. A philosophical tenant of evolutionary programming is that operators should act as directly as possible in the phenotype space to change the behavior of a system. Genetic algorithms, on the other hand, make changes to some encoding of a problem must be decoded and operationalized in order for behaviors to be observed and evaluated. Sometimes this (partially philosophical) distinction is clear in practice and sometimes it is not.

The term evolutionary programming dates back to early work in the 1960's by L. Fogel [16]. In this work, evolutionary methods were applied to the evolution of finite state machines. Mutation operators were used to alternate finite state machines that were being evolved for specific tasks. Evolutionary programming was dormant for many years, but the term was resurrected in the early 1990s. The new evolutionary programming, as reintroduced by D. Fogel, [15, 14, 13], is for all practical purposes, nearly identical to an evolution strategy. Mutation is done in a fashion that is more or less identical to that used in evolution strategies. A slightly different selection process (a form of Tournament Selection) is used than that normally used with evolution strategies, but this difference is not critical. Given that evolution strategies go back to the 1970's and predate the modern evolutionary programming methods by approximately 20 years, there appears to be no reason to see evolutionary programming as anything other than a minor variation on the well-established evolution strategy paradigm. Historically, however, evolution strategies were not well known outside of Germany until the early 1990's and evolutionary programming has now been widely promoted as one branch of Evolutionary Computation.

There are a couple of conceptual ideas that are closely associated with evolutionary programming. First, evolutionary programming does not use recombination and there is a general philosophical stance that recombination is unnecessary in evolutionary programming and in evolutionary computation in general! The second idea is related to the first. Evolutionary programming is viewed as working in the phenotype space whereas genetic algorithms are seen as working in the genotype space. A philosophical tenant of evolutionary programming is that operators should act as directly as possible in the phenotype space to change the behavior of a system. Genetic algorithms, on the other hand, make changes to some encoding of a problem must be decoded and operationalized in order for behaviors to be observed

and evaluated. Sometimes this (partially philosophical) distinction is clear in practice and sometimes it is not.

Evolutionary Programming (EP) is presented by L. J. Fogel [FOW66]. He initially studied this method to develop the artificial intelligence and succeeded in evolving a mathematical automaton that predicts a binary time series. Later, in the middle of 80s, his son David Fogel further developed it to solve more general tasks including prediction problems, optimization, and machine learning [Fog95]. Since this approach modeled organic evolution at the level of evolving species, the original EP does not rely on any kind of recombination.

The representations used in evolutionary programming are typically tailored to the problem domain [SJBF93]. One representation commonly used is a fixed-length real-valued vector. The primary difference between evolutionary programming and the other approaches (GA, GP, and ES) is that no exchange of material between individuals in the population is made. Thus, only mutation operators are used. For real-valued vector representations, evolutionary programming is very similar to evolutionary strategies without recombination.

A typical selection method is to select all the individuals in the population to be the N parents, to mutate each parent to form N offspring, and to probabilistically select, based upon fitness, N survivors from the total $2N$ individuals to form the next generation.

In general, each parent generates an offspring by the Gaussian mutation and better individuals among parents and offspring are selected as parents of the next generation. EP has been applied successfully for the optimization of the real-valued functions [FA90] [FS94] and other practical problems [SS]. For more details see [BRS93].

Evolutionary programming is one of the four major evolutionary algorithm paradigms.

It was first used by Lawrence J. Fogel in 1960 in order to use simulated evolution as a learning process aiming to generate artificial intelligence. Fogel used finite state machines as predictors and evolved them.

Currently evolutionary programming is a wide evolutionary computing dialect with no fixed structure or (representation), in contrast with some of the other dialects. It is becoming harder to distinguish from evolutionary strategies. Some of its original variants are quite similar to the later genetic

programming, except that the program structure is fixed and its numerical parameters are allowed to evolve.

Its main variation operator is mutation; members of the population are viewed as part of a specific species rather than members of the same species therefore each parent generates an offspring, using a (+) survivor selection

1.4 Hint for Genetic Algorithms

Mimicking biological evolution and harnessing its power for adaptation are problems that have intrigued computer scientists for at least four decades. Genetic algorithms (GAs), invented by John Holland in the 1960s, are the most widely used approaches to computational evolution. In his book *Adaptation in Natural and Artificial Systems* (Holland, 1992, also reviewed in this issue), Holland presented GAs in a general theoretical framework for adaptation in nature. Hollands motivation was largely scientific he was attempting to understand and link diverse types of natural phenomena but he also proposed potential engineering applications of GAs. Since the publication of Hollands book, the field of GAs has grown into a significant sub-area of artificial intelligence and machine learning. Nowadays one can find several international conferences each year as well as a number of journals devoted to GAs and other evolutionary computation approaches. Research on GAs has spread from computer science to engineering and, more recently, to fields such as molecular biology, immunology, economics, and physics. One result of this growth in interest has been a division of the field of GAs into several subspecies. One major division is between research on GAs as engineering tools and research

1.5 Hint

In Chapter 5 backpropagation was introduced. Backpropagation is a very effective means of training a neural network. However, there are some inherent flaws in the back propagation training algorithm. One of the most fundamental flaws is the tendency for the backpropagation training algorithm to fall into a local minima. A local minimum is a false optimal weight matrix that prevents the backpropagation training algorithm from seeing the true solution.

The most widely used method for supervised training of multilayer perceptrons turns out to be the back-propagation algorithm, which retrieves the desired weight matrix by optimizing a proper cost function. Unfortunately, the steepest descent technique, normally employed in the minimum search, is often slowed down by the presence of flat regions and by the high number of local minima in the surface of the considered cost function. Then, it can be convenient to adopt alternative training methods that ensure the achievement of the global optimum, such as the simulated annealing, widely used in the solution of complex problems, or the genetic algorithms, which perform the optimization process by following an approach similar to that employed in the evolution of living beings.

The different characteristics of these two global optimization techniques have been combined to obtain a new training method that maintains the reliability of simulated annealing and the efficiency of genetic algorithms, reducing at the same time their major drawbacks. The Interval Genetic Algorithm (IGA) represents a natural extension of classic genetic algorithm to the continuous gene setting [14]. Its ability of reaching the global minimum of a cost function is considerably better than that showed by simulated annealing and, in addition, the convergence speed of IGA turns out to be on the average 10 times higher.

Annealing, as mentioned, is a method used to toughen materials. Its importance is that it prevents the creation of defects in the atomic scale.

Defects like vacancies, misplacement etc. can really hurt the strength of the material. For example, in this glass factory (taken from "bullseye glass factory" homepage) it is extremely important to anneal the glass before it is ready or else it will be very fragile and even removing it from its mold will be impossible. and I quote:

"Annealing, the controlled cooling of a glass, is critical to its longevity. Glasses which are not properly annealed will contain stress which may result in breakage before or at any time subsequent to their removal from the kiln."

So how do we anneal?

Actually the process is very simple in concept. you just need to cool down over a relatively long period of time with frequent reheating. (as oppose to quenching which means rapid cooling.) the reheating is meant to prevent the defects (which are considered mathematically as a local minima)

and stabilized the material at its optimized position.

more on the math behind the process can be read at Dr Joan Adler and Amihay Silverman's simulated annealing page

And what can we do with it?

Well , besides its physical application, annealing is also used as an optimization tool. How is this made? our problem is finding the global optima of the function. the function is evaluated at each point and then and the annealing process begins. any downhill step is accepted and the process repeats from this new point. An uphill step may be accepted as it can escape from local optima. This uphill decision is made by the Metropolis criteria as described here(this is taken from the "MALLBA project" homepage) . Of course that by using this criteria we have some assumptions made on the function which must be obeyed.

usages:

glass making, ice cream making, mathematical optima problems, strength of materials etc.

a nice 1 dimensional annealing process can be viewed here . (this is taken from the "Taygeta scientific inc" page)

more about annealing can be read in the links section.

Simulated annealing is a generalization of a Monte Carlo method for examining the equations of state and frozen states of n-body systems [Metropolis et al. 1953]. The concept is based on the manner in which liquids freeze or metals recrystallize in the process of annealing. In an annealing process a melt, initially at high temperature and disordered, is slowly cooled so that the system at any time is approximately in thermodynamic equilibrium. As cooling proceeds, the system becomes more ordered and approaches a "frozen" ground state at $T=0$. Hence the process can be thought of as an adiabatic approach to the lowest energy state. If the initial temperature of the system is too low or cooling is done insufficiently slowly the system may become quenched forming defects or freezing out in metastable states (ie. trapped in a local minimum energy state). The original Metropolis scheme was that an initial state of a thermodynamic system was chosen at energy E and temperature T , holding T constant the initial configuration is perturbed and the change in energy dE is computed. If the change in energy is negative the new configuration is accepted. If the change in energy is positive it is accepted with a probability given by the Boltzmann factor

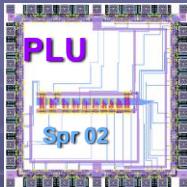
$\exp{-(dE/T)}$. This processes is then repeated sufficient times to give good sampling statistics for the current temperature, and then the temperature is decremented and the entire process repeated until a frozen state is achieved at $T=0$.

By analogy the generalization of this Monte Carlo approach to combinatorial problems is straight forward [Kirkpatrick et al. 1983, Cerny 1985]. The current state of the thermodynamic system is analogous to the current solution to the combinatorial problem, the energy equation for the thermodynamic system is analogous to at the objective function, and ground state is analogous to the global minimum. The major difficulty (art) in implementation of the algorithm is that there is no obvious analogy for the temperature T with respect to a free parameter in the combinatorial problem. Furthermore, avoidance of entrainment in local minima (quenching) is dependent on the "annealing schedule", the choice of initial temperature, how many iterations are performed at each temperature, and how much the temperature is decremented at each step as cooling proceeds.

1.6 Hint for Mathematical Notations

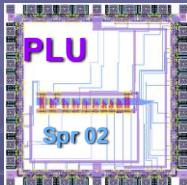
$$N = \{v \in \mathbb{R} / v * \sqrt{2} \in \mathbb{N}\}$$

$$\lim_{x \rightarrow \infty} f(x) = 0$$



Genetic Algorithms

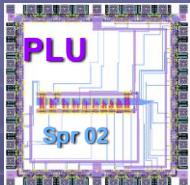
A 3D rendering of a spiral made of blue and white horizontal bands, resembling a DNA helix or a coiled ribbon. It is positioned behind the text 'Genetic Algorithms'.



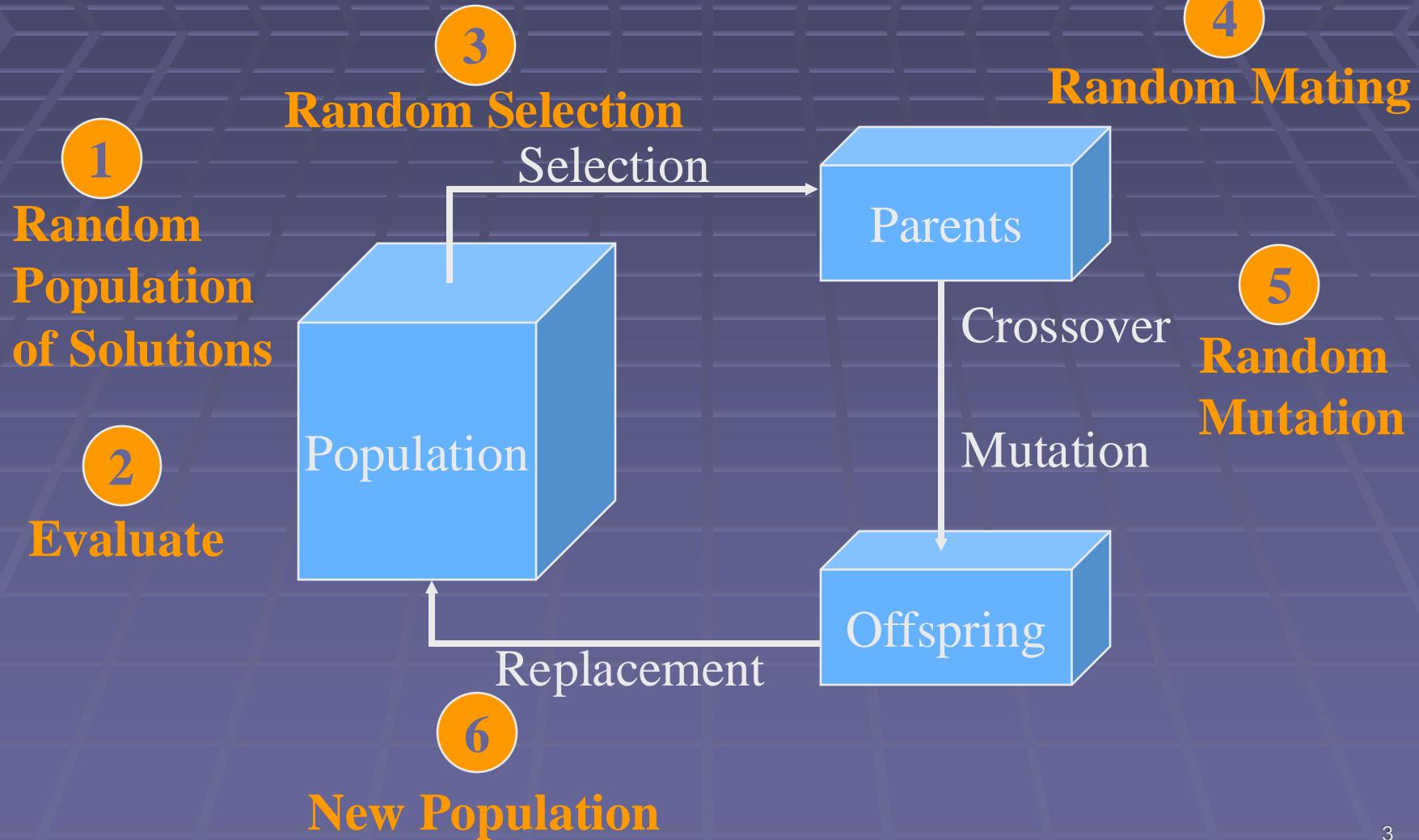
Definition

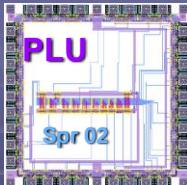
- Goldberg, 1989: “Genetic Algorithms are search algorithms based on the mechanics of natural selection and natural genetics.”
- A genetic algorithms is a directed random search procedure

A genetic algorithm borrows ideas from biology to search a solution space for a target value.

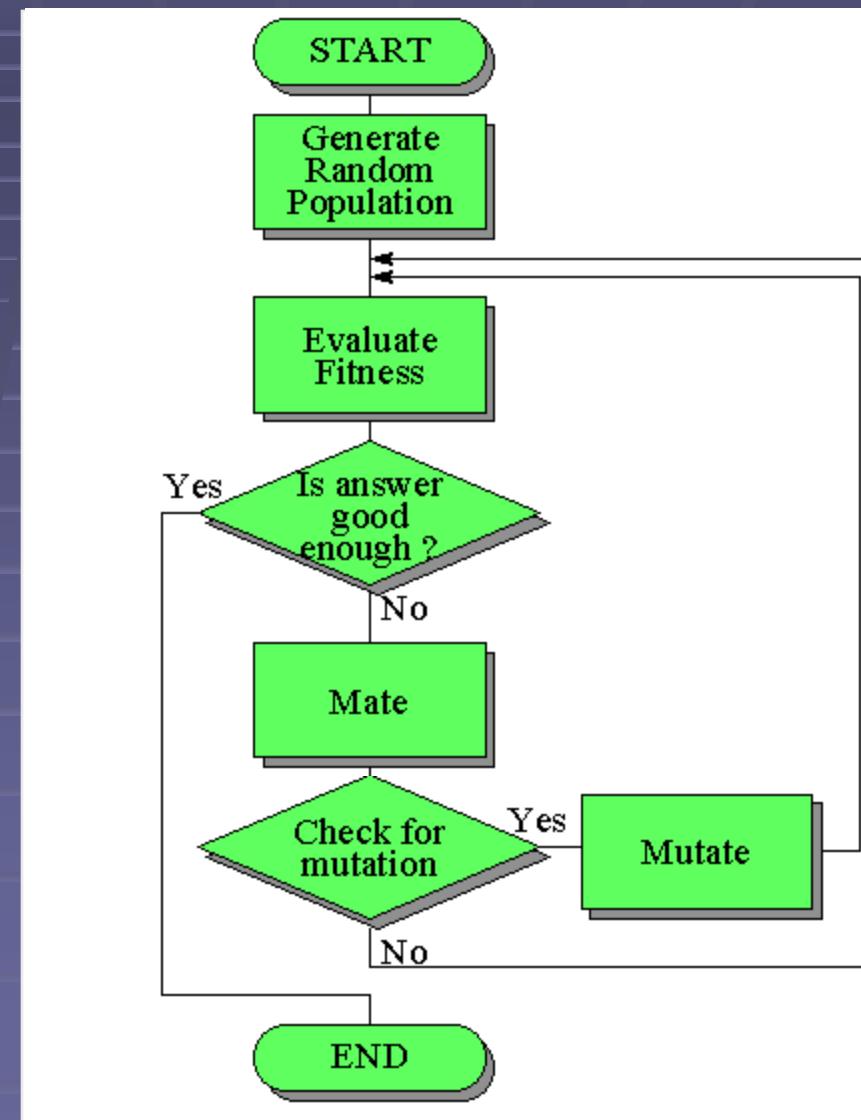


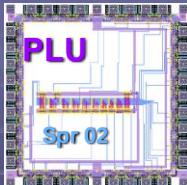
General Approach





GA Flow Chart

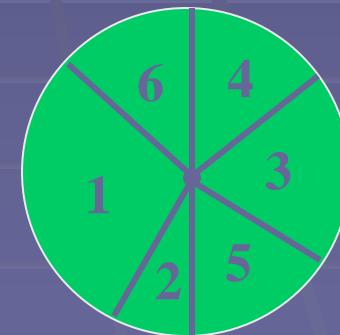




Parent Selection

- The process in which individual strings in the population are selected to contribute to the next generation is called **parent selection**
 - based on fitness
 - strings with a high fitness have a higher probability of contributing one or more offspring to the next generation
- Biased Roulette Wheel Selection

When you spin the wheel, items 1 and 5 have the greatest chance of coming up while item 2 has the smallest



Example

- Given the following population of chromosomes, select two parents:

Chromosome	Fitness	% fitness	cf
(1 0 1 0 0 1)	23	0.28	0.28
(1 1 1 0 0 1)	12	0.15	0.43
(0 1 1 0 1 1)	25	0.30	0.73
(0 1 0 1 1 0)	5	0.06	0.79
(0 1 1 0 1 0)	17	0.21	1.00
<hr/>		Total Fitness	82

Find the total fitness of the population

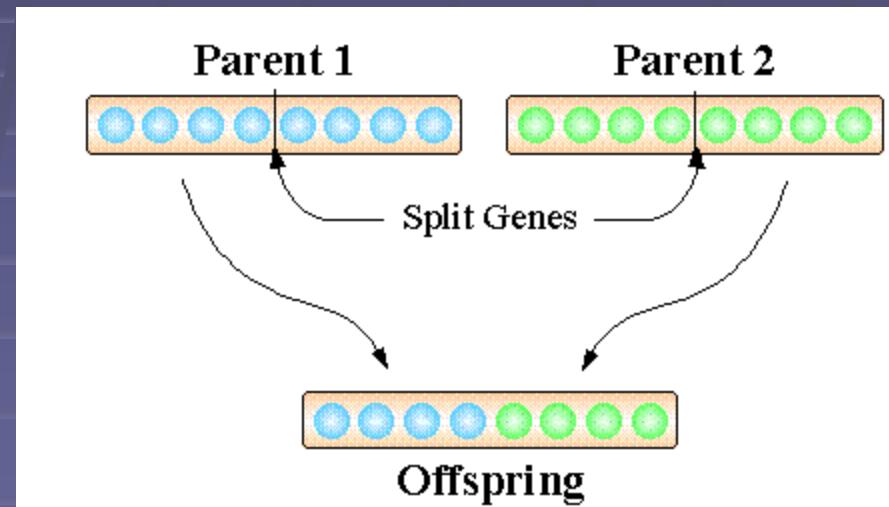
Find the %fitness of each element

Find the cumulative fitness

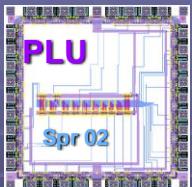
Now, throw a random number between 0 and 1, if it is in the range 0 to 0.28 select element 1, between 0.28 and 0.43 select element 2,...

Crossover

- Once two parents are selected, their chromosomes are mixed to create the children for the next generation

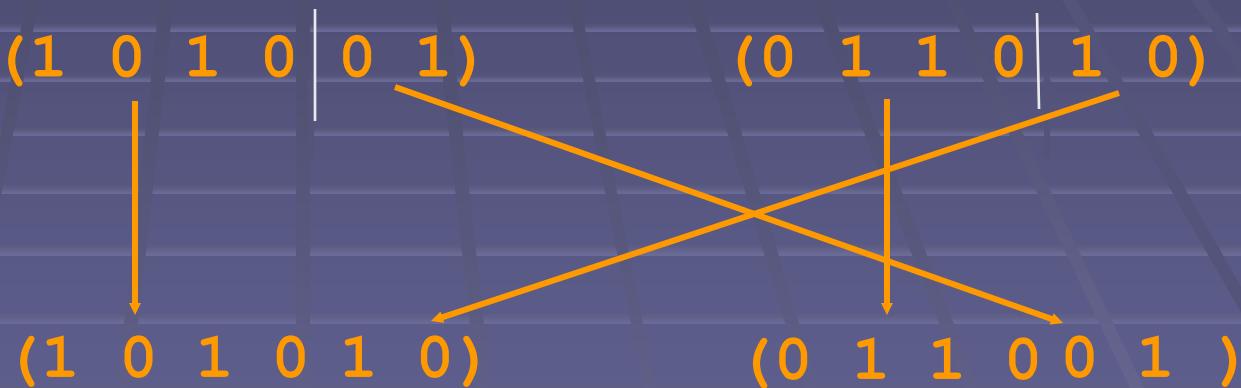


Called single point crossover



Example

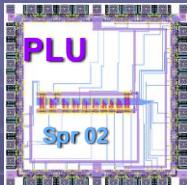
- Assume the parents selected from the previous example are: (1 0 1 0 | 0 1)



These are the two children which are now part of the next generation

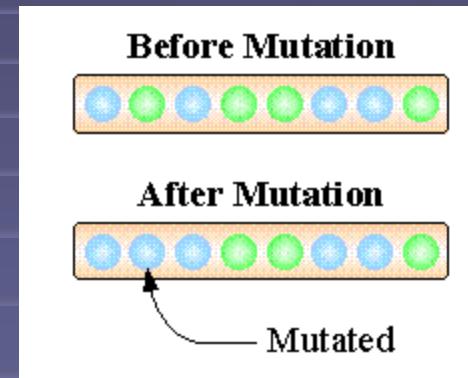
Find a random crossover point

Swap the bits after the crossover point



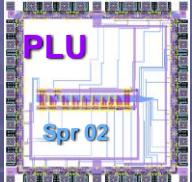
Mutation

- A bit in a child is changed (from 1 to 0 or from 0 to 1) at random



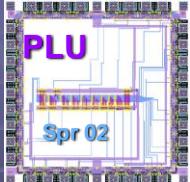
This is a small probability event

The effect is to prevent a premature convergence to a local minimum or maximum



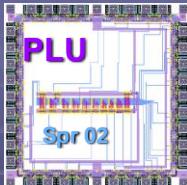
GA Performance

- Increasing diversity by genetic operators
 - mutation
 - Recombination
- Decreasing diversity by selection
 - of parents
 - of survivors



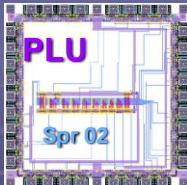
Effects of the Genetic Operators

- Using selection alone will tend to fill the population with copies of the best individual from the initial population
- Using selection and crossover will tend to cause the algorithm to converge on a good but sub-optimal solution
- Using mutation alone induces a random walk through the search space
- Using selection and mutation creates a parallel, noise-tolerant, hill climbing algorithm



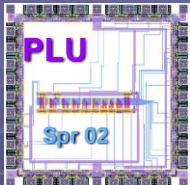
The Algorithm

- randomly initialize population(t)
- determine fitness of population(t)
- repeat
 - select parents from population(t)
 - perform crossover on parents creating population($t+1$)
 - perform mutation on population($t+1$)
 - determine fitness of population($t+1$)
- until best individual is good enough



GA Applications

- GA's can be applied to several parts of the physical design problem
 - Partitioning
 - Placement
 - Other . . .
- Scope of the Partitioning problem
 - A standard layout benchmark suite has circuits ranging from 13,000 to 200,000 nodes.
 - The number of links range from 50,000 to 800,000

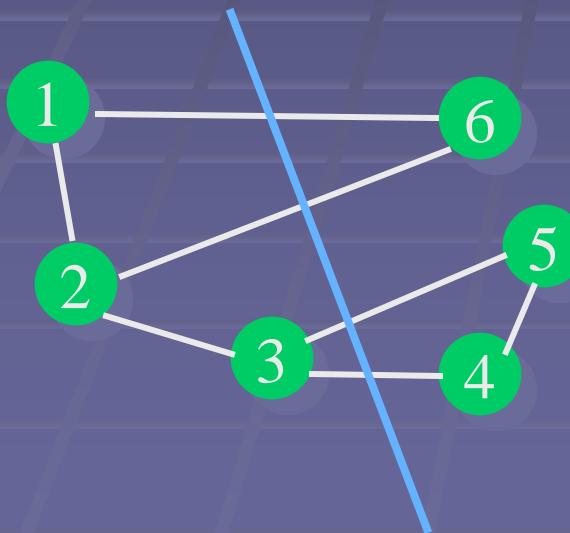


Representation

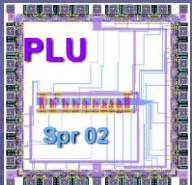
- A graph partition is represented by a binary string

Each node is represented by a bit

The 0 nodes are in one segment, the 1 nodes are in the other segment

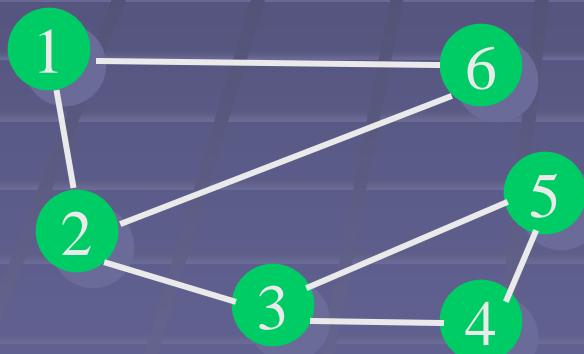


(0 0 0 1 1 1)



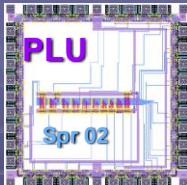
Population

- A random population of binary strings is produced



(0 0 0 1 1 1)	4
(1 0 0 1 0 1)	4
(1 0 1 1 0 0)	5
(1 0 0 1 1 0)	4

The fitness is the number of links

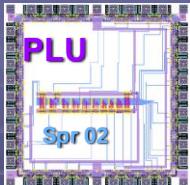


Crossover

- Parents are randomly selected (with a bias to the better fit elements)
- The parents are combined to create two children (single point crossover)

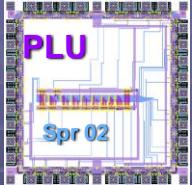
$(0 \left| 1 0 1 0 1)$
 $(1 \left| 0 0 1 0 1)$

$(1 1 0 1 0 1)$
 $(0 0 0 1 0 1)$



Mutation

- For a small number of the new population elements perform a mutation operation
 - Randomly select two nodes and swap their positions



Thank you

Chapter 5: **Adversarial Search** **&** **Game Playing**

Typical assumptions

- Two agents whose actions alternate
- Utility values for each agent are the opposite of the other
 - creates the adversarial situation
- Fully observable environments
- In game theory terms:
 - “Deterministic, turn-taking, zero-sum games of perfect information”
- Can generalize to stochastic games, multiple players, non zero-sum, etc

Search versus Games

- Search – no adversary
 - Solution is (heuristic) method for finding goal
 - Heuristics and CSP techniques can find *optimal* solution
 - Evaluation function: estimate of cost from start to goal through given node
 - Examples: path planning, scheduling activities
- Games – adversary
 - Solution is strategy (strategy specifies move for every possible opponent reply).
 - Time limits force an *approximate* solution
 - Evaluation function: evaluate “goodness” of game position
 - Examples: chess, checkers, Othello, backgammon

Types of Games

	deterministic	chance
perfect information	chess, checkers, go, othello	backgammon monopoly
imperfect information		bridge, poker, scrabble nuclear war

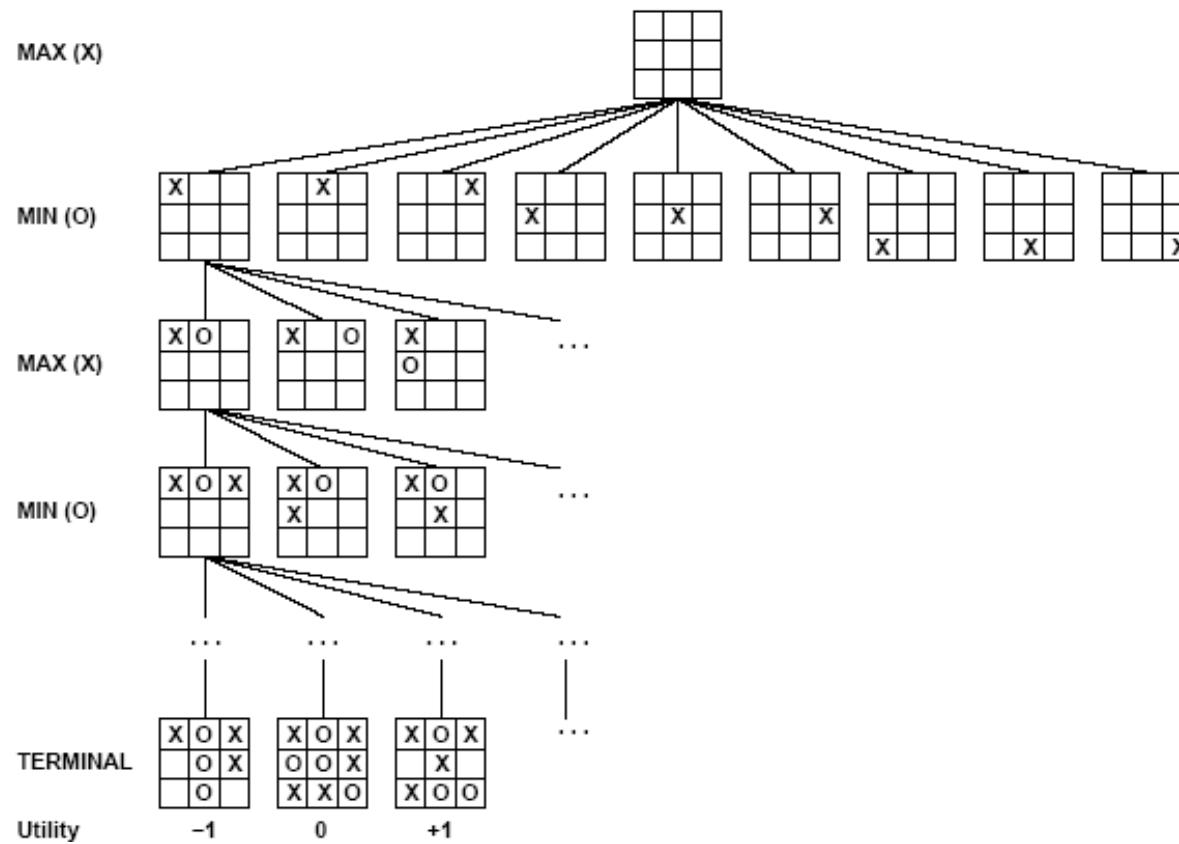
Game Setup

- Two players: **MAX** and **MIN**
- MAX moves first and they take turns until the game is over
 - Winner gets award, loser gets penalty.
- Games as search:
 - **Initial state**: e.g. board configuration of chess
 - **Successor function**: list of (move, state) pairs specifying legal moves.
 - **Terminal test**: Is the game finished?
 - **Utility function**: Gives numerical value of terminal states. E.g. **win (+1)**, **lose (-1)** and **draw (0)** in tic-tac-toe or chess
- MAX uses search tree to determine next move.

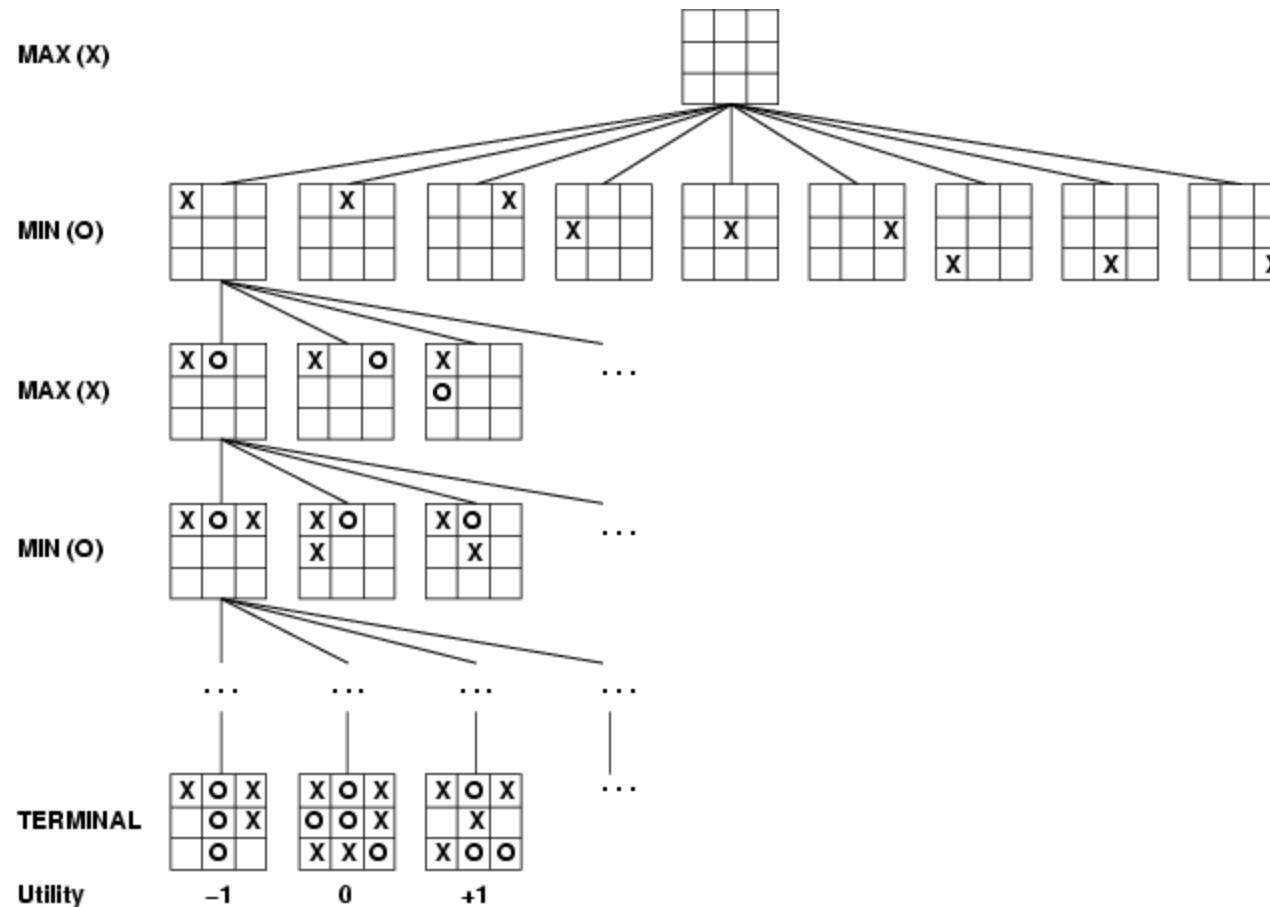
Size of search trees

- b = branching factor
- d = number of moves by both players
- Search tree is $O(b^d)$
- Chess
 - $b \sim 35$
 - $d \sim 100$
 - search tree is $\sim 10^{154}$ (!!)
 - completely impractical to search this
- Game-playing emphasizes being able to **make optimal decisions** in a **finite amount of time**
 - Somewhat realistic as a model of a real-world agent
 - Even if games themselves are artificial

Partial Game Tree for Tic-Tac-Toe



Game tree (2-player, deterministic, turns)



How do we search this tree to find the optimal move?

Minimax strategy

- Find the optimal *strategy* for MAX assuming an infallible MIN opponent
 - Need to compute this all the way down the tree
- Assumption: **Both players play optimally!**
- Given a game tree, the optimal strategy can be determined by using the minimax value of each node:

MINIMAX-VALUE(n) =

UTILITY(n)

$\max_{s \in \text{successors}(n)} \text{MINIMAX-VALUE}(s)$

$\min_{s \in \text{successors}(n)} \text{MINIMAX-VALUE}(s)$

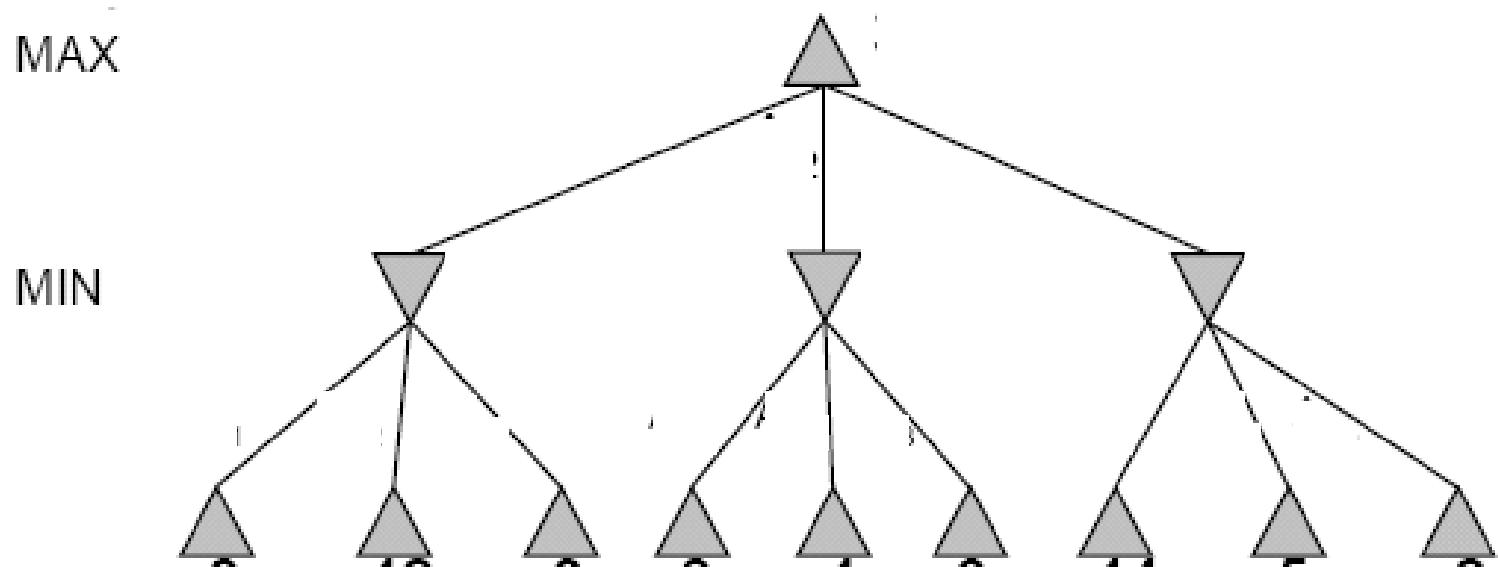
node

If n is a terminal

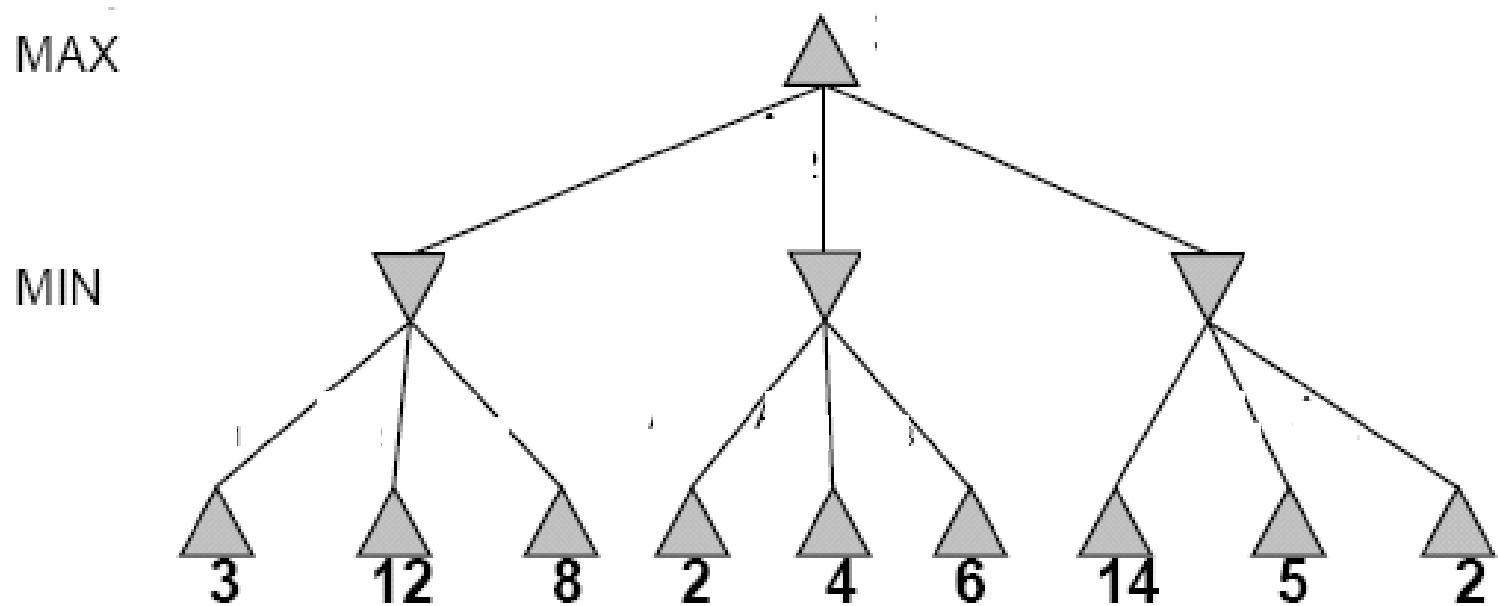
If n is a max node

If n is a min

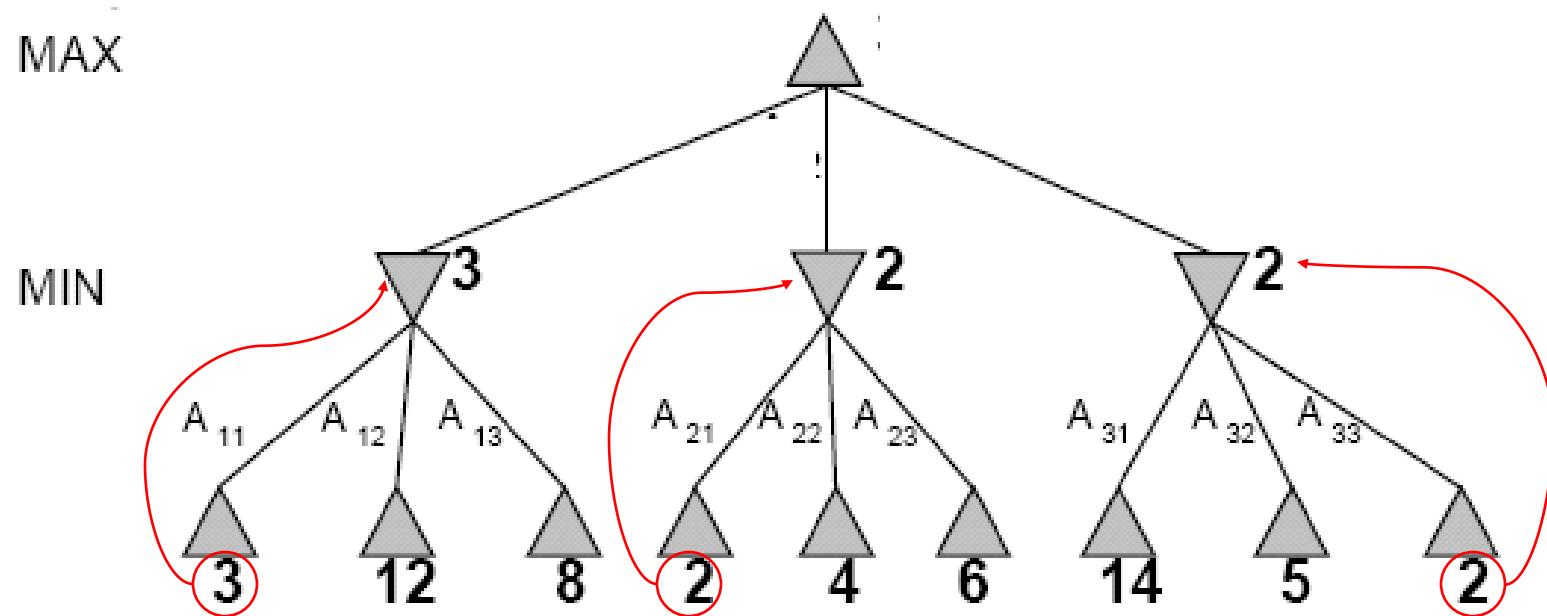
Two-Ply Game Tree



Two-Ply Game Tree

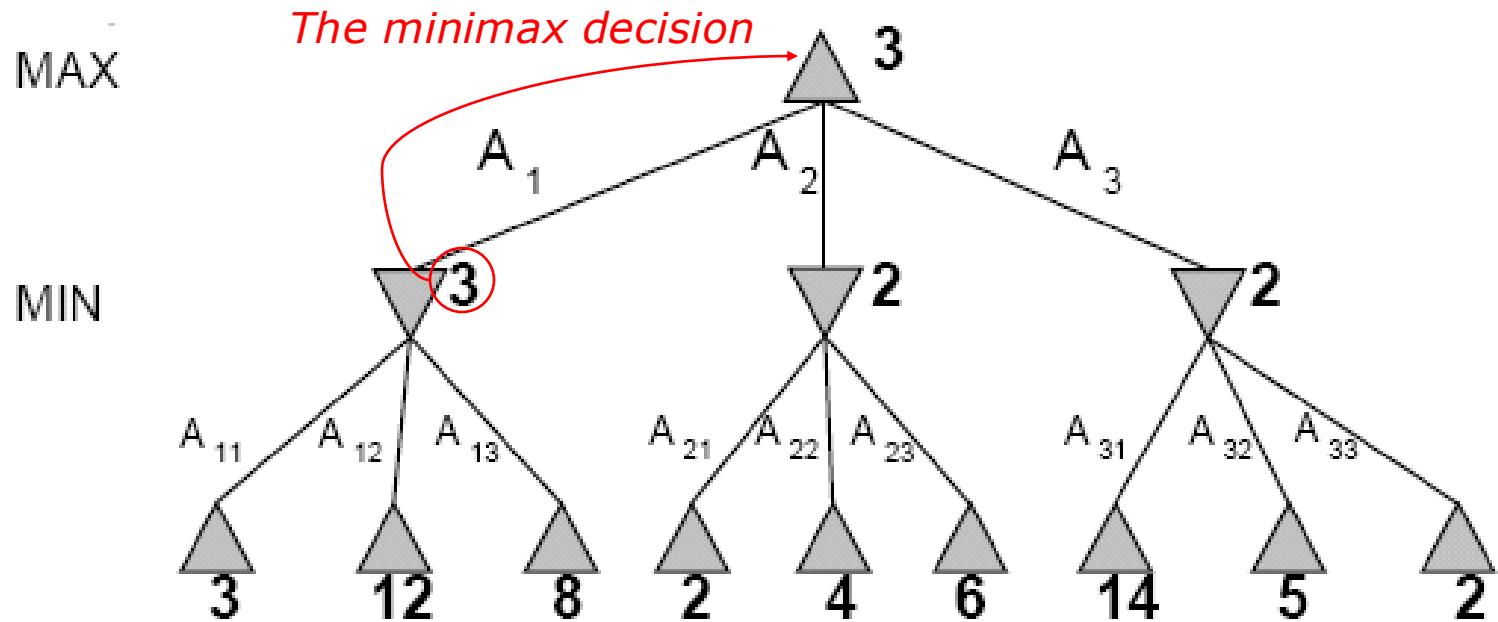


Two-Ply Game Tree



Two-Ply Game Tree

Minimax maximizes the utility for the worst-case outcome for max



What if MIN does not play optimally?

- Definition of optimal play for MAX assumes MIN plays optimally:
 - maximizes worst-case outcome for MAX
- But if MIN does not play optimally, MAX will do even better
 - Can prove this (Problem 6.2)

Minimax Algorithm

- Complete depth-first exploration of the game tree
- Assumptions:
 - Max depth = d , b legal moves at each point
 - E.g., Chess: $d \sim 100$, $b \sim 35$

Criterion	Minimax
Time	$O(b^m)$ 
Space	$O(bm)$ 

Pseudocode for Minimax Algorithm

function MINIMAX-DECISION(*state*) **returns** *an action*
inputs: *state*, current state in game
 $v \leftarrow \text{MAX-VALUE}(state)$
 return the *action* in SUCCESSORS(*state*) with value v

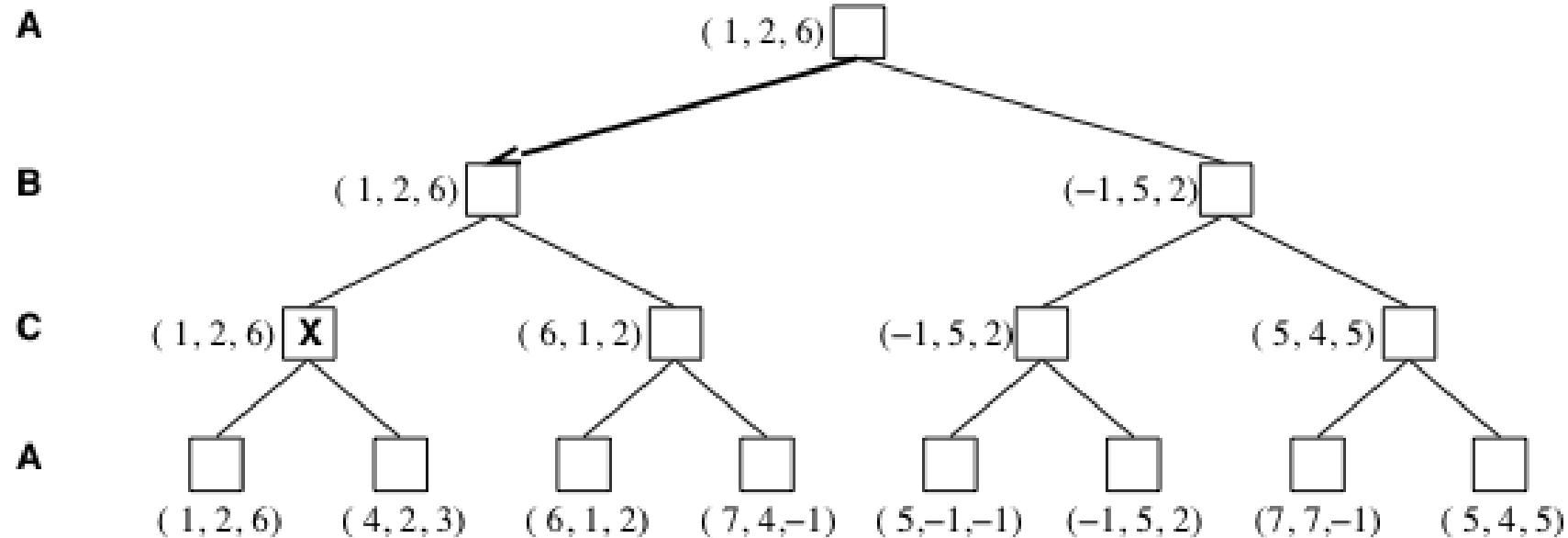
function MAX-VALUE(*state*) **returns** *a utility value*
 if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
 $v \leftarrow \infty$
 for *a,s* in SUCCESSORS(*state*) **do**
 $v \leftarrow \text{MAX}(v, \text{MIN-VALUE}(s))$
 return v

function MIN-VALUE(*state*) **returns** *a utility value*
 if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
 $v \leftarrow \infty$
 for *a,s* in SUCCESSORS(*state*) **do**
 $v \leftarrow \text{MIN}(v, \text{MAX-VALUE}(s))$
 return v

Multiplayer games

- Games allow more than two players
- Single minimax values become vectors

to move



Aspects of multiplayer games

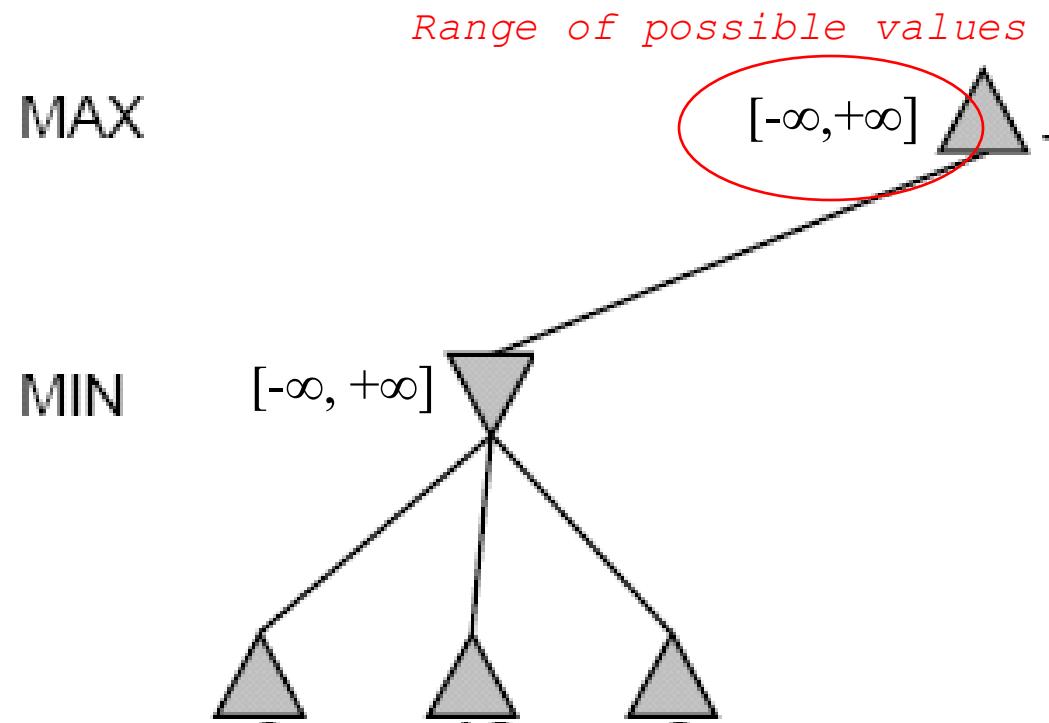
- Previous slide (standard minimax analysis) assumes that each player operates to maximize only their own utility
- In practice, players make alliances
 - E.g., C strong, A and B both weak
 - May be best for A and B to attack C rather than each other
- If game is not zero-sum (i.e., $\text{utility}(A) = - \text{utility}(B)$) then alliances can be useful even with 2 players
 - e.g., both cooperate to maximize the sum of the utilities

Practical problem with minimax search

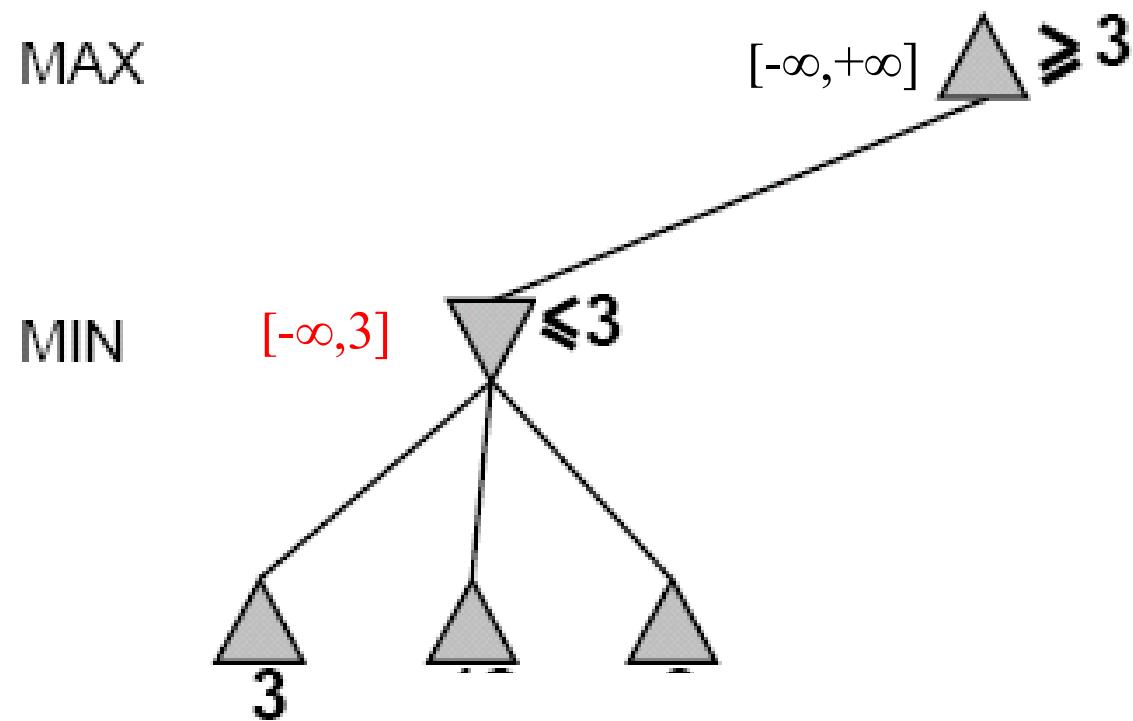
- Number of game states is **exponential** in the number of moves.
 - Solution: **Do not examine every node**
=> **pruning**
 - Remove branches that do not influence final decision
- Revisit example ...

Alpha-Beta Example

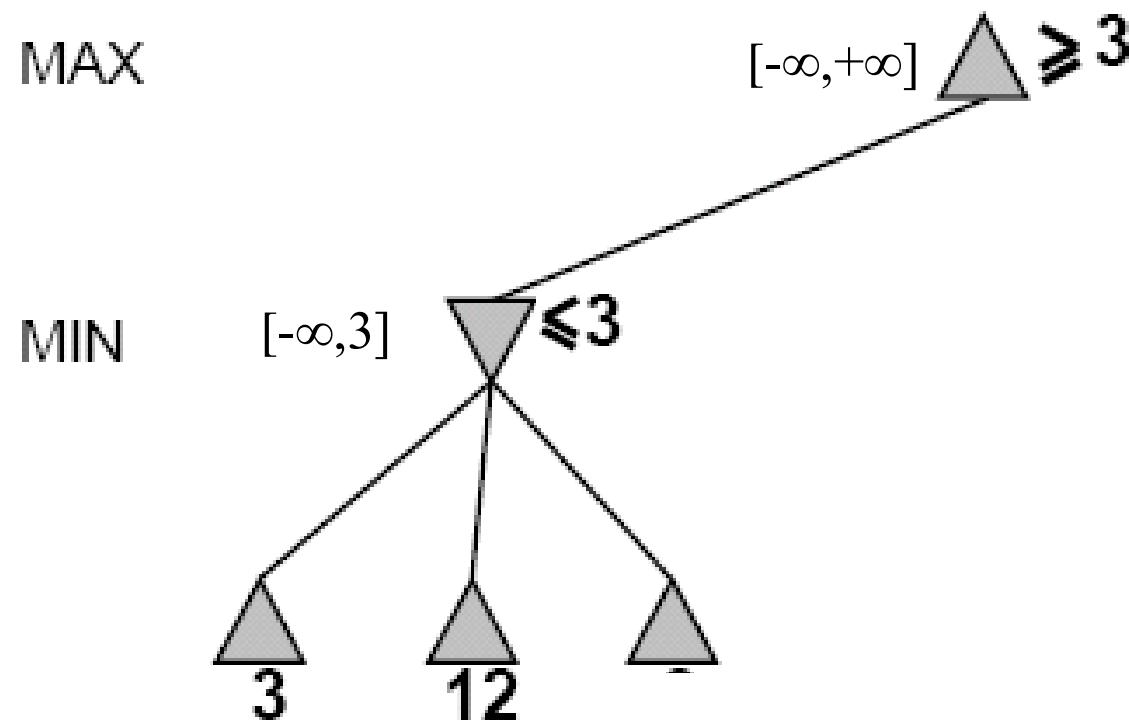
Do DF-search until first leaf



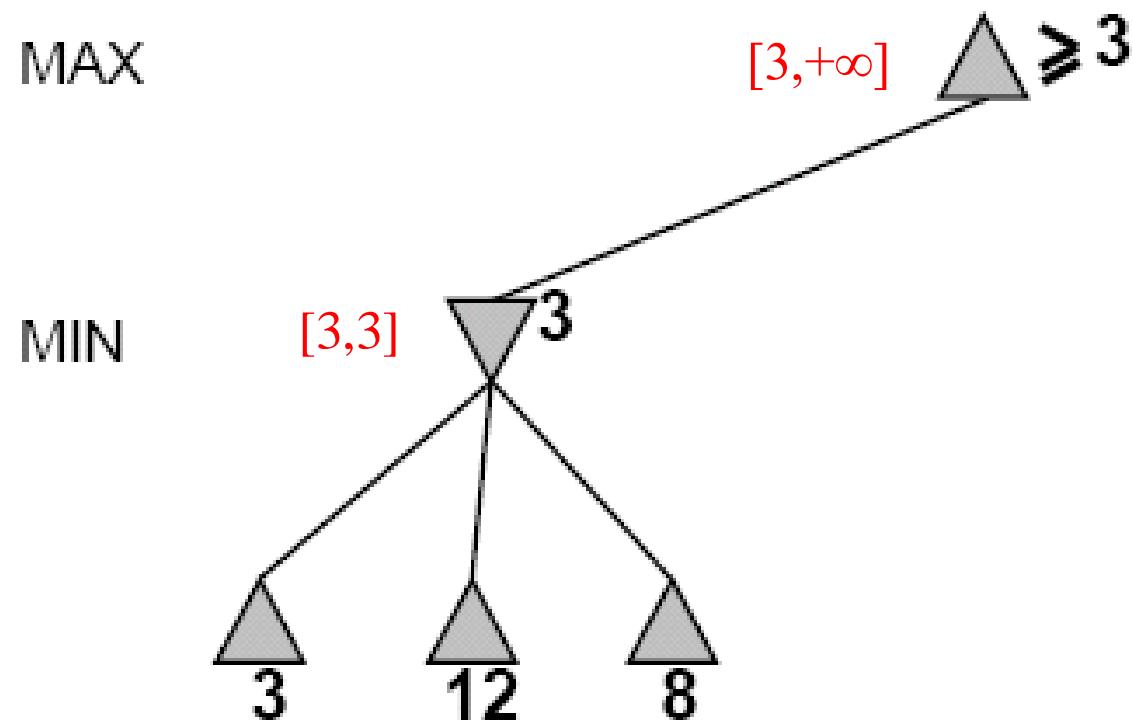
Alpha-Beta Example (continued)



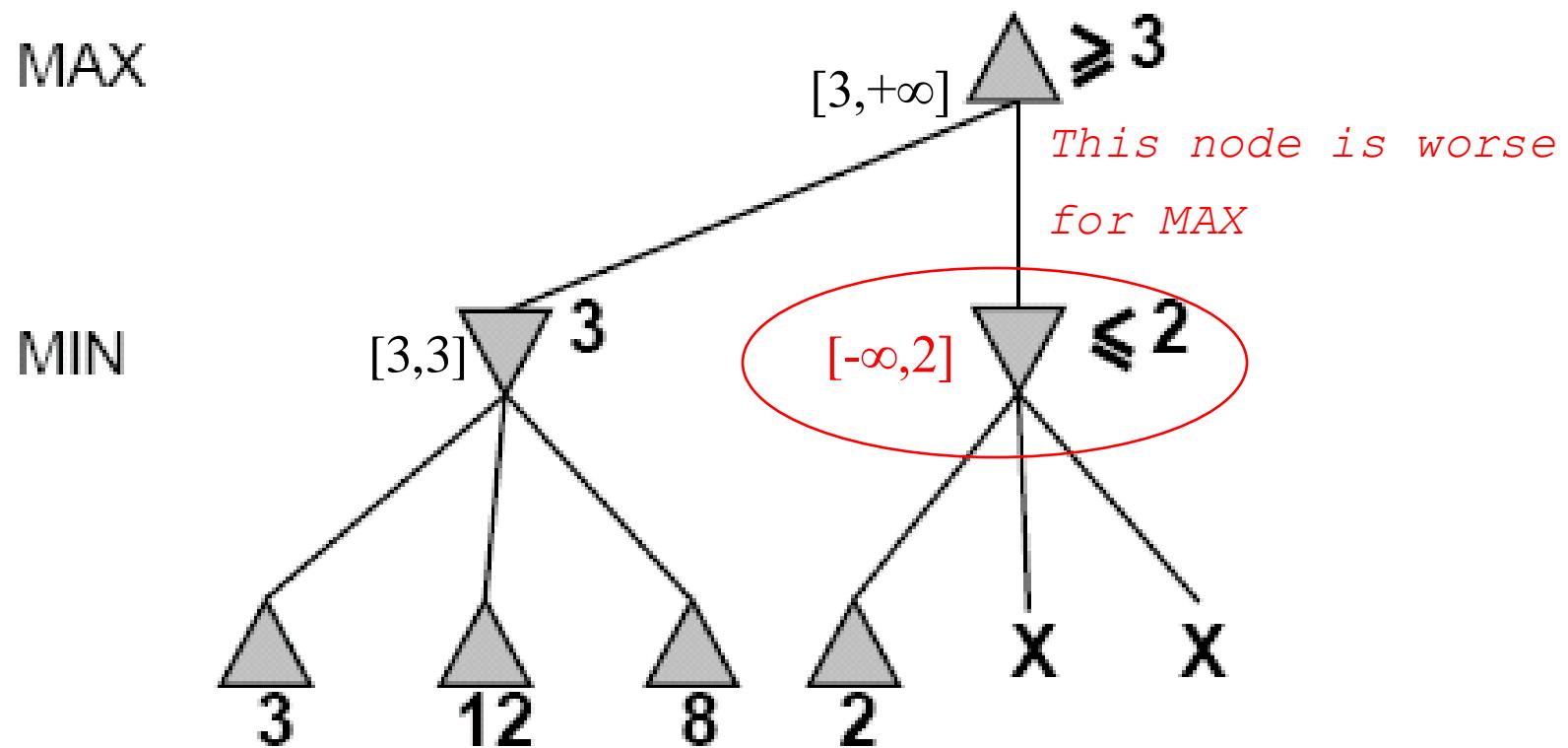
Alpha-Beta Example (continued)



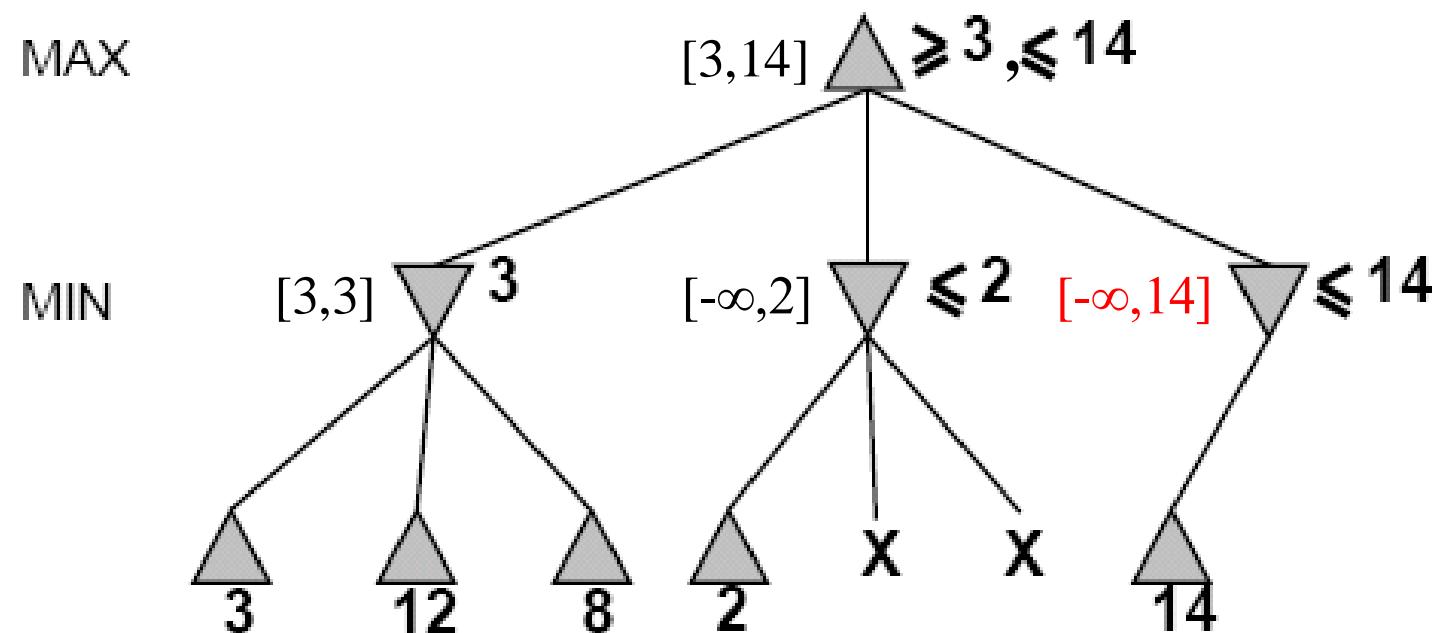
Alpha-Beta Example (continued)



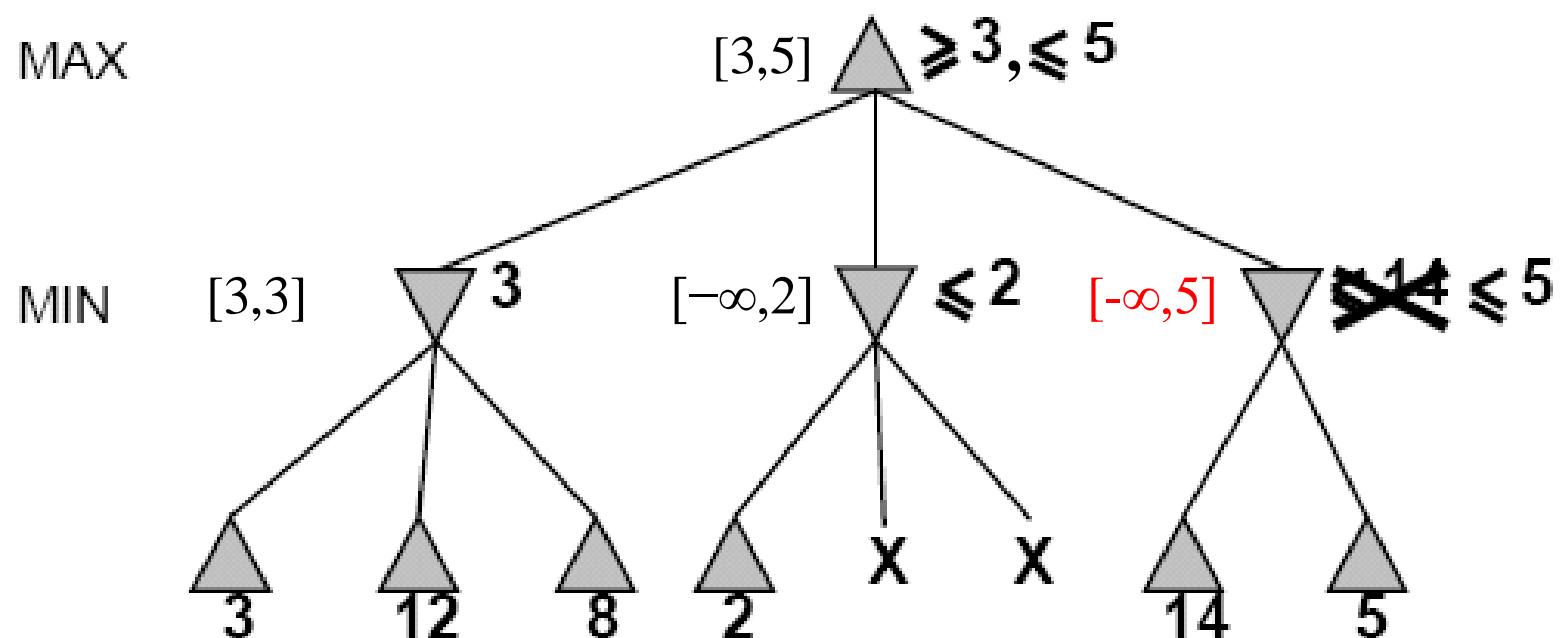
Alpha-Beta Example (continued)



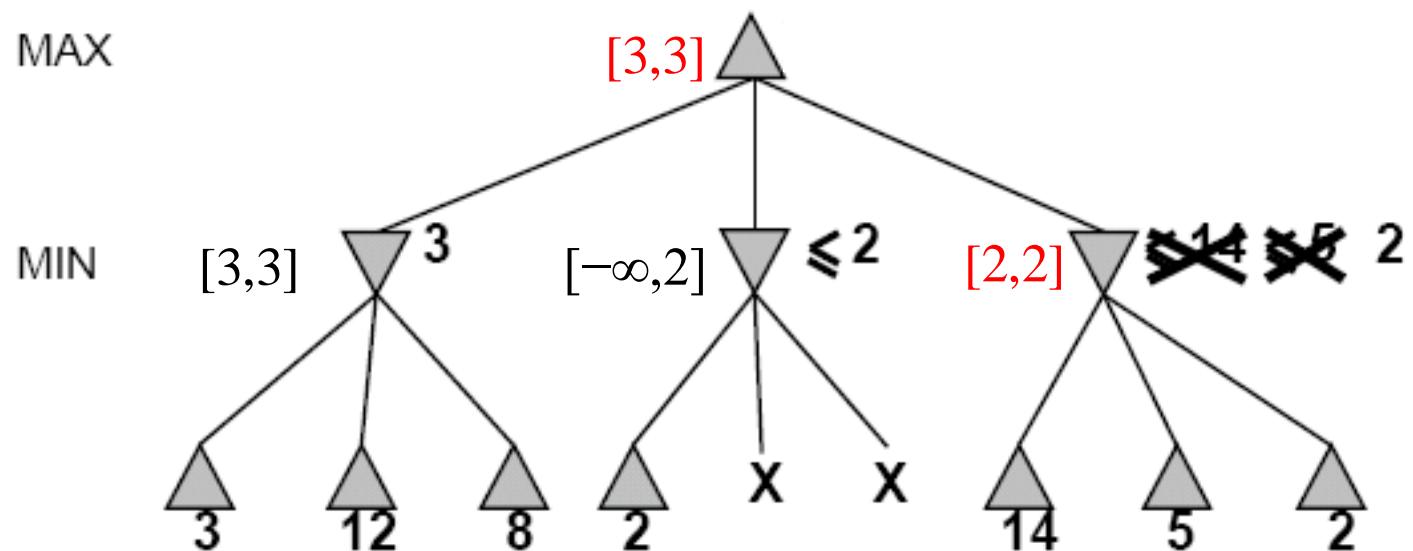
Alpha-Beta Example (continued)



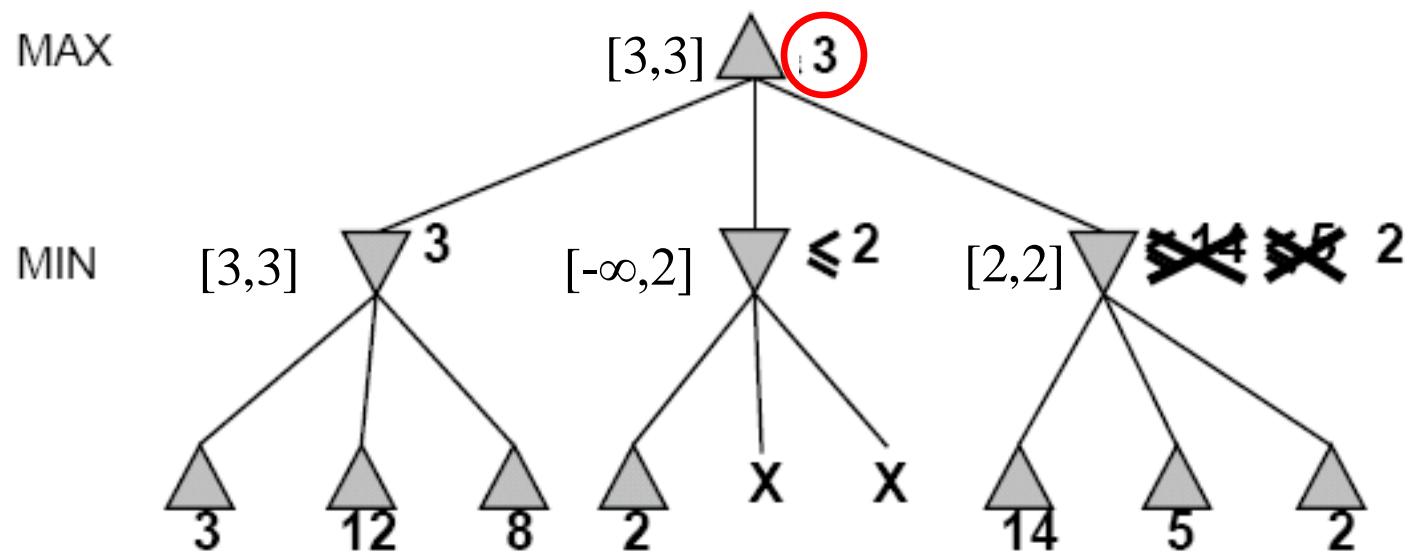
Alpha-Beta Example (continued)



Alpha-Beta Example (continued)

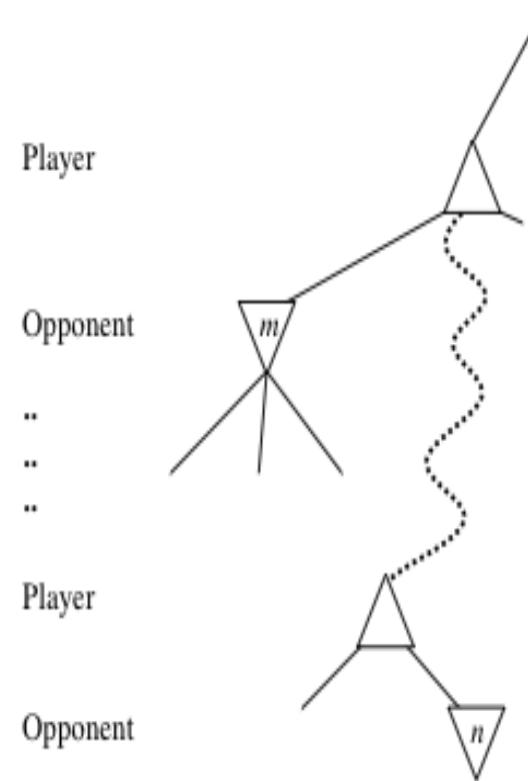


Alpha-Beta Example (continued)



General alpha-beta pruning

- Consider a node n somewhere in the tree
- If player has a better choice at
 - Parent node of n
 - Or any choice point further up
- n will **never** be reached in actual play.
- Hence when enough is known about n , it can be pruned.



Alpha-beta Algorithm

- Depth first search – only considers nodes along a single path at any time

α = highest-value choice we have found at any choice point along the path for MAX

β = lowest-value choice we have found at any choice point along the path for MIN

- update values of α and β during search and prunes remaining branches as soon as the value is known to be worse than the current α or β value for MAX or MIN

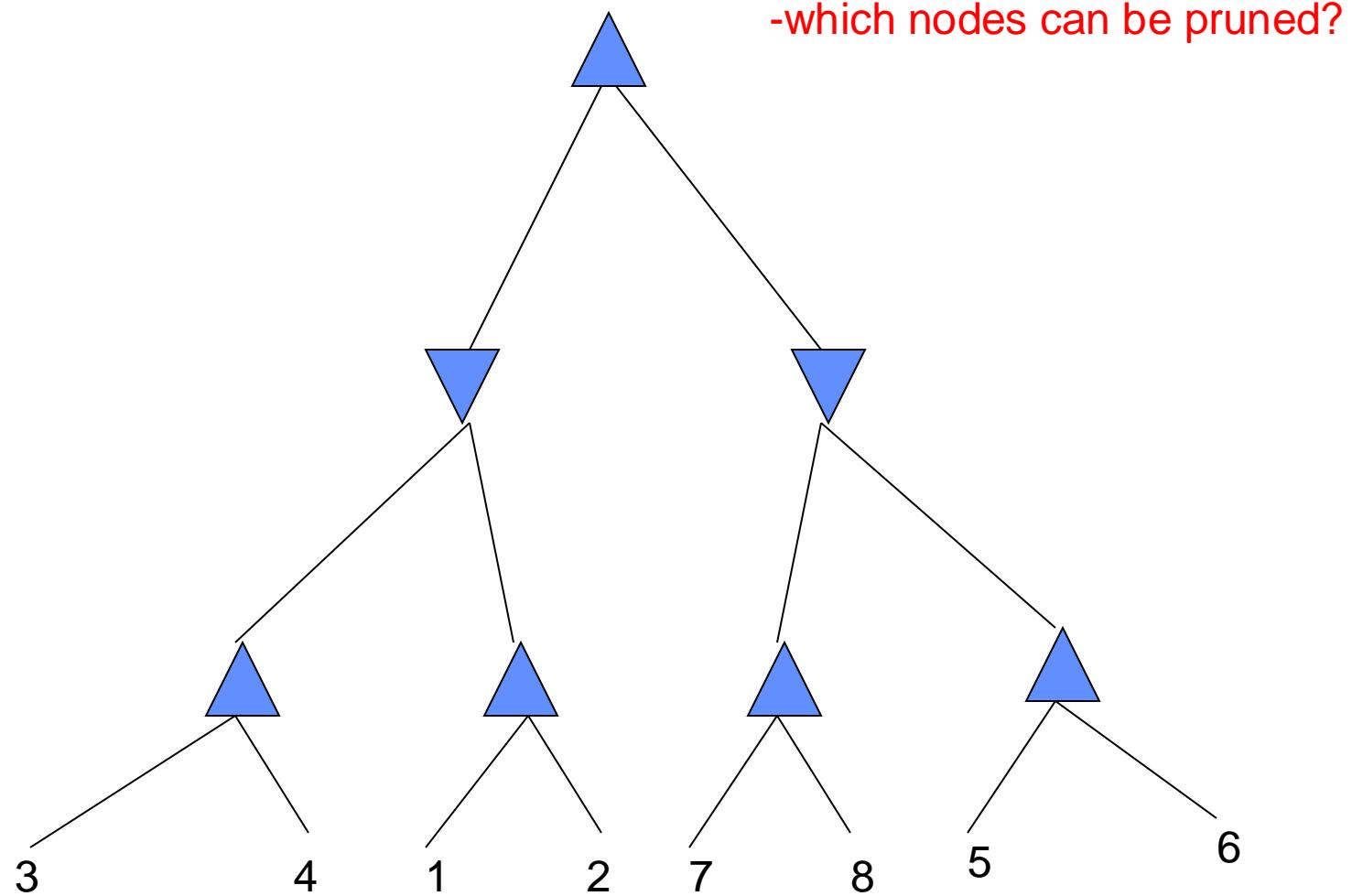
Effectiveness of Alpha-Beta Search

- Worst-Case
 - branches are ordered so that no pruning takes place. In this case alpha-beta gives no improvement over exhaustive search
- Best-Case
 - each player's best move is the left-most alternative (i.e., evaluated first)
 - in practice, performance is closer to best rather than worst-case
- In practice often get $O(b^{(d/2)})$ rather than $O(b^d)$
 - this is the same as having a branching factor of \sqrt{b} ,
 - since $(\sqrt{b})^d = b^{(d/2)}$
 - i.e., we have effectively gone from b to square root of b
 - e.g., in chess go from $b \sim 35$ to $b \sim 6$
 - this permits much deeper search in the same amount of time

Final Comments about Alpha-Beta Pruning

- Pruning does not affect final results
- Entire subtrees can be pruned.
- Good move *ordering* improves effectiveness of pruning
- Repeated states are again possible.
 - Store them in memory = transposition table

Example



Practical Implementation

How do we make these ideas practical in real game trees?

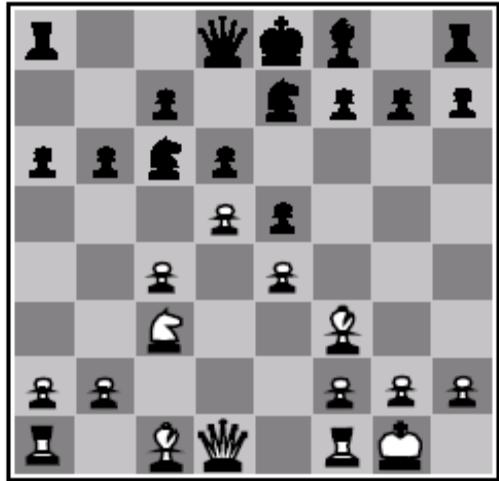
Standard approach:

- **cutoff test:** (where do we stop descending the tree)
 - depth limit
 - better: iterative deepening
 - cutoff only when no big changes are expected to occur next (**quiescence search**).
- **evaluation function**
 - When the search is cut off, we evaluate the current state by estimating its utility. This estimate is captured by the evaluation function.

Static (Heuristic) Evaluation Functions

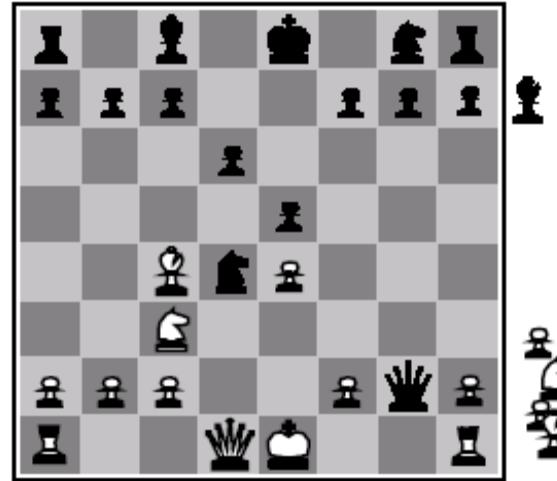
- An Evaluation Function:
 - estimates how good the current board configuration is for a player.
 - Typically, one figures how good it is for the player, and how good it is for the opponent, and subtracts the opponents score from the players
 - Othello: Number of white pieces - Number of black pieces
 - Chess: Value of all white pieces - Value of all black pieces
- Typical values from -infinity (loss) to +infinity (win) or [-1, +1].
- If the board evaluation is X for a player, it's -X for the opponent
- Example:
 - Evaluating chess boards,
 - Checkers
 - Tic-tac-toe

Evaluation functions



Black to move

White slightly better



White to move

Black winning

For chess, typically *linear* weighted sum of *features*

$$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \dots + w_n f_n(s)$$

e.g., $w_1 = 9$ with

$f_1(s) = (\text{number of white queens}) - (\text{number of black queens}), \text{ etc.}$

Iterative (Progressive) Deepening

- In real games, there is usually a time limit T on making a move
- How do we take this into account?
 - using alpha-beta we cannot use “partial” results with any confidence unless the full breadth of the tree has been searched
 - So, we could be conservative and set a conservative depth-limit which guarantees that we will find a move in time $< T$
 - disadvantage is that we may finish early, could do more search
- In practice, iterative deepening search (IDS) is used
 - IDS runs depth-first search with an increasing depth-limit
 - when the clock runs out we use the solution found at the previous depth limit

Heuristics and Game Tree Search

- The Horizon Effect
 - sometimes there's a major “effect” (such as a piece being captured) which is just “below” the depth to which the tree has been expanded
 - the computer cannot see that this major event could happen
 - it has a “limited horizon”
 - there are heuristics to try to follow certain branches more deeply to detect such important events
 - this helps to avoid catastrophic losses due to “short-sightedness”
- Heuristics for Tree Exploration
 - it may be better to explore some branches more deeply in the allotted time
 - various heuristics exist to identify “promising” branches

The State of Play

- Checkers:
 - Chinook ended 40-year-reign of human world champion Marion Tinsley in 1994.
- Chess:
 - Deep Blue defeated human world champion Garry Kasparov in a six-game match in 1997.
- Othello:
 - human champions refuse to compete against computers: they are too good.
- Go:
 - human champions refuse to compete against computers: they are too bad
 - $b > 300$ (!)
- See (e.g.) <http://www.cs.ualberta.ca/~games/> for more information

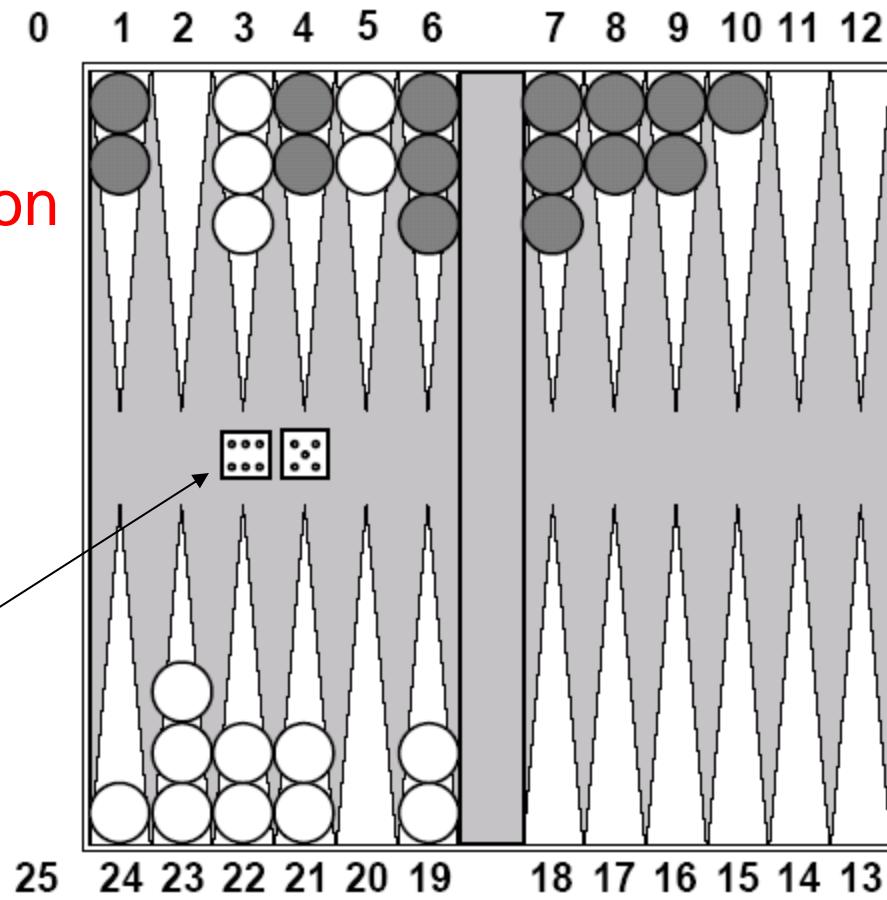
Deep Blue

- 1957: Herbert Simon
 - “within 10 years a computer will beat the world chess champion”
- 1997: Deep Blue beats Kasparov
- Parallel machine with 30 processors for “software” and 480 VLSI processors for “hardware search”
- Searched 126 million nodes per second on average
 - Generated up to 30 billion positions per move
 - Reached depth 14 routinely
- Uses iterative-deepening alpha-beta search with transpositioning
 - Can explore beyond depth-limit for interesting moves

Chance Games.

Backgammon

your element of chance



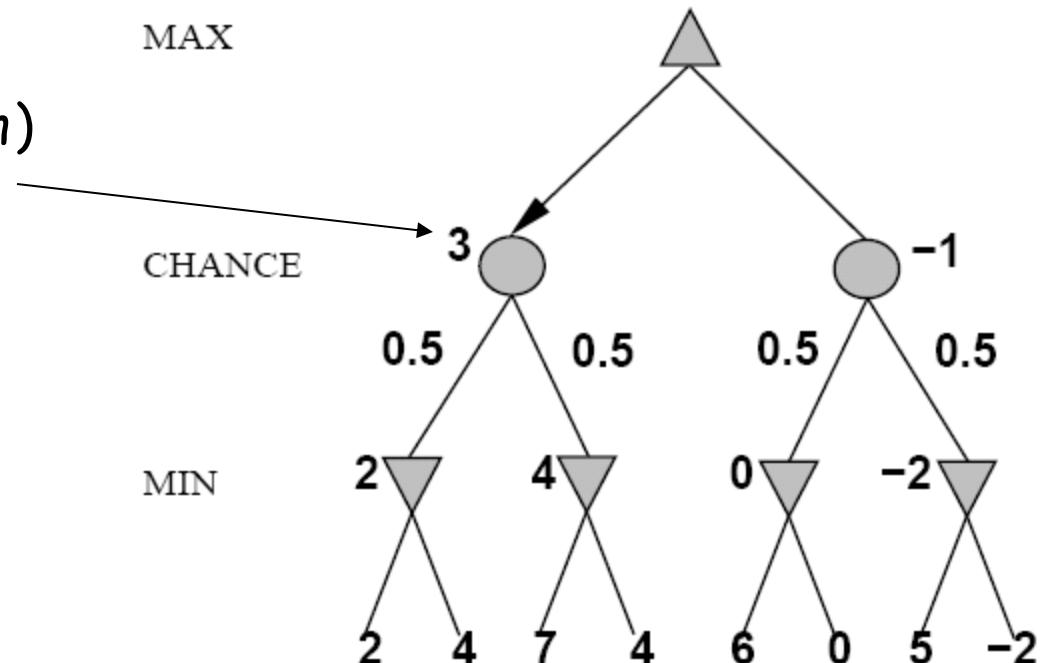
Expected Minimax

$$v = \sum_{\text{chance nodes}} P(n) \times \text{Minimax}(n)$$

$$3 = 0.5 \times 4 + 0.5 \times 2$$

Interleave chance nodes
with min/max nodes

Again, the tree is constructed
bottom-up



Summary

- Game playing can be effectively modeled as a search problem
- Game trees represent alternate computer/opponent moves
- Evaluation functions estimate the quality of a given board configuration for the Max player.
- Minimax is a procedure which chooses moves by assuming that the opponent will always choose the move which is best for them
- Alpha-Beta is a procedure which can prune large parts of the search tree and allow search to go deeper
- For many well-known games, computer algorithms based on heuristic search match or outperform human world experts.

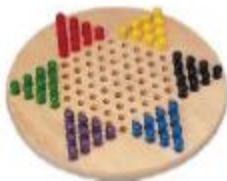
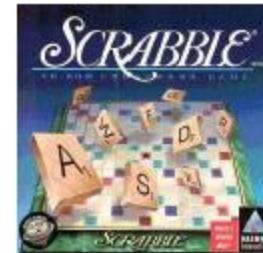
CSE 401

*Game Playing
Alpha-Beta Search
and
General Issues*

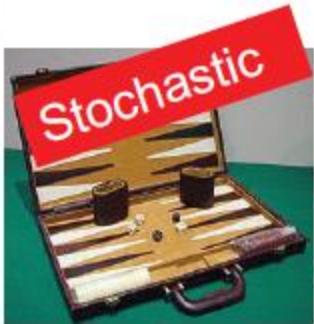
Which of these are: 2-player zero-sum discrete finite deterministic games of perfect information



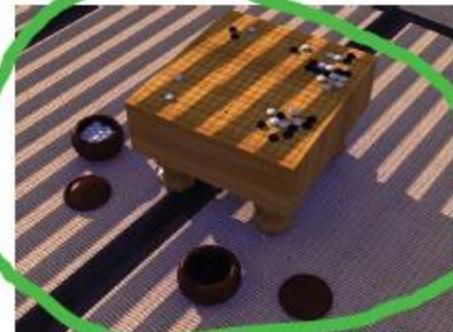
- **Two player:** Duh!
- **Zero-sum:** In any outcome of any game, Player A's gains equal player B's losses.
- **Discrete:** All game states and decisions are discrete values.
- **Finite:** Only a finite number of states and decisions.
- **Deterministic:** No chance (no die rolls).
- **Games:** See next page
- **Perfect information:** Both players can see the state, and each decision is made sequentially (no simultaneous moves).



Which of these are: 2-player zero-sum discrete finite deterministic games of perfect information



- **Two player:** Duh!
- **Zero-sum:** In any outcome of any game, Player A's gains equal player B's losses.
- **Discrete:** All game states and decisions are discrete values.
- **Finite:** Only a finite number of states and decisions.
- **Deterministic:** No chance (no die rolls).
- **Games:** See next page
- **Perfect information:** Both players can see the state, and each decision is made sequentially (no simultaneous moves).



Definition

A Two-player zero-sum discrete finite deterministic game of perfect information is a quintuplet: $(S, I, \text{Succs}, T, V)$ where

S	$=$	a finite set of states (note: state includes information sufficient to deduce who is due to move)
I	$=$	the initial state
Succs	$=$	a function which takes a state as input and returns a set of possible next states available to whoever is due to move
T	$=$	a subset of S . It is the terminal states: the set of states at which the game is over
V	$=$	a mapping from terminal states to real numbers. It is the amount that A wins from B. (If it's negative A loses money to B).

Convention: assume Player A moves first.

For convenience: assume turns alternate.

Utility Function

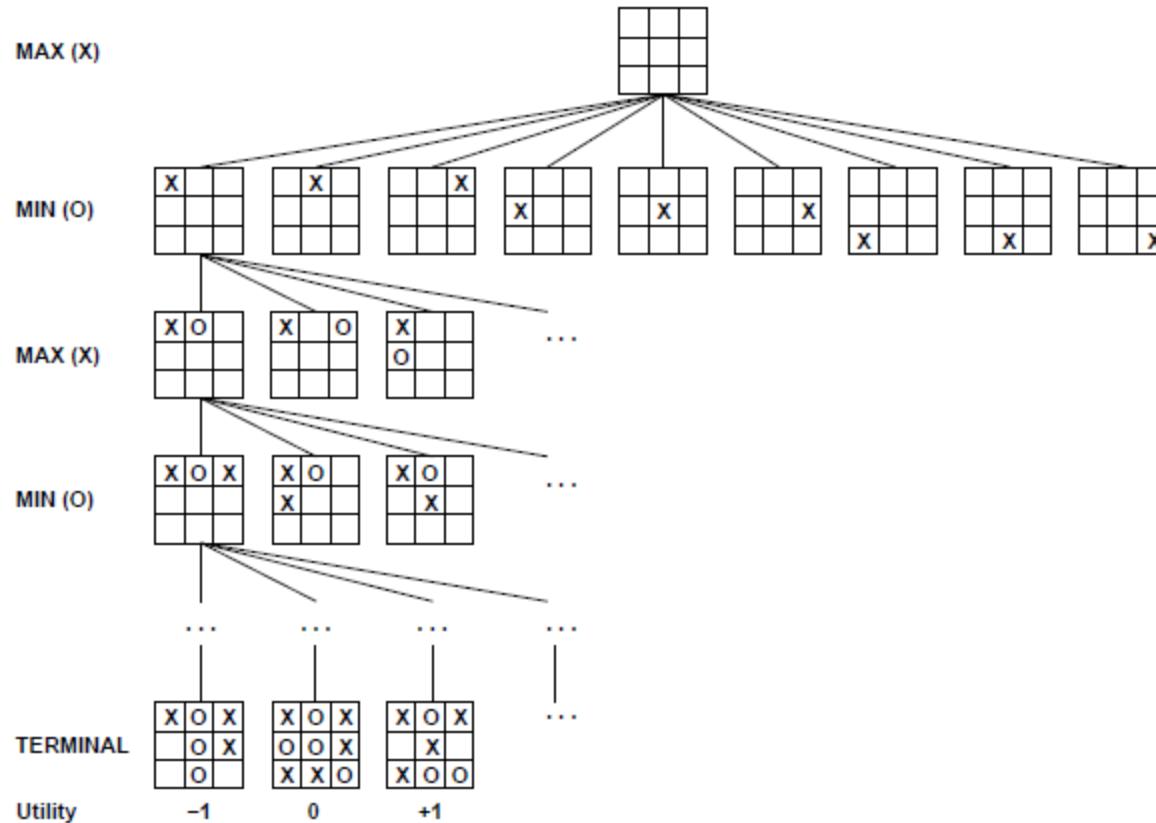
- Gives the utility of a game state
 - $\text{utility}(\text{State})$
- Examples
 - -1, 0, and +1, for Player 1 loses, draw, Player 1 wins, respectively
 - Difference between the point totals for the two players
 - Weighted sum of factors (e.g. Chess)
 - $\text{utility}(S) = w_1 f_1(S) + w_2 f_2(S) + \dots + w_n f_n(S)$
 - $f_1(S) = (\text{Number of white queens}) - (\text{Number of black queens}), \quad w_1 = 9$
 - $f_2(S) = (\text{Number of white rooks}) - (\text{Number of black rooks}), \quad w_2 = 5$

Game Playing

- **Game tree**
 - describes the possible sequences of play
 - is a graph if we merge together identical states
- **Minimax:**
 - utility values assigned to the leaves
- **Values “backed up” the tree by**
 - MAX node takes max value of children
 - MIN node takes min value of children
 - Can read off best lines of play
- **Depth Bounded Minimax**
 - utility of terminal states estimated using an “evaluation function”

Game Tree for Tic-tac-toe

Game tree (2-player, deterministic, turns)



Minimax

$\text{MINIMAX-VALUE}(n) =$

$$\begin{cases} \text{UTILITY}(n) & \text{if } n \text{ is a terminal state} \\ \max_{s \in \text{Successors}(n)} \text{MINIMAX-VALUE}(s) & \text{if } n \text{ is a MAX node} \\ \min_{s \in \text{Successors}(n)} \text{MINIMAX-VALUE}(s) & \text{if } n \text{ is a MIN node.} \end{cases}$$

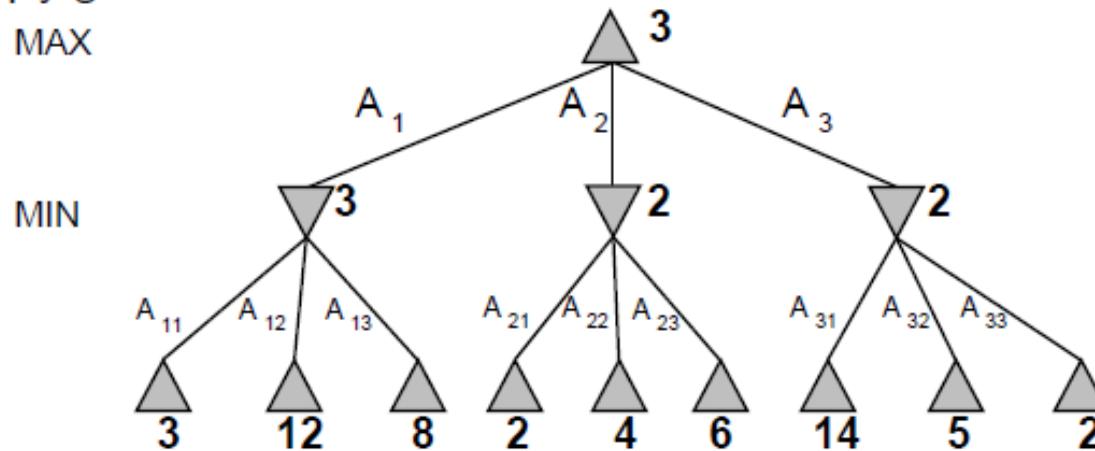
Minimax

Minimax

Perfect play for deterministic, perfect-information games

Idea: choose move to position with highest **minimax value**
= best achievable payoff against best play

E.g., 2-ply game:



Basic Algorithm

function **MINIMAX-DECISION**(*state*) **returns** *an action*

inputs: *state*, current state in game

return the *a* in **ACTIONS**(*state*) maximizing **MIN-VALUE**(**RESULT**(*a, state*))

function **MAX-VALUE**(*state*) **returns** *a utility value*

if **TERMINAL-TEST**(*state*) **then return** **UTILITY**(*state*)

v $\leftarrow -\infty$

for *a, s* in **SUCCESSORS**(*state*) **do** *v* $\leftarrow \text{MAX}(v, \text{MIN-VALUE}(s))$

return *v*

function **MIN-VALUE**(*state*) **returns** *a utility value*

if **TERMINAL-TEST**(*state*) **then return** **UTILITY**(*state*)

v $\leftarrow \infty$

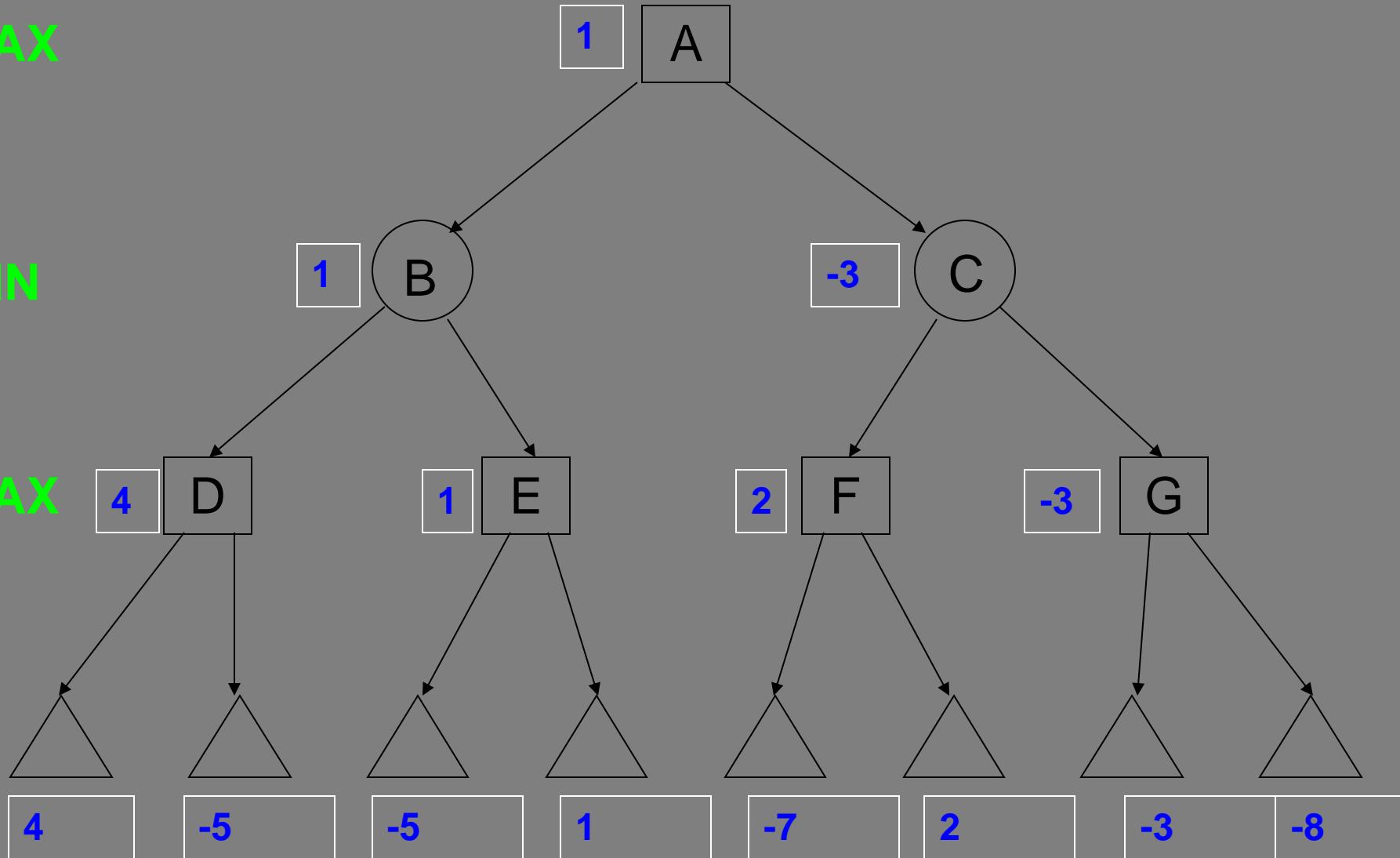
for *a, s* in **SUCCESSORS**(*state*) **do** *v* $\leftarrow \text{MIN}(v, \text{MAX-VALUE}(s))$

return *v*

MAX

MIN

MAX



= terminal position



= agent



= opponent

Game Playing – Beyond Minimax

- **Efficiency of the search**
 - Game trees are very big
 - Evaluation of positions is time-consuming
- **How can we reduce the number of nodes to be evaluated?**
 - “alpha-beta search”
- **Bounding the depth of minimax has deficiencies**
 - Why?
 - How can we mitigate these deficiencies?

Game Playing – Improving Efficiency

- Suppose that we are doing depth-bounded minimax
- We have a game tree to create and then insert the minimax values in order to find the values for our possible moves from the current position

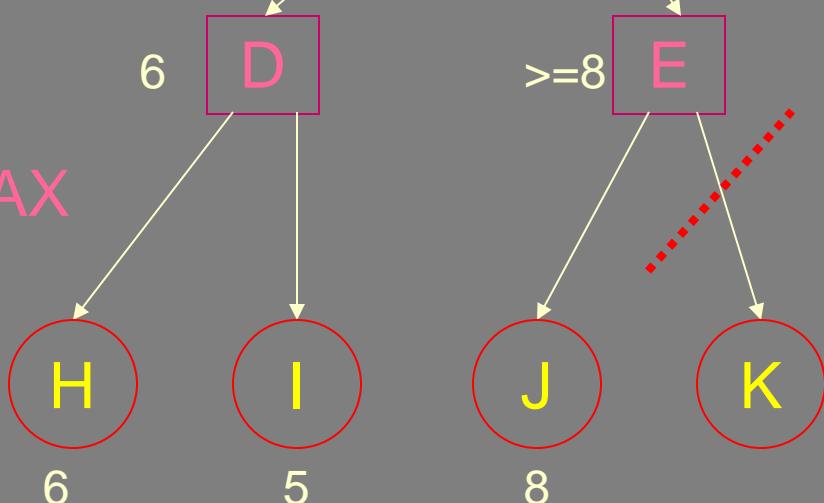
Game Playing – Minimax using DFS

- The presentation of minimax was done by “backing up from the leaves” – a “bottom-up” breadth-first search.
- This has the disadvantage of taking a lot of space
 - Compare this with the space usage issues for DFS vs. BFS in earlier lectures
- If we can do minimax using DFS then it is likely to take a lot less space
- Minimax can be implemented using DFS
- But reduced space is not the only advantage:

MAX

MIN

MAX



**STOP! What else can
you deduce now!?**

On discovering $\text{util}(D) = 6$

we know that $\text{util}(B) \leq 6$

On discovering $\text{util}(J) = 8$

we know that $\text{util}(E) \geq 8$

Can stop expansion of E as best
play will not go via E

Value of K is irrelevant – prune it!

= agent

= opponent

Game Playing – Pruning nodes

- If we are scanning the tree using DFS then there was no point in evaluating node K
- Whatever the value of K there cannot be any rational sequence of play that would go through it
 - Node K can be pruned from the search: i.e. just not selected for further expansion
- “At node B then MIN will never select E; because J is better than D for MAX and so MIN must not allow MAX to have that opportunity”
- Q. So what! It’s just one node?
- A. Suppose that the depth limit were such that K was far from the depth bound. Then evaluating K corresponds to a large sub-tree. Such prunings can save an exponential amount of work

Game Playing – Improving Efficiency

- Suppose that we were doing Breadth-First Search, would you still be able to prune nodes in this fashion?
- NO! Because the pruning relied on the fact that we had already evaluated node D by evaluating the tree underneath D
- This form of pruning is an example of “alpha-beta pruning” and relies on doing a DEPTH-FIRST search of the depth bounded tree

Game Playing – Node-ordering

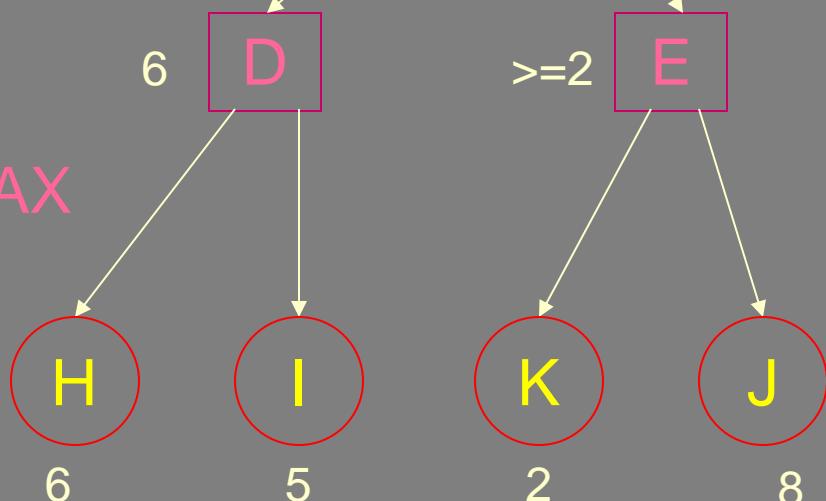
- **Suppose that**
 - nodes K and J were evaluated in the opposite order
 - can we expect that we would be able to do a similar pruning?
- The answer depends on the value of K
- Suppose that K had a value of 2 and is expanded first:

**STOP! What else can
you deduce now!?**

MAX

MIN

MAX



On discovering $\text{util}(D) = 6$
we know that $\text{util}(B) \leq 6$

On discovering $\text{util}(K) = 2$
we know that $\text{util}(E) \geq 2$

Can NOT stop expansion of E as
best play might still go via E
Value of J is relevant – no pruning

 = agent

 = opponent

Game Playing – Node-ordering

- When K had a value of 2 and was expanded first then we did not get to prune a child of E
- To maximise pruning we want to first expand those children that are best for the parent
 - cannot know which ones are really best
 - use heuristics for the “best-first” ordering
- If this is done well then alpha-beta search can effectively double the depth of search tree that is searchable in a given time
 - Effectively reduces the branching factor in chess from about 30 to about 8
 - This is an enormous improvement!

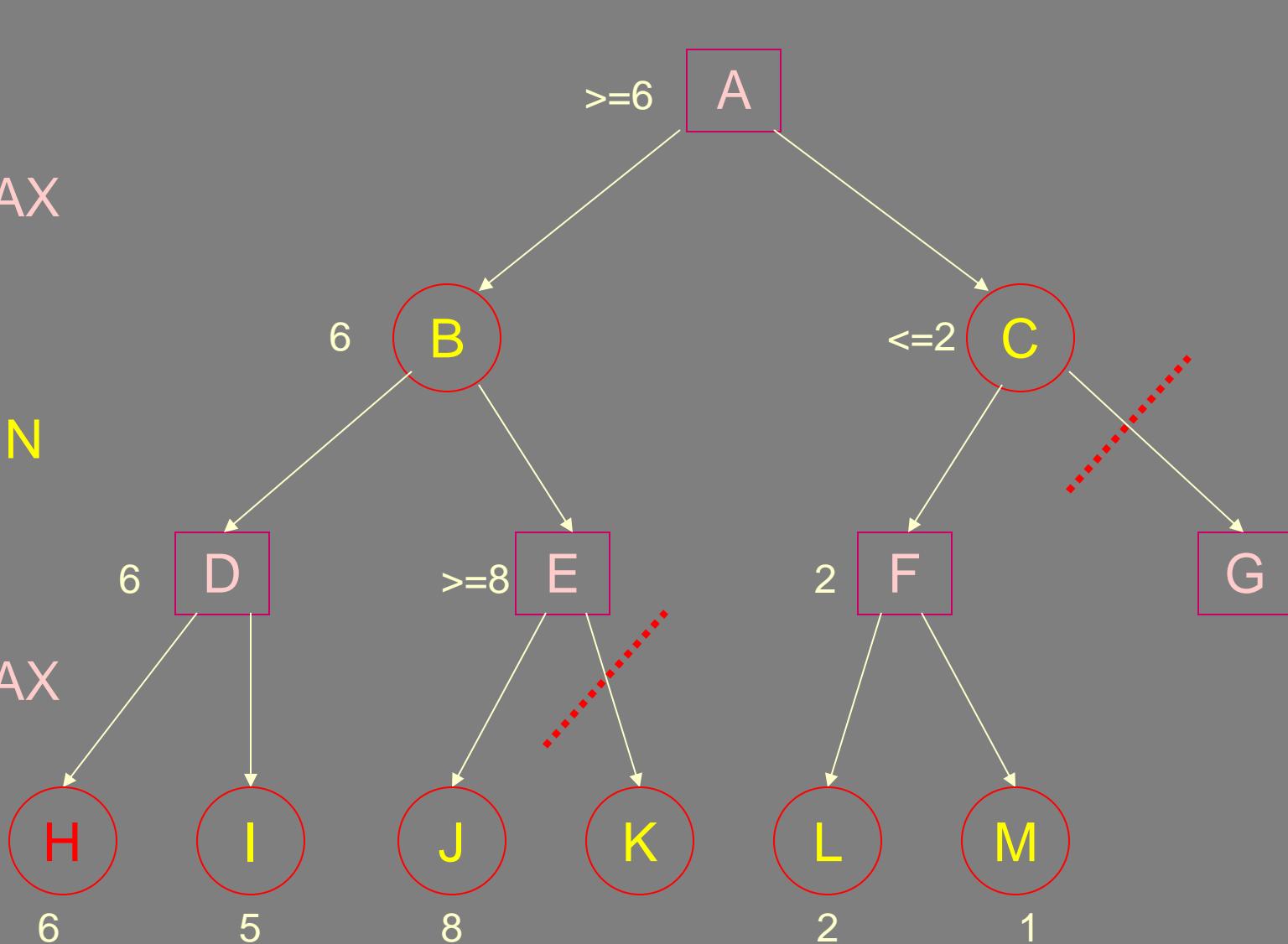
Game Playing – Improving Efficiency

- The games are symmetric so is natural that we can also do a similar pruning with the MIN and MAX roles reversed
- The reasoning is identical other than for the reversal of roles
- Can deduce that some other nodes can not be involved in the line of best play

MAX

MIN

MAX



= agent

= opponent

Game Playing – Alpha-Beta Implementation

- The pruning was based on using the results of the “DFS so far” to deduce upper and lower bounds on the values of nodes
- Conventionally these bounds are stored in terms of two parameters
 - alpha α
 - beta β

Game Playing – Alpha-Beta Implementation

- **a values are stored with each MAX node**
- **each MAX node is given a value of alpha that is the current best lower-bound on its final value**
 - initially is $-\infty$ to represent that nothing is known
 - as we do the search then α at a node can increase, but it can never decrease – it always gets better for MAX

Game Playing – Alpha-Beta Implementation

- **β values are stored with each MIN node**
- each MIN node is given a value of beta that is the current best upper-bound on its final value
 - initially is $+\infty$ to represent that nothing is known
 - as we do the search then β at a node can decrease, but it can never increase – it always gets better for MIN

Alpha-beta Pruning

MAX

alpha pruning as
 $\beta(C) < \alpha(A)$

MIN

beta pruning as
 $\alpha(E) > \beta(B)$

MAX

6

D

$\beta = 6$

$\alpha = 8$

E

2

G

H

6

I

5

J

8

K

L

2

M

1



= agent



= opponent

Properties of α - β

Pruning **does not** affect final result

Good move ordering improves effectiveness of pruning

With “perfect ordering,” time complexity = $O(b^{m/2})$
⇒ **doubles** solvable depth

A simple example of the value of reasoning about which computations are relevant (a form of metareasoning)

Unfortunately, 35^{50} is still impossible!

Resource limits

Standard approach:

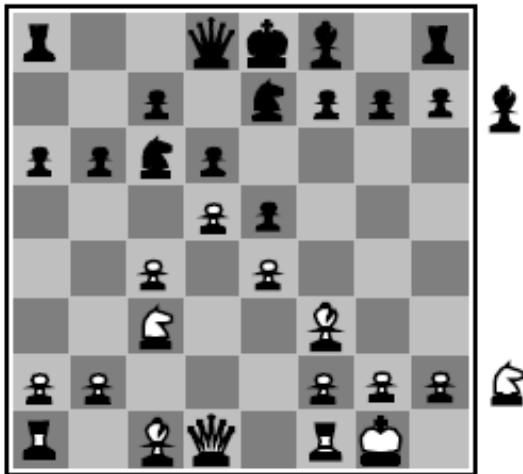
- Use CUTOFF-TEST instead of TERMINAL-TEST
 - e.g., depth limit (perhaps add quiescence search)
- Use EVAL instead of UTILITY
 - i.e., evaluation function that estimates desirability of position

Suppose we have 100 seconds, explore 10^4 nodes/second

$\Rightarrow 10^6$ nodes per move $\approx 35^{8/2}$

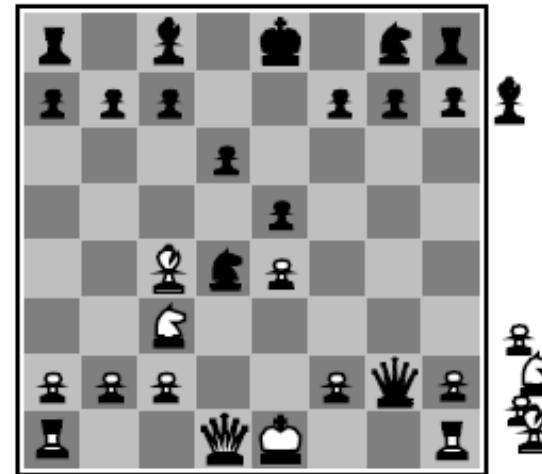
$\Rightarrow \alpha-\beta$ reaches depth 8 \Rightarrow pretty good chess program

Evaluation functions



Black to move

White slightly better



White to move

Black winning

For chess, typically linear weighted sum of features

$$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \dots + w_n f_n(s)$$

e.g., $w_1 = 9$ with

$f_1(s) = (\text{number of white queens}) - (\text{number of black queens})$, etc.

Game Playing – Deficiencies of Minimax

- The bound on the depth of search is artificial and can lead to many anomalies.
- We only consider two:
 1. Non-quiescence
“quiescent” = inactive, quiet, calm, ...
 2. Horizon Effect
- (These deficiencies also apply to alpha-beta as it is just a more efficient way to do the same calculation as minimax)

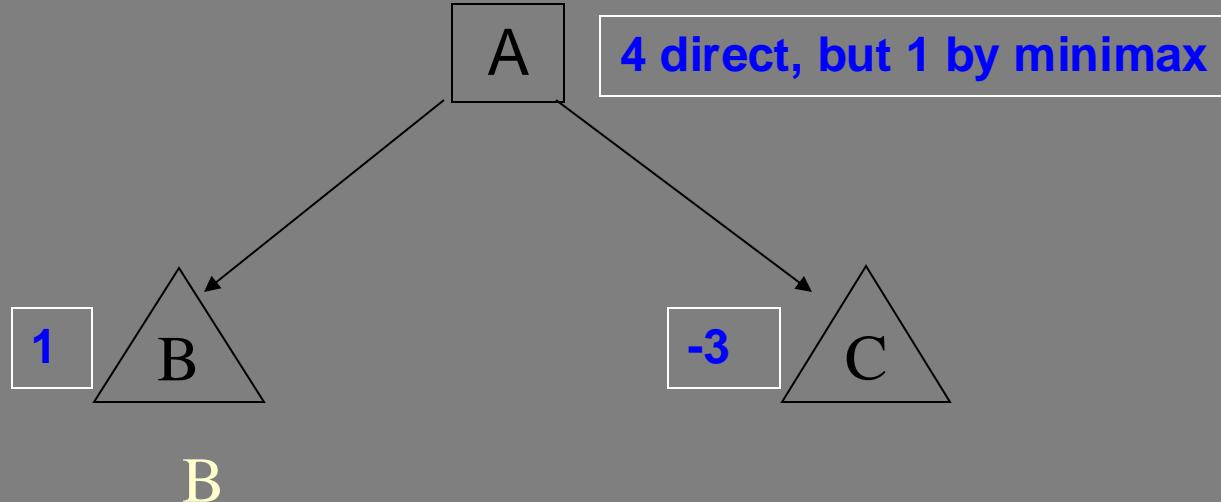
Game Playing – Non-Quiescence

- Suppose that change depth bound from k to $k+1$ – i.e. expand one more move
- The values given to a node might change wildly

Example of non-quiescence

MAX

MIN



Utility values of “terminal” positions obtained by an evaluation function

Direct evaluation does not agree with one more expansion and then using of minimax



= terminal position



= agent



= opponent

Game Playing – Quiescence Search

- Suppose that change depth bound from k to $k+1$ – i.e. expand one more move
- The values given to a node might change wildly
- Keep on increasing the depth bound in that region of the game tree until the values become “quiescent” (“quiet”, i.e. stop being “noisy”)

Game Playing – Quiescence Search

- In quiescence search the depth bound is not applied uniformly but adapted to the local situation
 - in this case so that the values are not wildly changing
- Many other improvements to minimax also work by adapting to depth-bound to get better results and/or do less work

Game Playing – Horizon Effect

- Sometimes there is a bad thing, “X”, such that
 1. X cannot be avoided
 2. X can be delayed by some pointless moves
 3. X is not detected by the evaluation function
- In this case depth-limited minimax can be fooled
- It will use the pointless moves to push X beyond the depth limit, “horizon”, in which case it will “not see X”, and ignore it.
- This can lead the search to take bad moves because it ignores the inevitability of X

Game Playing – Beyond alpha-beta

- **We looked briefly at two problems**
 - “non-quiescence”, and the “horizon effect”
- **and one solution “quiescence search”**
- **To seriously implement a game**
 - Deep-blue, chinook, etc
- **it is necessary to solve many such problems!**
- **Good programs implement many techniques and get them to work together effectively**

Game Playing – Game Classification

- So far have only considered games such as chess, checkers, and nim.
These games are:

1. Fully observable
 - Both players have full and perfect information about the current state of the game
2. Deterministic
 - There is no element of chance
 - The outcome of making a sequence of moves is entirely determined by the sequence itself

Game Playing – Game Classification

- **Fully vs. Partially Observable**
- **Some games are only partially observable**
- **Players do not have access to the full “state of the game”**
- **E.g. card games – you typically cannot see all of your opponents cards**

Game Playing – Game Classification

- **Deterministic vs. Stochastic**
- **In many games there is some element of chance**
- **E.g. Backgammon – throw dice in order to move**
- **(You are expected to be aware of these simple classifications)**

Game Playing – Summary

- **Game Trees**
- **Minimax**
 - utility values propagate back to the root
- **Bounded Depth Minimax**
- **Alpha-Beta Search**
 - uses DFS
 - with depth bound
 - ordering of nodes is important in order to maximise pruning
- **Deficiencies of Bounded Depth Search**
 - Non-quiescence
 - Combat using quiescence search
 - Horizon Problem
 - Combat with ?? (look it up!)

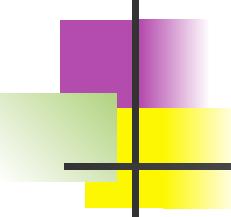
End of Game Playing



Garry Kasparov and Deep Blue. © 1997, GM Gabriel Schwartzman's Chess Camera, courtesy IBM.

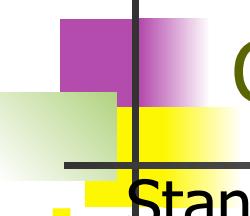
Constraint Satisfaction Problems

Chapter 5
Section 1 – 3



Outline

- Constraint Satisfaction Problems (CSP)
- Backtracking search for CSPs
- Local search for CSPs



Constraint satisfaction problems (CSPs)

- Standard search problem:

- **state** is a "black box" – any data structure that supports successor function, heuristic function, and goal test

- CSP:

- **state** is defined by **variables** X_i with **values** from **domain** D_i
 - **goal test** is a set of **constraints** specifying allowable combinations of values for subsets of variables

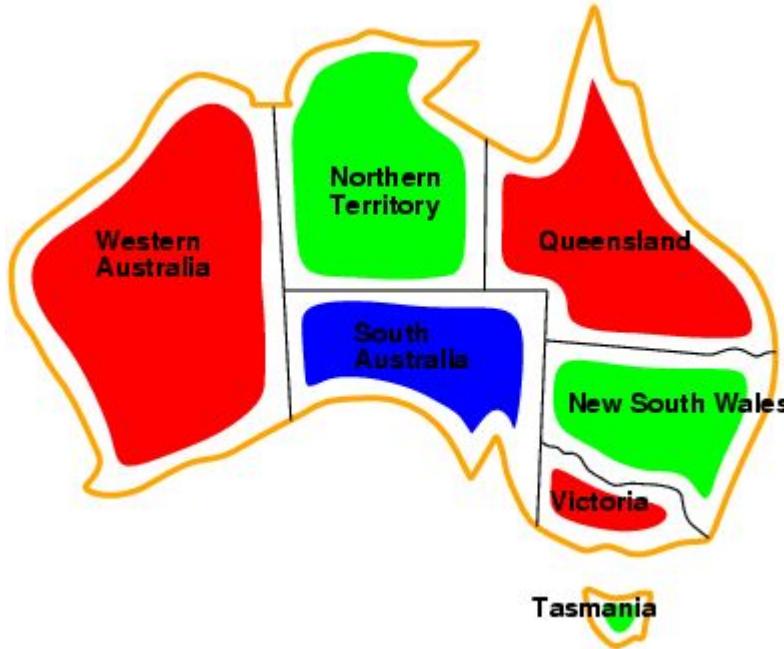
- Simple example of a **formal representation language**
- Allows useful **general-purpose** algorithms with more power than standard search algorithms

Example: Map-Coloring



- **Variables** WA, NT, Q, NSW, V, SA, T
- **Domains** $D_i = \{\text{red,green,blue}\}$
- **Constraints**: adjacent regions must have different colors
- e.g., $WA \neq NT$, or $(WA,NT) \in \{(\text{red,green}),(\text{red,blue}),(\text{green,red}),(\text{green,blue}),(\text{blue,red}),(\text{blue,green})\}$

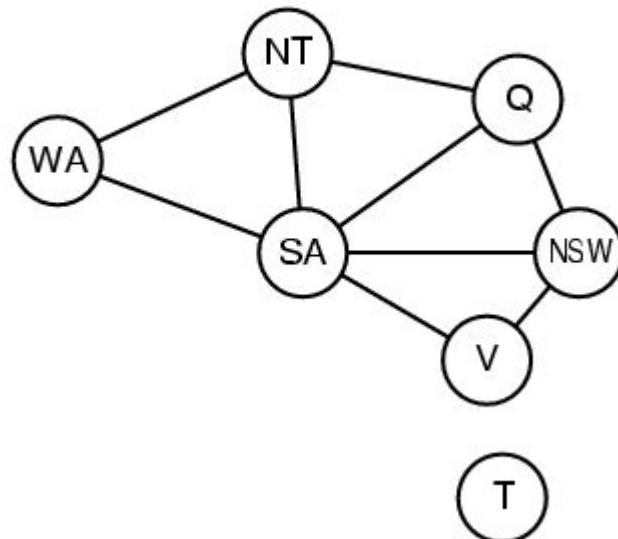
Example: Map-Coloring

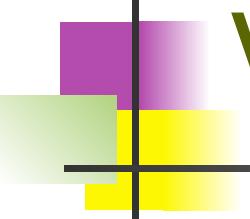


- Solutions are **complete** and **consistent** assignments, e.g., WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = green

Constraint graph

- **Binary CSP:** each constraint relates two variables
- **Constraint graph:** nodes are variables, arcs are constraints





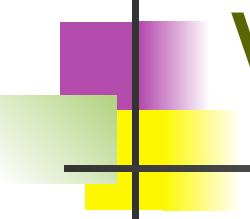
Varieties of CSPs

- Discrete variables

- finite domains:
 - n variables, domain size $d \square O(d^n)$ complete assignments
 - e.g., Boolean CSPs, incl. \sim Boolean satisfiability (NP-complete)
- infinite domains:
 - integers, strings, etc.
 - e.g., job scheduling, variables are start/end days for each job
 - need a constraint language, e.g., $StartJob_1 + 5 \leq StartJob_3$

- Continuous variables

- e.g., start/end times for Hubble Space Telescope observations
- linear constraints solvable in polynomial time by linear programming

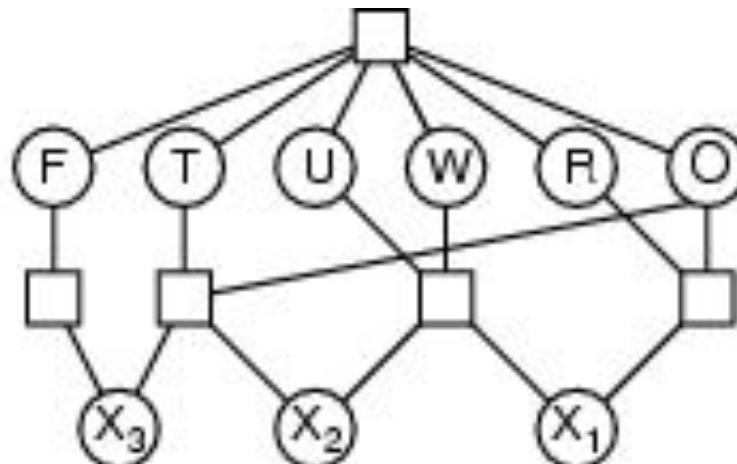


Varieties of constraints

- **Unary** constraints involve a single variable,
 - e.g., SA \neq green
- **Binary** constraints involve pairs of variables,
 - e.g., SA \neq WA
- **Higher-order** constraints involve 3 or more variables,
 - e.g., cryptarithmetic column constraints

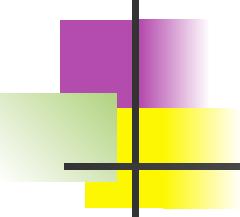
Example: Cryptarithmetic

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



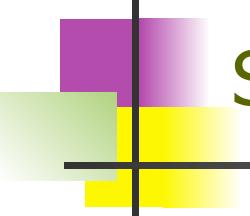
- **Variables:** $F, T, U, W, R, O, X_1, X_2, X_3$
- **Domains:** $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
- **Constraints:** $\text{Alldiff}(F, T, U, W, R, O)$

- $O + O = R + 10 \cdot X_1$
- $X_1 + W + W = U + 10 \cdot X_2$
- $X_2 + T + T = O + 10 \cdot X_3$
- $X_3 = F, T \neq 0, F \neq 0$



Real-world CSPs

- Assignment problems
 - e.g., who teaches what class
- Timetabling problems
 - e.g., which class is offered when and where?
- Transportation scheduling
- Factory scheduling
- Notice that many real-world problems involve real-valued variables

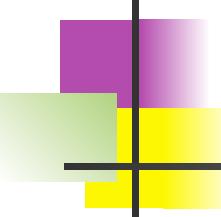


Standard search formulation (incremental)

Let's start with the straightforward approach, then fix it

States are defined by the values assigned so far

- **Initial state:** the empty assignment { }
 - **Successor function:** assign a value to an unassigned variable that does not conflict with current assignment
 - fail if no legal assignments
 - **Goal test:** the current assignment is complete
1. This is the same for all CSPs
 2. Every solution appears at depth n with n variables
 - use depth-first search
 3. Path is irrelevant, so can also use complete-state formulation
 4. $b = (n - \ell)d$ at depth ℓ , hence $n! \cdot d^n$ leaves



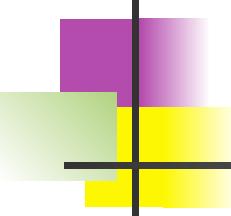
Backtracking search

- Variable assignments are **commutative**, i.e.,
[WA = red then NT = green] same as [NT = green then WA = red]
- Only need to consider assignments to a single variable at each node
 - $b = d$ and there are d^n leaves
- Depth-first search for CSPs with single-variable assignments is called **backtracking** search
- Backtracking search is the basic uninformed algorithm for CSPs
- Can solve n -queens for $n \approx 25$

Backtracking search

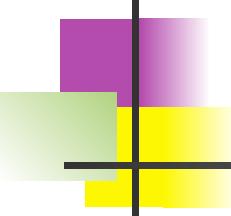
```
function BACKTRACKING-SEARCH( csp) returns a solution, or failure
  return RECURSIVE-BACKTRACKING( {}, csp)

function RECURSIVE-BACKTRACKING( assignment, csp) returns a solution, or
failure
  if assignment is complete then return assignment
  var  $\leftarrow$  SELECT-UNASSIGNED-VARIABLE( Variables[csp], assignment, csp)
  for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
    if value is consistent with assignment according to Constraints[csp] then
      add { var = value } to assignment
      result  $\leftarrow$  RECURSIVE-BACKTRACKING(assignment, csp)
      if result  $\neq$  failure then return result
      remove { var = value } from assignment
  return failure
```

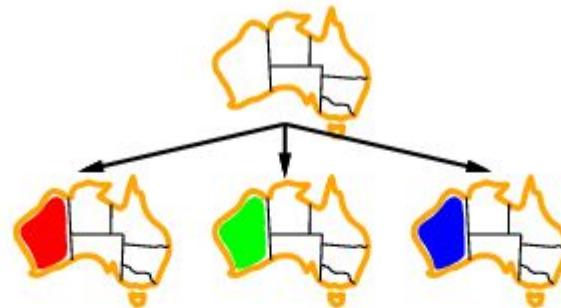


Backtracking example

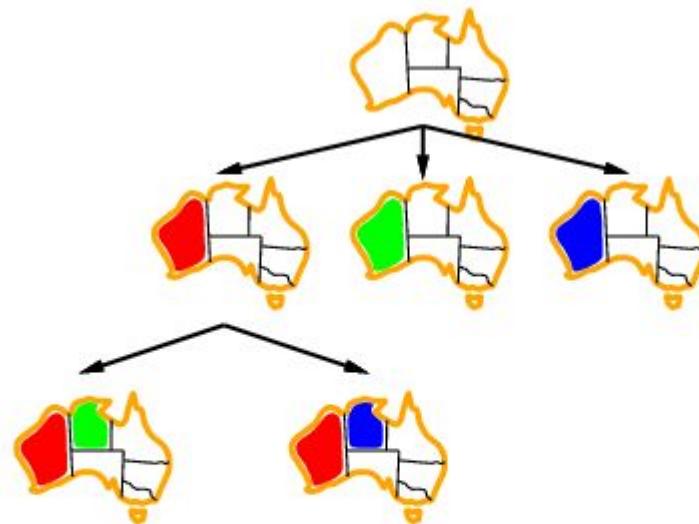




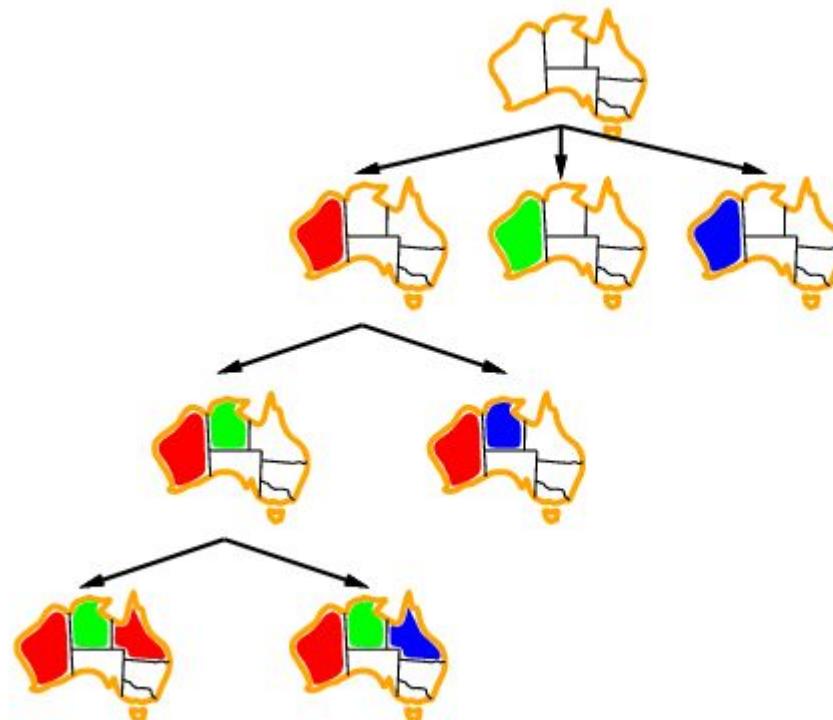
Backtracking example

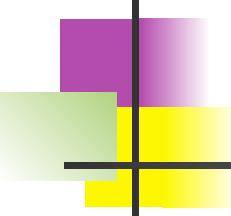


Backtracking example



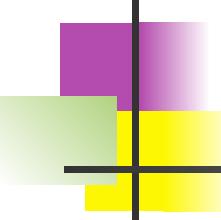
Backtracking example





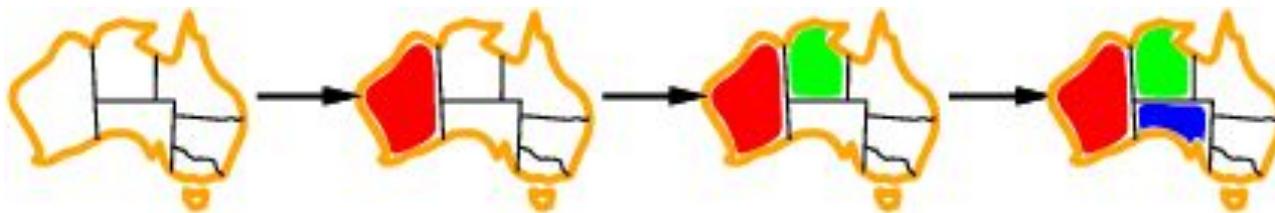
Improving backtracking efficiency

- **General-purpose** methods can give huge gains in speed:
 - Which variable should be assigned next?
 - In what order should its values be tried?
 - Can we detect inevitable failure early?

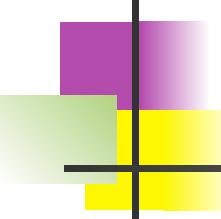


Most constrained variable

- Most constrained variable:
choose the variable with the fewest legal values

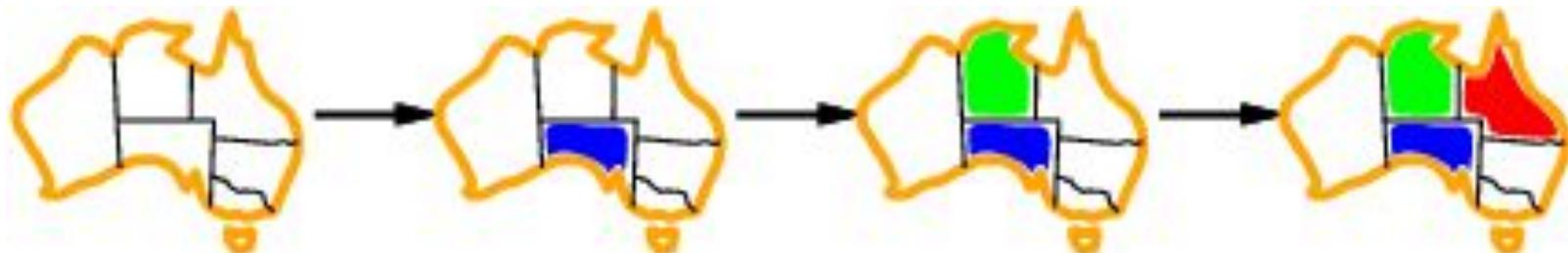


- a.k.a. **minimum remaining values (MRV)** heuristic



Most constraining variable

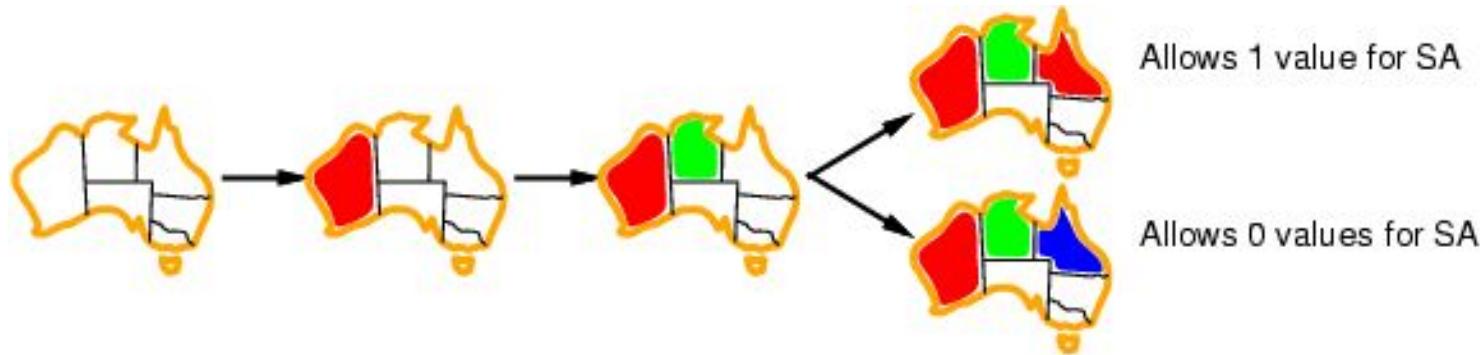
- Tie-breaker among most constrained variables
- Most constraining variable:
 - choose the variable with the most constraints on remaining variables



Least constraining value

- Given a variable, choose the least constraining value:

- the one that rules out the fewest values in the remaining variables

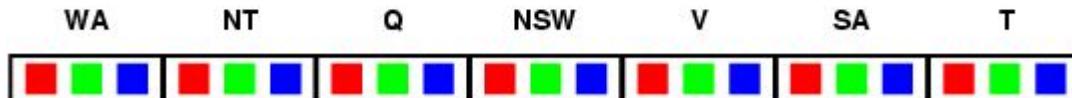


- Combining these heuristics makes 1000 queens feasible

Forward checking

- **Idea:**

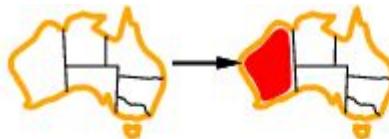
- Keep track of remaining legal values for unassigned variables
- Terminate search when any variable has no legal values



Forward checking

- **Idea:**

- Keep track of remaining legal values for unassigned variables
- Terminate search when any variable has no legal values



WA	NT	Q	NSW	V	SA	T
■ Red	■ Green	■ Blue	■ Red	■ Green	■ Blue	■ Red
■ Red	■ Green	■ Blue	■ Red	■ Green	■ Blue	■ Red

Forward checking

Idea:

- Keep track of remaining legal values for unassigned variables
- Terminate search when any variable has no legal values

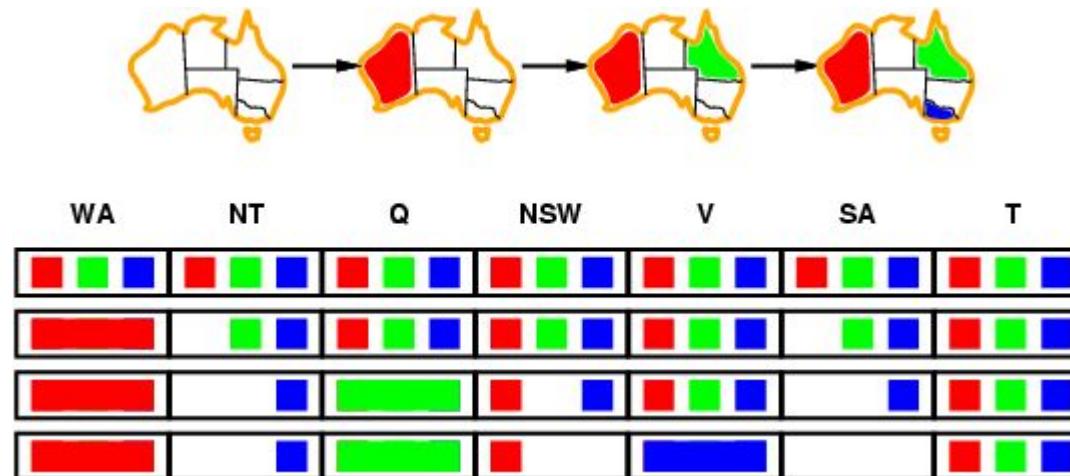


WA	NT	Q	NSW	V	SA	T
■■■■■	■■■■■	■■■■■	■■■■■	■■■■■	■■■■■	■■■■■
■■■■■	■■■■■	■■■■■	■■■■■	■■■■■	■■■■■	■■■■■
■■■■■	■■■■■	■■■■■	■■■■■	■■■■■	■■■■■	■■■■■

Forward checking

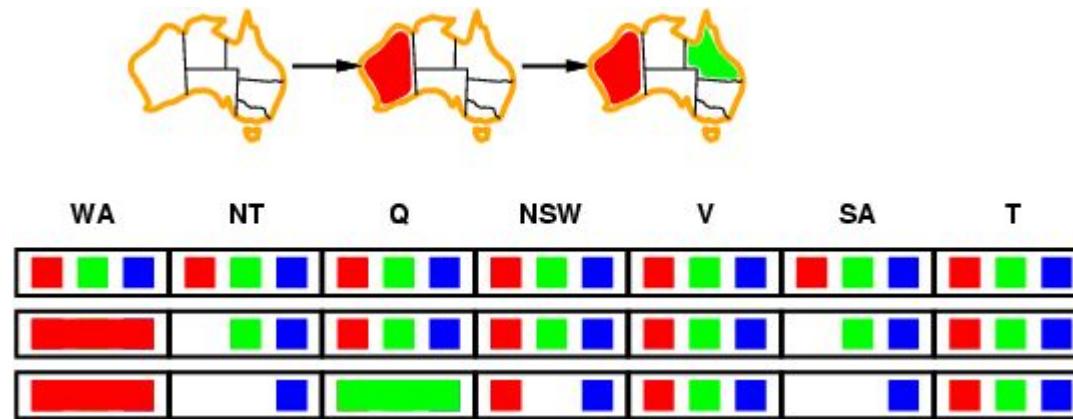
- Idea:

- Keep track of remaining legal values for unassigned variables
- Terminate search when any variable has no legal values



Constraint propagation

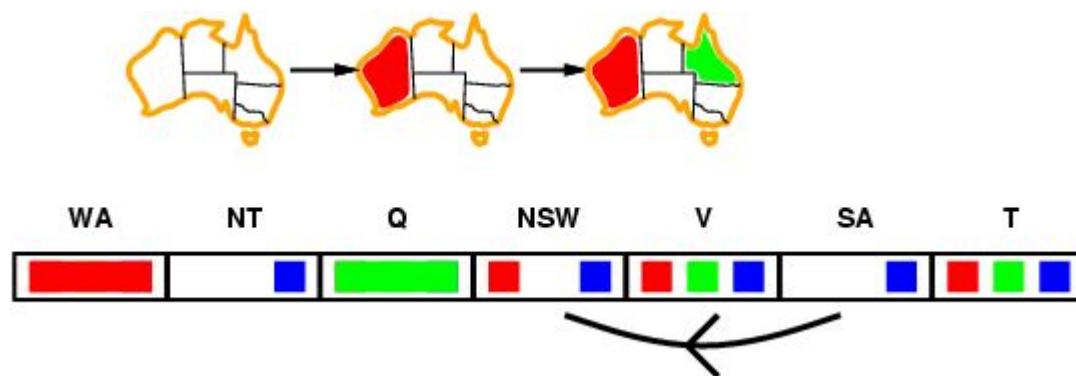
- Forward checking propagates information from assigned to unassigned variables, but doesn't provide early detection for all failures:



- NT and SA cannot both be blue!
- Constraint propagation repeatedly enforces constraints locally

Arc consistency

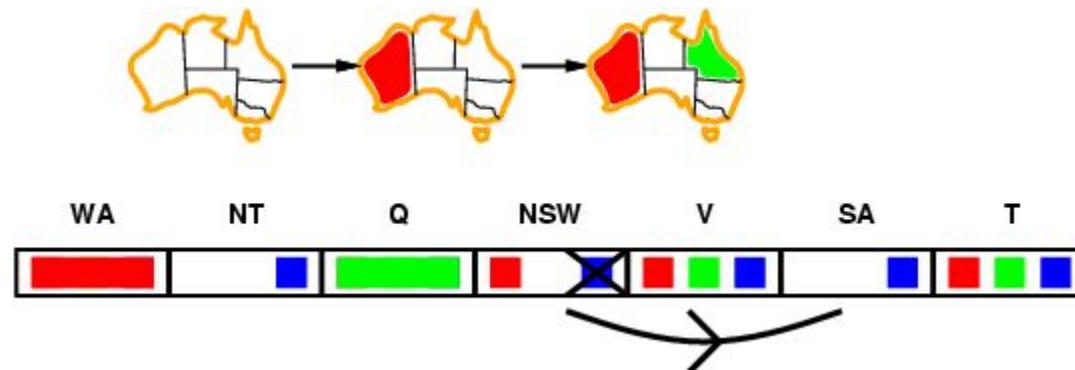
- Simplest form of propagation makes each arc **consistent**
- $X \square Y$ is consistent iff
 - for **every** value x of X there is **some** allowed y



Arc consistency

- Simplest form of propagation makes each arc **consistent**
- $X \square Y$ is consistent iff

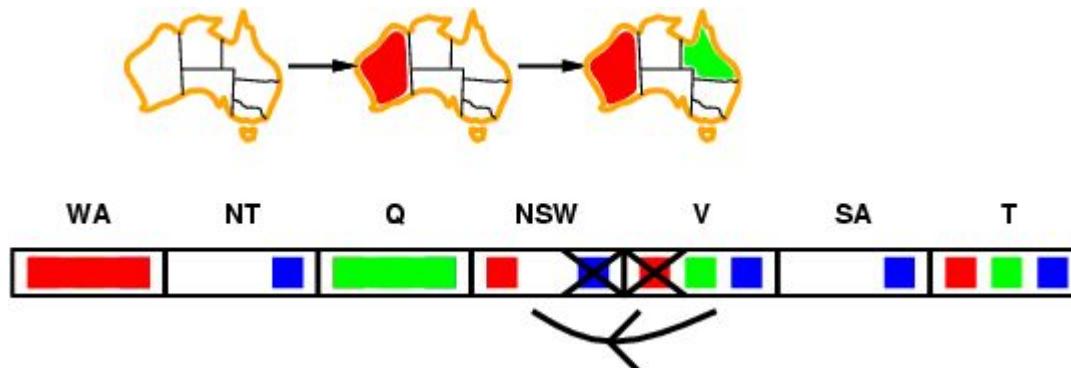
for **every** value x of X there is **some** allowed y



Arc consistency

- Simplest form of propagation makes each arc **consistent**
- $X \square Y$ is consistent iff

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

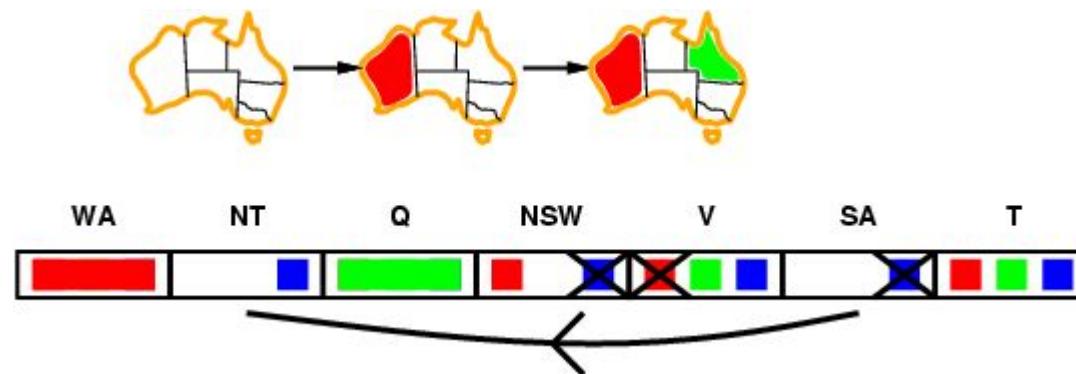


- If X loses a value, neighbors of X need to be rechecked

Arc consistency

- Simplest form of propagation makes each arc **consistent**
- $X \square Y$ is consistent iff

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



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

Arc consistency algorithm AC-3

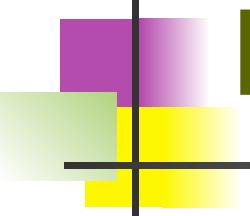
```
function AC-3(csp) returns the CSP, possibly with reduced domains
  inputs: csp, a binary CSP with variables  $\{X_1, X_2, \dots, X_n\}$ 
  local variables: queue, a queue of arcs, initially all the arcs in csp
  while queue is not empty do
     $(X_i, X_j) \leftarrow \text{REMOVE-FIRST}(\text{queue})$ 
    if RM-INCONSISTENT-VALUES( $X_i, X_j$ ) then
      for each  $X_k$  in NEIGHBORS[ $X_i$ ] do
        add  $(X_k, X_i)$  to queue



---


function RM-INCONSISTENT-VALUES( $X_i, X_j$ ) returns true iff remove a value
  removed  $\leftarrow \text{false}$ 
  for each  $x$  in DOMAIN[ $X_i$ ] do
    if no value  $y$  in DOMAIN[ $X_j$ ] allows  $(x, y)$  to satisfy constraint( $X_i, X_j$ )
      then delete  $x$  from DOMAIN[ $X_i$ ]; removed  $\leftarrow \text{true}$ 
  return removed
```

- Time complexity: $O(n^2d^3)$

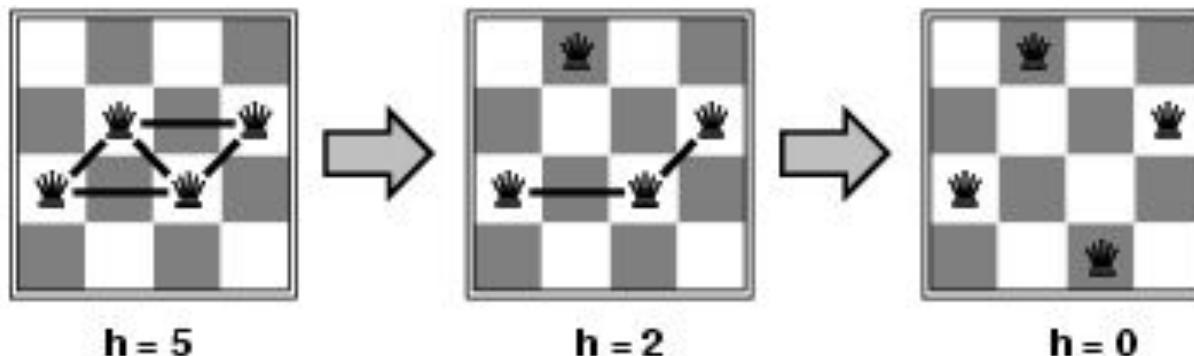


Local search for CSPs

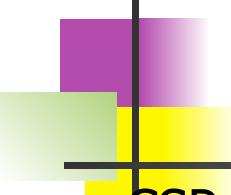
- Hill-climbing, simulated annealing typically work with "complete" states, i.e., all variables assigned
- To apply to CSPs:
 - allow states with unsatisfied constraints
 - operators **reassign** variable values
- Variable selection: randomly select any conflicted variable
- Value selection by **min-conflicts** heuristic:

Example: 4-Queens

- States: 4 queens in 4 columns ($4^4 = 256$ states)
Actions: move queen in column
Goal test: no attacks
Evaluation: $h(n)$ = number of attacks



- Given random initial state, can solve n -queens in almost constant time for arbitrary n with high probability (e.g., $n = 10,000,000$)



Summary

- CSPs are a special kind of 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