

CSCI 561

Foundation for Artificial Intelligence

Welcome!

Professor Wei-Min Shen

University of Southern California

Lectures and Discussions

- Lectures: 5:00-7:20PM
 - Introduction
 - Who are us? Who are you?
 - What will we learn and do this semester?
 - Course details: lectures, discussions, homeworks, exams, grades, ...
 - Overview of Artificial Intelligence, and Basics of Search
 - Intelligence, Agents, and Artificial Intelligence
 - Agent Types and Rational Actions
 - Problem Solving, Search Space and Algorithms
- Discussions: 7:30-8:20PM
 - What have AI accomplished?
 - How to measure intelligence?

CS 561: Artificial Intelligence

Instructor: Professor Wei-Min Shen (wmshen@usc.edu)

Office Hours: after lectures (by email appointment)

TAs and CPs: Please see DEN and Piazza

Office Hours and Zooms: please see Piazza

Lectures: DEN WebEx, 5:00 – 7:20PM, **Wednesday**

Discussion: DEN WebEx, 7:30 – 8:20PM, **Wednesday**

This class will use **courses.uscden.net** (Desire2Learn, D2L)

- Up to date information, lecture notes, lecture videos
- Homework posting and submission information
- Grades, relevant dates, links, etc.

Textbook: [AIMA] Artificial Intelligence: A Modern Approach, by Russell & Norvig. (4th ed)

Optional: [ALFE] Autonomous Learning from the Environment, by Shen

CS 561: Artificial Intelligence

Course overview: foundations of symbolic intelligent systems. Agents, search, problem solving, logic, representation, reasoning, symbolic programming, machine learning, and robotics

Prerequisites: CS 455x, i.e., programming principles, discrete mathematics for computing, software design and software engineering concepts.

Good knowledge for (1) algorithm design, and (2) programming languages such as Python (preferred), and others (Java, C++ or C) are needed for homework

Grading:

- 20% for midterm-1 exam
- 20% for midterm-2 exam
- 30% for final exam
- 30% for 3 mandatory homework assignments

CS 561: Artificial Intelligence

Grading:

Grading is absolute and according to the following scale:

≥ 90 A+ (honorary – shows as A on transcript)

≥ 80 A

≥ 75 A-

≥ 70 B+

≥ 60 B

≥ 55 B-

≥ 50 C+

≥ 40 C

≥ 35 C-

< 35 F

Practical Issues

- **Class mailing list:** has been setup on the D2L system and Piazza.
- **Homework:** See class web page on D2L, and they are take-home programming assignments
 - Week-2 – HW1 out Topic: search or function optimization
 - Week-5 – HW1 due
 - Week-5 – HW2 out Topic: game playing or constraint satisfaction
 - Week-11 – HW2 due
 - Week-11 – HW3 out Topic: neural networks or logic reasoning/inference
 - Week-14 – HW3 due
- **Late homework:** you lose 20% of the homework's grade per 24-hour period that you are late. Beware, the penalty grows very fast: $\text{grade} = \text{points} * (1 - n * 0.2)$ where n is the number of days late (n=0 if submitted on time, n=1 if submitted between 1 second and 24h late, etc).
- **Homework grading:** your work will be graded by an A.I. agent (given to you in advance for testing) through the online system at vocareum.com.
- **Grade review / adjustment:** Requests will be considered up to 2 weeks after the grade is released. After that, it will be too late and requests for grading review will be ignored.
- **Exams:**
 - Week-6 – midterm 1 (5-7PM, online)
 - Week-10 – midterm 2 (5-7PM, online)
 - Week-16 – final exam (5-7PM, online)

**Syllabus
and
Schedules**

Date	Topic	Reading
Week-1	1. Welcome – Introduction. Why study AI? What is AI? The Turing test. Rationality. Branches of AI. Brief history of AI. Challenges for the future. What is an intelligent agent? Doing the right thing (rational action)? Performance measures. Autonomy. Environment and agent. Types of agents.	AIMA 1, 2 (ALFE 1)
	2. Problem Solving & Search – Types of problems. Example problems. Basic idea behind search algorithms. Complexity. Combinatorial explosion and NP completeness. Polynomial hierarchy.	AIMA 3 (ALFE 2, 6)
Week-2	3. Uninformed Search - Depth-first. Breadth-first. Uniform-cost. Depth-limited. Iterative deepening. Examples. Properties.	AIMA 3 HW1 out
	4. Informed Search – Best-First, A* search, Heuristics, Hill climbing, Problem of local extrema, Simulated annealing, and Genetic Algorithms.	AIMA 3, 4 (ALFE 6)
Week-3	5. Constraint satisfaction. Node, arc, path, and k-consistency. Backtracking search. Local search using min-conflicts.	AIMA 6
	6. Game Playing - The minimax algorithm. Resource limitations. Alpha-beta pruning. Chance and non-deterministic games.	AIMA 5
Week-4	7. Advanced Game Playing - Agent Interaction with environment and other agents. 8. Reinforcement Learning.	AIMA 21 ALFE 6.1
Week-5	9. Agents that reason logically 1 – Knowledge-based agents. Logic and representation. Propositional (Boolean) logic.	AIMA 7 (ALFE 3) HW1 due
	10. Agents that reason logically 2 – Inference in propositional logic. Syntax. Semantics. Examples.	AIMA 7 HW2 out
Week-6	Midterm exam 1 (no make-up exam)	

Syllabus and Schedules	Week-7	11. First-order logic 1 – Syntax. Semantics. Atomic sentences. Complex sentences. Quantifiers. Examples. FOL knowledge base. Situation calculus.	AIMA 8, AIMA 12
		12. First-order logic 2 – Describing actions. Planning. Action sequences.	AIMA 8
	Week-8	13. Inference in first-order logic – Proofs. Unification. Generalized modus ponens. Forward and backward chaining.	AIMA 9
		14. Continue Inference in first-order logic. Resolution. Proof by contradiction.	AIMA 9
	Week-9	15. Logical reasoning systems – Indexing, retrieval, and unification. The Prolog language. Theorem provers. Frame systems and semantic networks.	AIMA 9
		16. Planning – Definition and goals. Basic representations for planning. Situation space and plan space. Examples.	AIMA 10 (ALFE 6)
	Week-10	Midterm exam 2 (no make-up exam)	HW2 due HW3 out

Syllabus and Schedules	Week-11	17. Learning from examples – supervised learning, learning decision trees, support vector machines.	AIMA 18 Handout
		18. Learning with neural networks – Perceptron, Hopfield networks. How to size a network? What can neural networks achieve? Deep learning and state of the art.	AIMA 18 Handout
	Week-12	19. Reasoning under uncertainty – probabilities, conditional independence, Markov blanket, Bayes nets.	AIMA 13, 14
		20. Reasoning under uncertainty – Probabilistic inference, enumeration, variable elimination, approximate inference by stochastic simulation, Markov chain Monte Carlo, Gibbs sampling.	AIMA 14, 15 (ALFE 5)
	Week-13	21. Probabilistic decision making – utility theory, decision networks, value iteration, policy iteration, Markov decision processes (MDP), partially observable MDP (POMDP).	AIMA 16, 17 (ALFE 5)
		22. Probabilistic Reasoning over time: Temporal models, Hidden Markov Models, Kalman filters, Dynamic Bayesian Networks, Automata theory.	AIMA15
	Week-14	11/22-26 Thanksgiving Break	HW3 due
	Week-15	23. Probability-Based Learning: Probabilistic Models, Naive Bayes Models, EM algorithm, Reinforcement Learning.	AIMA 20 (ALFE 5.10)
		24. Towards intelligent machines – The challenge of robots: with what we have learned, what hard problems remain to be solved? Different types and architectures of robots.	AIMA 24,26-27 (ALFE 13)
	Week-16	Final Exam (Time to be set by the University/School) (no make-up exam)	

More on Homework and Grading

- In each homework you will implement some algorithms **from scratch by yourself**
- But our goal is to focus on A.I. algorithms, not on low-level programming. Hence, we recommend Python (PREFERRED), and others (Java, or C++/STL) so that you can use the STL containers (queue, map, etc) instead of pointers and memory management. But the language you use is up to you
- Code editing, compiling, testing: we will use www.vocareum.com which will be accessible for you
 - Please learn how to use the terminal window how to debug your code on Vocareum
- Vocareum supports many languages of your choice: Python (preferred), Java, C++, C++11, C, etc.
- Your program should take no command-line arguments. It should read a text file called "input.txt" that contains a problem definition, and write a file "output.txt" with your solution. For each homework, format for files input.txt and output.txt will be specified and examples will be given to you. Please follow the exact format or else the grading script will fail your code.
 - The grading will test your code on 40-50 test cases:
 - Create an input.txt file
 - Run your code
 - Compare output.txt created by your program with the correct one
 - If your outputs for all test cases are correct, you get **100** points
 - If one or more test case fails, you will get deductions based on the difficulty-level of the test cases

Vocareum.com (subject to upgrades or changes)

gcc - 5.5

valgrind - 3.11

ar - 2.26.1

php - 7.0.32

python2 - 2.7.12

python3 - 3.5.2, 3.6.4 (preferred)

Java 1.8 - 1.8.0_191

perl - 5.22.1

make - 4.1

Discussion Sections, TA Office Hours

- You must register the discussion section. Contents there will be tested in the exams
- Discussion sections will
 - Provide more details, discussion and examples on complex topics
 - Run algorithms on more complex examples than during lectures
 - Relate lecture concepts to latest research topics
 - Showcase cool demos of recent A.I. achievements
- TA/CourseProducer/Grader Office Hours
 - see announcements on DEN and Piazza
 - Please attend them, and we have **excellent TAs/CPs** for you

Your Teaching Team This Semester

please see On Piazza: /resources/staff/

For their office hours and Zoom/locations

Piazza

- We will use www.piazza.com for questions and answers related to class material
- Please register by clicking on the "Piazza" button on DEN (this is **mandatory**)
- Guidelines:
 - Be polite, professional, clear, and precise, and please don't throw tantrum ☺
 - You may ask any question related to material covered in lectures, discussions, or exams
 - You may ask **clarification questions only** related to the homework definitions
 - You may **not** ask for advice on how to solve some aspect of a homework problem
 - You may **not** post code snippets related to homework problems
 - You may **not** post test cases or input/output examples related to homework problems
 - Please remember that homework assignments are to be solved strictly individually

Academic Integrity

- Familiarize yourself with the USC Academic Integrity guidelines.
- Violations of the Student Conduct Code will be filed with the Office of Student Judicial Affairs, and appropriate sanctions will be given.
- Homework assignments are to be solved **individually**.
- You are welcome to discuss class material in review groups, but do not discuss how to solve the homeworks.
- **Exams are closed-book with no questions allowed.**
- **Please read and understand:**

<http://policy.usc.edu/student/scampus/>

<https://sjacs.usc.edu/students/>

Academic Integrity

- **All students are responsible for reading and following the Student Conduct Code.**
Note that the USC Student Conduct Code prohibits plagiarism.
- Some examples of what is not allowed by the conduct code: copying all or part of someone else's work (by hand or by looking at others' files, either secretly or if shown), and submitting it as your own; giving another student in the class a copy of your assignment solution; and consulting with another student during an exam. If you have questions about what is allowed, please discuss it with the instructor.
- Students who violate university standards of academic integrity are subject to disciplinary sanctions, including failure in the course and suspension from the university. Since dishonesty in any form harms the individual, other students, and the university, policies on academic integrity will be strictly enforced. Violations of the Student Conduct Code will be filed with the Office of Student Judicial Affairs.

Academic Integrity

USC

UNIVERSITY
OF SOUTHERN
CALIFORNIA

October 5, 2007

M [REDACTED]

Case # [REDACTED]

Los Angeles, CA [REDACTED]

Dear M [REDACTED]

Division of
Student Affairs

Student Judicial
Affairs and
Community Standards

I have received a report from Professor Itti Engineering, concerning an alleged act of academic dishonesty which occurred in CSCI-561 (#30219) during the Fall Semester (2007).

Specifically, the complaint alleges that you violated Student Conduct Code §§:

11.12A Acquisition of term papers or other assignments from any source and the subsequent presentation of those materials as the student's own work, or providing term papers or assignments that another student submits as his/her own.

11.14B Unauthorized collaboration on a project, homework or other assignment.

Collaboration between students will be considered unauthorized unless expressly part of the assignment in question or expressly permitted by the instructor.

11.15A Attempting to benefit from the work of another or attempting to hinder the work of another student.

11.15B Any act which may jeopardize another student's academic standing

11.15B Any act which may jeopardize another student's academic standing

11.21 Any act which gains or is intended to gain an unfair academic advantage may be considered an act of academic dishonesty.

The complaint concerns your assignment completed on or about September 26, 2007.

As a consequence of the complaint a review of the allegations is necessary. The guidelines for the review process (summary enclosed) can be found in the Student Conduct Code in the current *SCampus*. Please familiarize yourself with the standards and expectations concerning academic honesty prior to our meeting and the review. According to University policy, you will not be permitted to drop the course with a mark of 'W' (see enclosure).

Please contact the Office of Student Judicial Affairs and Community Standards at (213) 821-7373 to schedule a meeting with me and for a review of the matter. If you do not respond by October 19, 2007, an administrative hold may be placed on your record prohibiting further registration and enrollment transactions. A review also may be conducted in your absence should you choose not to respond.

Sincerely,

Raquel Torres-Retana

Director, Office of Student Judicial Affairs and Community Standards

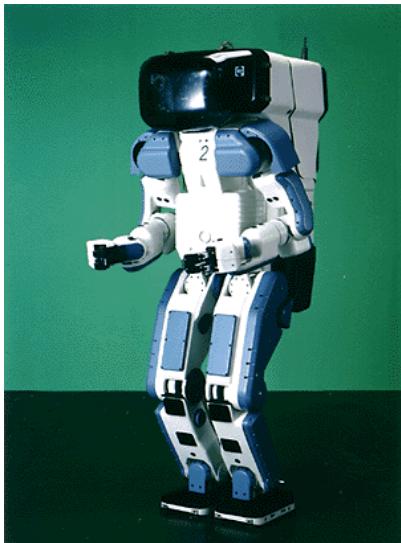
cc: Professor Laurent Itti
Kelly Goulios, Viterbi School of Engineering

Enclosures

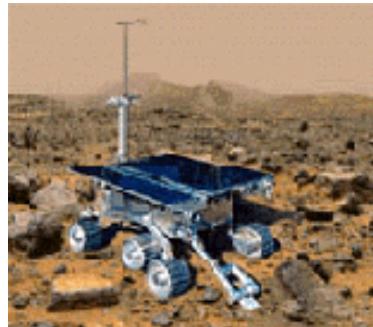
University of
Southern California
Figueroa Building
Room 107
Los Angeles,
California 90089-1265
Tel: 213 821 7373
Fax: 213 740 7152

GENERAL KNOWLEDGE OF ARTIFICIAL INTELLIGENCE

Why Study AI?



Labor



Science



Appliances /
Internet of Things (IoT)



Search engines

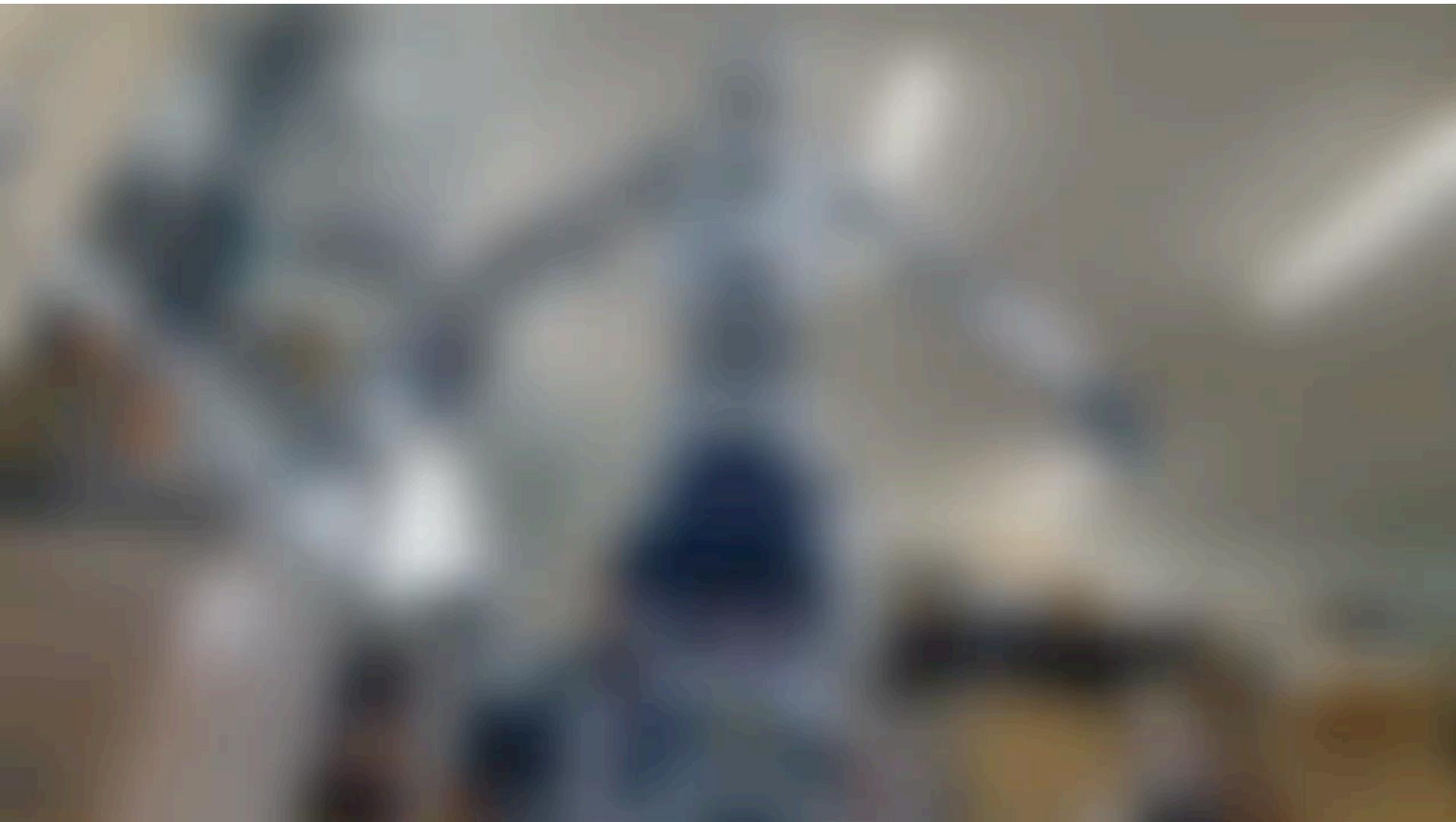


Medicine /
Diagnosis

What else?

What Could AI Do Now?





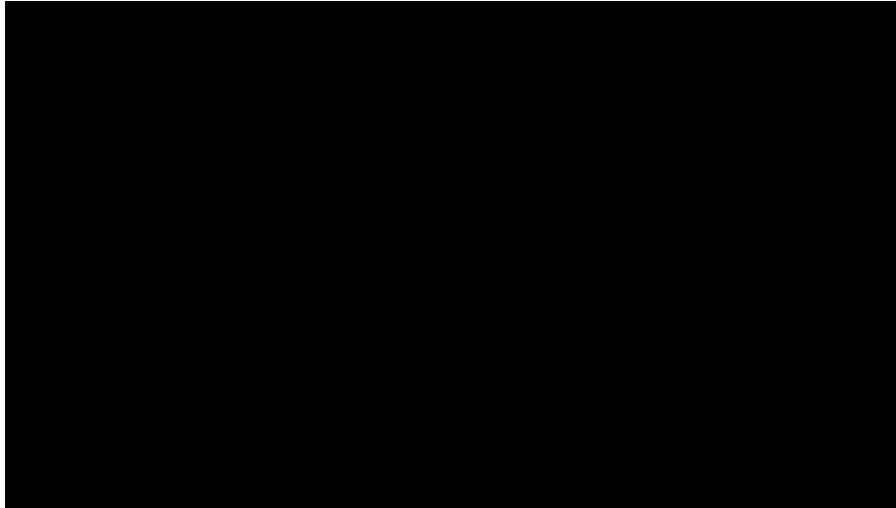
DARPA Robotics Challenge



drc-laíst-winner



Wearable computing



Google glass



Microsoft Hololens



Zypad



The Perfect Toy for Billionaires (2023)





The perfect toy for billionaires – This \$2.75 million giant mech robot from Japan is a real-world transformer bot with a cockpit. It is almost 15 feet tall, weighs 3.5 tons, has func...

Luxurylaunches

Apple News

HOW TO MEASURE “INTELLIGENCE” ?!

What is AI? Two Basic Views

The exciting new effort to make computers think ... machines with minds, in the full and literal sense"
(Haugeland 1985)

"The art of creating machines that perform functions that require intelligence when performed by people" (Kurzweil, 1990)

Systems that think like humans

Systems that act **like humans**

"The study of mental faculties through the use of computational models"
(Charniak et al. 1985)

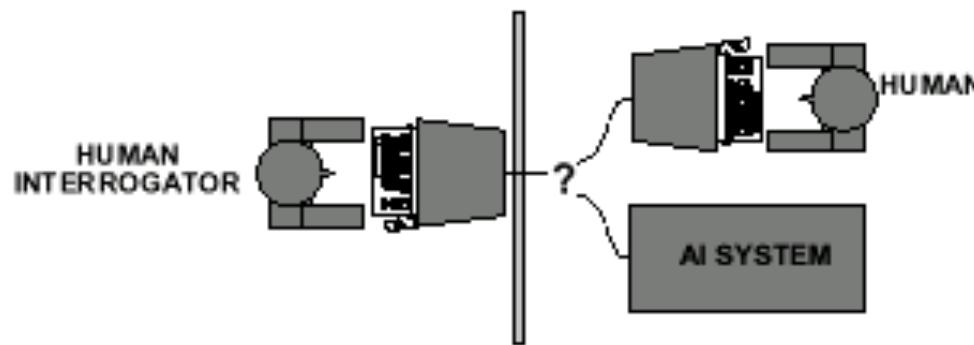
A field of study that seeks to explain and emulate intelligent behavior in terms of computational processes" (Schalkol, 1990)

Systems that think rationally

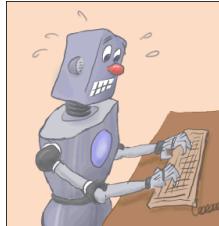
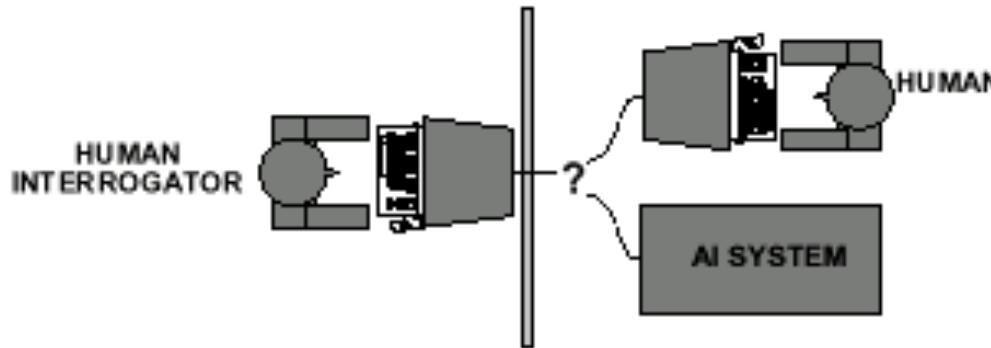
Systems that act rationally

Acting Humanly: The (basic) Turing Test

- Alan Turing's 1950 article Computing Machinery and Intelligence discussed conditions for considering a machine to be intelligent
 - "Can machines think?" ↔ "Can machines behave intelligently?"
 - The Turing test (The Imitation Game): Operational definition of intelligence.



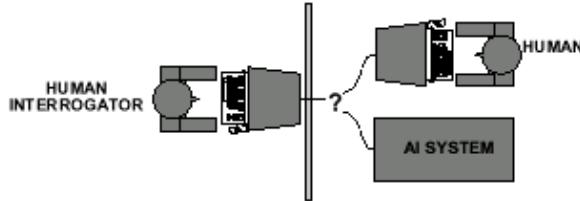
Acting Humanly: The (basic) Turing Test



- Computer needs to possess: Natural language processing, Knowledge representation, Automated reasoning, and Machine learning
- Are there any problems/limitations to the Turing Test?

Acting Humanly: The Full or Total Turing Test

- Alan Turing's 1950 article Computing Machinery and Intelligence discussed conditions for considering a machine to be intelligent
 - "Can machines think?" \leftrightarrow "Can machines behave intelligently?"
 - The Turing test (The Imitation Game): Operational definition of intelligence.

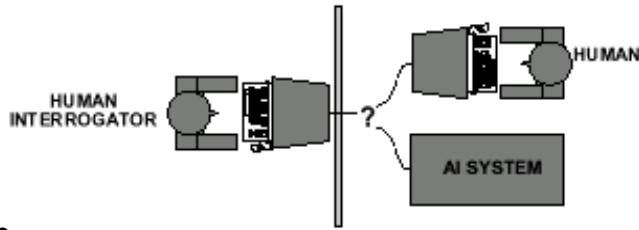


- Computer needs to possess: Natural language processing, Knowledge representation, Automated reasoning, and Machine learning
- **Problem:** 1) Turing test is not reproducible, constructive, and amenable to mathematical analysis. 2) What about physical interaction with interrogator and environment?
- **Total Turing Test:** Requires physical interaction and needs perception and actuation.

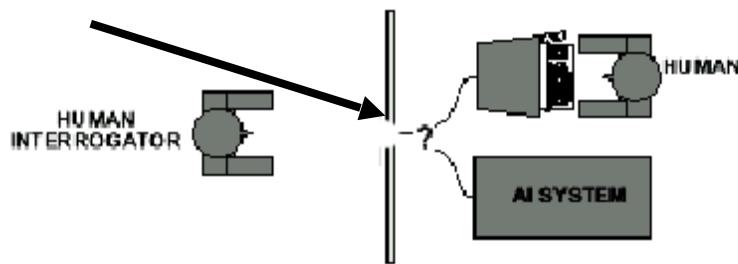
Acting Humanly: The Full Turing Test

Problem:

- 1) Turing test is not reproducible, constructive, and amenable to mathematic analysis.
- 2) What about physical interaction with interrogator and environment?



Trap door



What would a computer need to pass the Basic Turing test?

- **Natural language processing:** to communicate with examiner.
- **Knowledge representation:** to store and retrieve information provided before or during interrogation.
- **Automated reasoning:** to use the stored information to answer questions and to draw new conclusions.
- **Machine learning:** to adapt to new circumstances and to detect and extrapolate patterns.



What would a computer need to pass the Basic Turing test?

- Natural language processing: to communicate with examiner.
- Knowledge representation: to store and retrieve information provided before or during interrogation.
- Automated reasoning: to use the stored information to answer questions and to draw new conclusions.
- Machine learning: to adapt to new circumstances and to detect and extrapolate patterns.

Core focus in this course



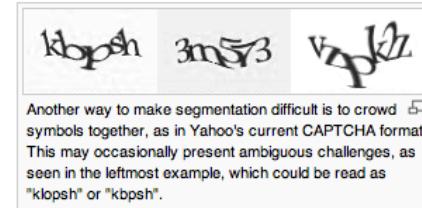
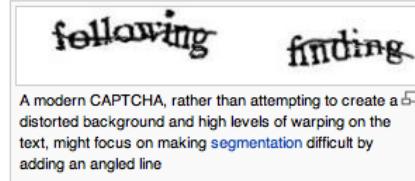
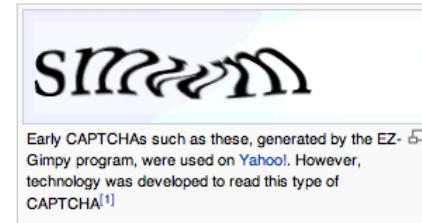
What would a computer need to pass the Full/Total Turing test?

- **Vision** (for Total Turing test): to recognize the examiner's actions and various objects presented by the examiner.
- **Motor control** (total test): to act upon objects as requested.
- **Other senses** (total test): such as audition, smell, touch, etc.

CAPTCHAs or “Reverse Turing Tests”

- Vision is a particularly difficult one for machines...
- Gave rise to “Completely Automated Public Turing test to tell Computers and Humans Apart” (CAPTCHA)

wikipedia



Acting Rationally: The Rational Agent

- Another measurement of Intelligence (other than human-like)!
- Rational behavior: Doing the right thing!
- The right thing: That which is expected to maximize the expected return
- Provides the most general view of AI because it includes:
 - Correct inference ("Laws of thought")
 - Uncertainty handling
 - Resource limitation considerations (e.g., reflex vs. deliberation)
 - Cognitive skills (NLP, AR, knowledge representation, ML, etc.)
- Advantages:
 - 1) More general
 - 2) Its goal of rationality is well defined (you will learn that this semester)

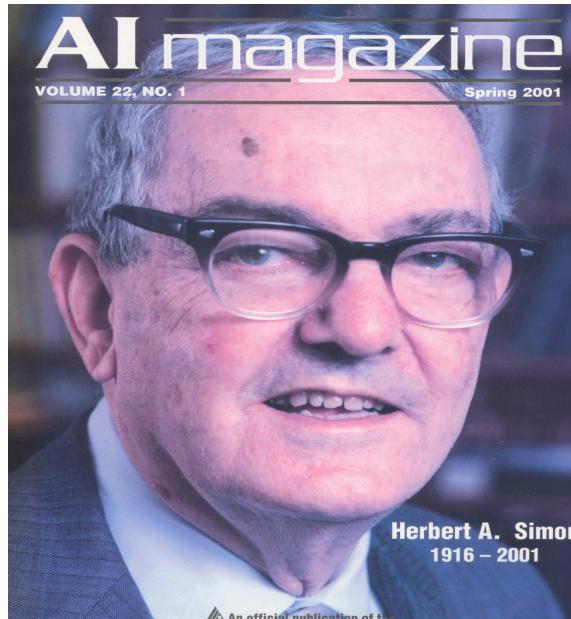
AI Prehistory

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
Psychology	adaptation phenomena of perception and motor control experimental techniques (psychophysics, etc.)
Linguistics	knowledge representation grammar
Neuroscience	physical substrate for mental activity
Control theory	homeostatic systems, stability simple optimal agent designs

AI History

- 1943 McCulloch & Pitts: Boolean circuit model of brain
- 1950 Turing's "Computing Machinery and Intelligence"
- 1952–69 Look, Ma, no hands!
- 1950s Early AI programs, including Samuel's checkers program,
Newell & Simon's Logic Theorist, Gelernter's Geometry Engine
- 1956 Dartmouth meeting: "Artificial Intelligence" adopted
- 1965 Robinson's complete algorithm for logical reasoning
- 1966–74 AI discovers computational complexity
Neural network research almost disappears
- 1969–79 Early development of knowledge-based systems
- 1980–88 Expert systems industry booms
- 1988–93 Expert systems industry busts: "AI Winter"
- 1985–95 Neural networks return to popularity
- 1988– Resurgence of probabilistic and decision-theoretic methods
Rapid increase in technical depth of mainstream AI
"Nouvelle AI": ALife, GAs, soft computing

Today: you and me ☺



Forecasting the Future or Shaping It?

October 19, 2000

Our task is not to *predict* the future; our task is to *design* a future for a sustainable and acceptable world, and then to devote our efforts to bringing that future about.

Professor Herbert A. Simon

Nobel Prize Laureate
A Founding Father of Artificial Intelligence



AI State of the art

- Have the following been achieved by AI?
 - Pass the (basic) Turing test
 - World-class players for Chess and Go games
 - Playing table tennis
 - Autonomous cross-country driving
 - Solving mathematical problems
 - Discovering and proving mathematical theories
 - Engaging in a meaningful conversation
 - Understanding spoken language
 - Observing and understanding human emotions
 - Expressing emotions
 - ...

NEWS

[Home](#) | [Video](#) | [World](#) | [UK](#) | [Business](#) | [Tech](#) | [Science](#) | [Magazine](#) | [Entertainment & Arts](#)[Technology](#)

Computer AI passes Turing test in 'world first'

🕒 9 June 2014 | [Technology](#)

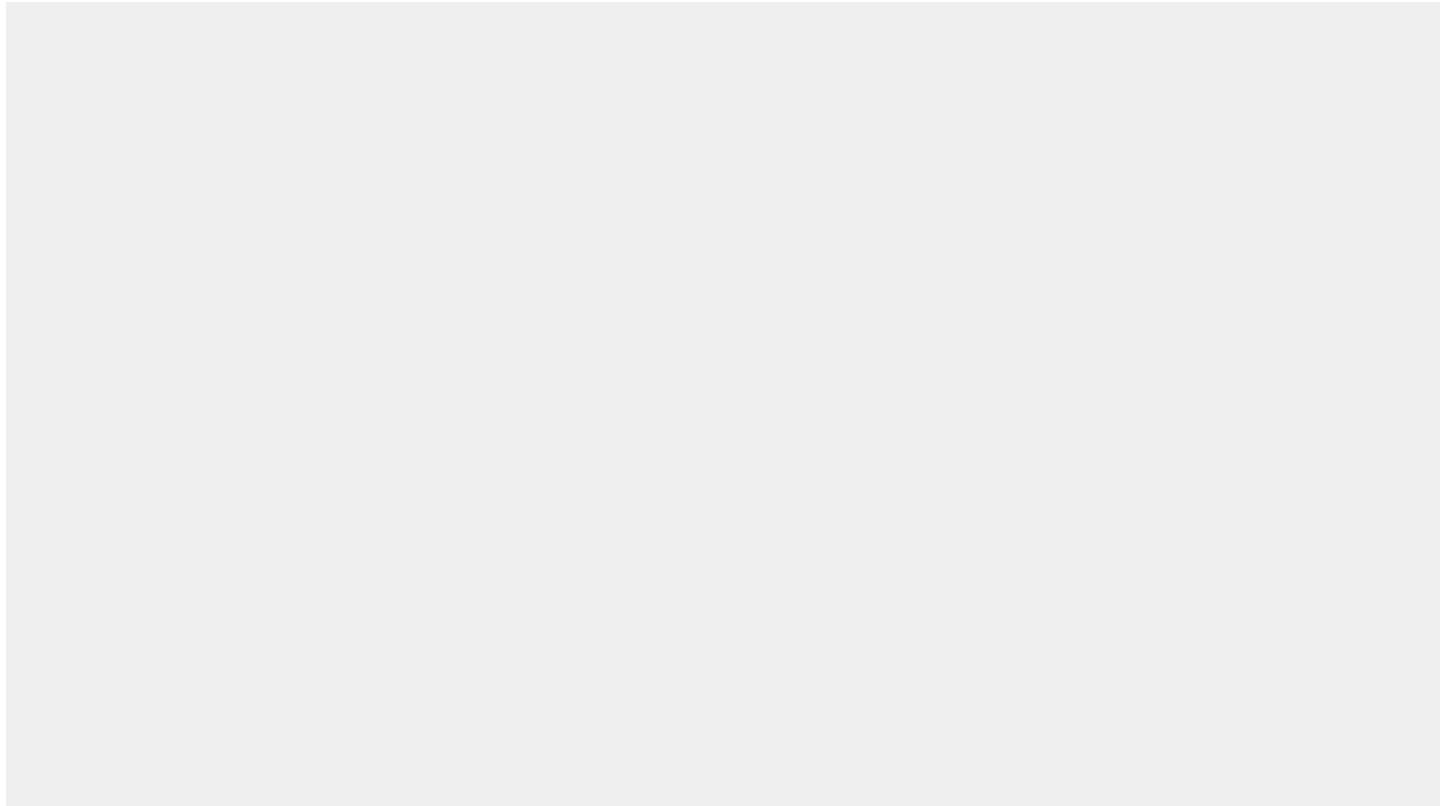
The screenshot shows a web-based interface for the Eugene Goostman AI. At the top, there's a blue header with the text "Eugene Goostman" and "THE WEIRDEST CREATURE IN THE WORLD". Below the header is a large image of a young man with glasses and short brown hair, wearing a dark blue vest over a light-colored shirt. To the right of the image is a white text input field with the placeholder "Type your question here:" and a blue "reply" button below it. At the bottom of the interface, a black footer bar contains the text "Eugene Goostman simulates a 13-year-old Ukrainian boy".



Google DeepMind
Challenge Match
8 - 15 March 2016



AI State of the Art (a movie of robots play table tennis)



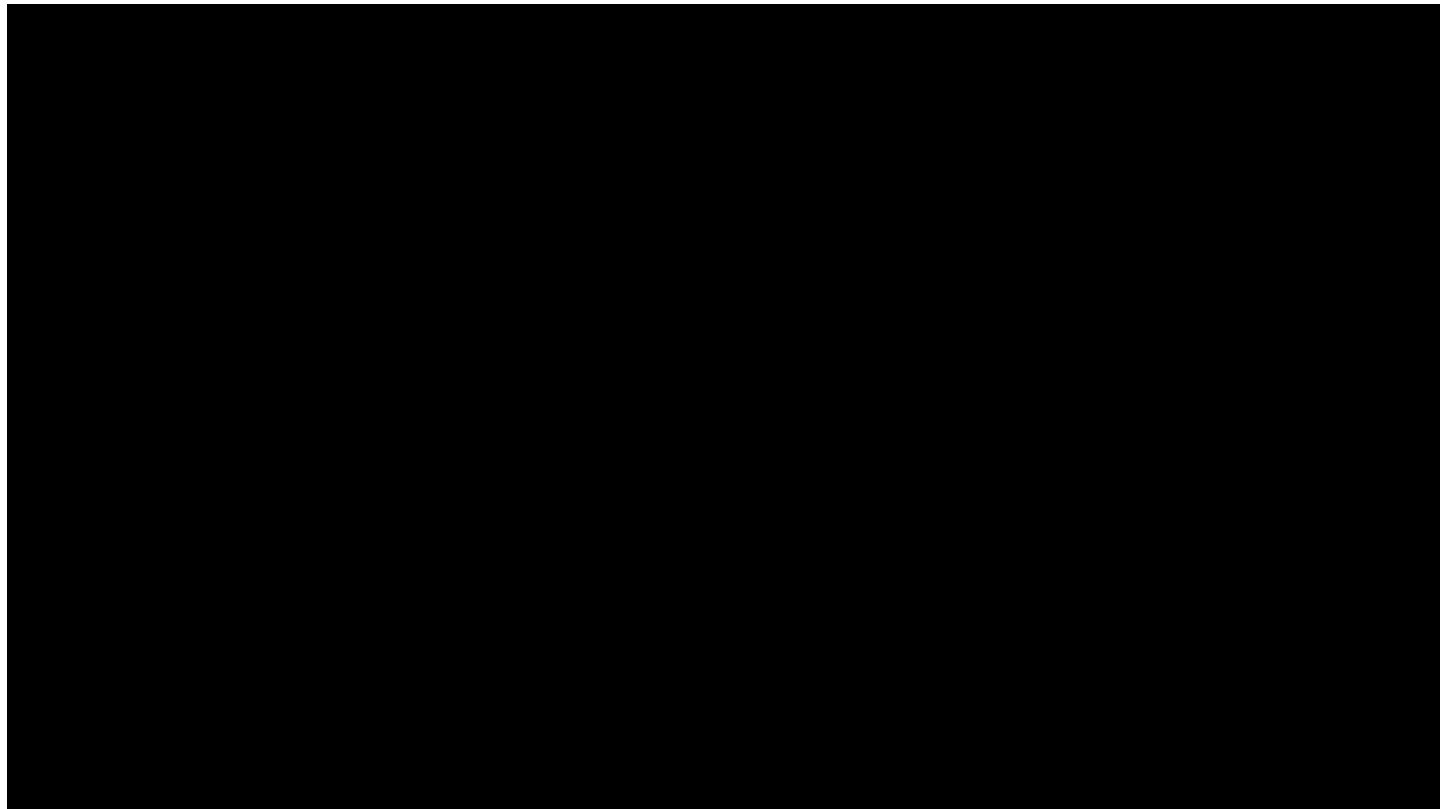
AI State of the Art (Self-Driving Cars)



BOSTON DYNAMICS



AI State of the Art (A female humanoid robot that sings)

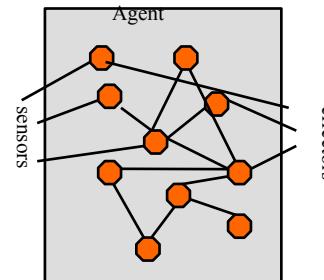


CSCI-561 Course Overview

General Introduction

- **01-Introduction.** [AIMA Ch 1] Course Schedule. Homeworks, exams and grading. Course material, TAs and office hours. Why study AI? What is AI? The Turing test. Rationality. Branches of AI. Research disciplines connected to and at the foundation of AI. Brief history of AI. Challenges for the future. Overview of class syllabus.

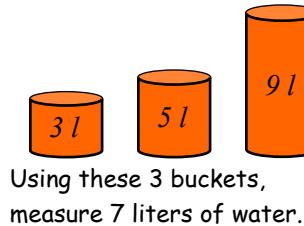
Intelligent Agents. [AIMA Ch 2] What is an intelligent agent? Examples. Doing the right thing (rational action). Performance measure. Autonomy. Environment and agent design. Structure of agents. Agent types. Reflex agents. Reactive agents. Reflex agents with state. Goal-based agents. Utility-based agents. Mobile agents. Information agents.



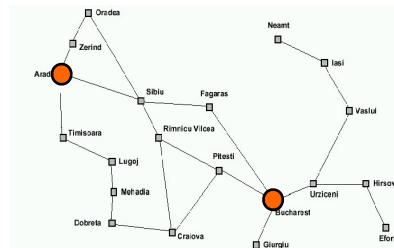
Course Overview (cont.)

How can we solve complex problems?

- **02-Problem solving and search.** [AIMA Ch 3] Example: measuring problem. Types of problems. More example problems. Basic idea behind search algorithms. Complexity. Combinatorial explosion and NP completeness. Polynomial hierarchy.
- **03-Uninformed search.** [AIMA Ch 3] Depth-first. Breadth-first. Uniform-cost. Depth-limited. Iterative deepening. Examples. Properties.
- **04-Informed search.** [AIMA Ch 4] Best-first. A* search. Heuristics. Hill climbing. Problem of local extrema. Simulated annealing. Genetic algorithms.



Using these 3 buckets,
measure 7 liters of water.

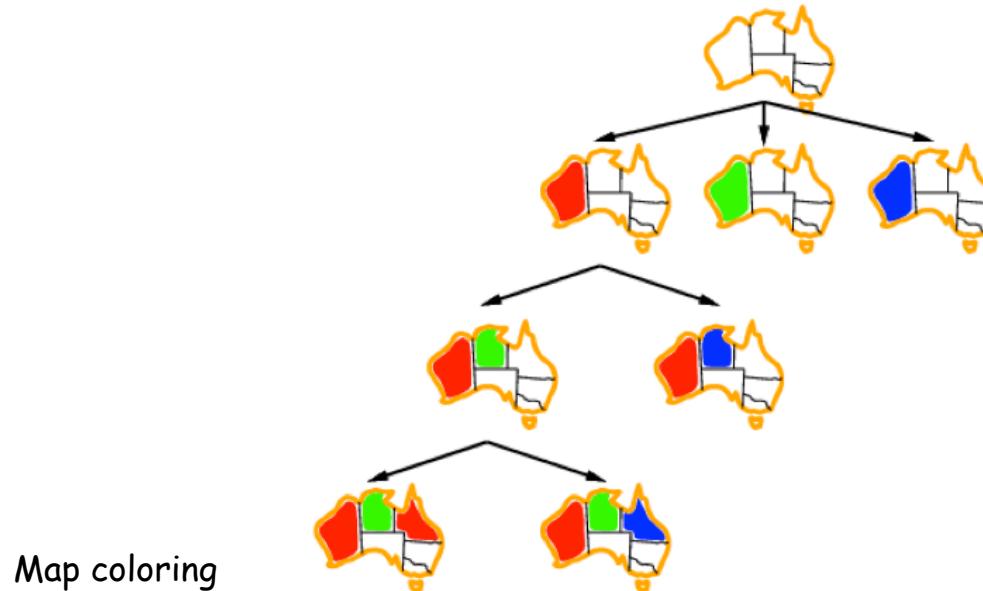


Traveling salesperson problem

Course Overview (cont.)

Search under constraints

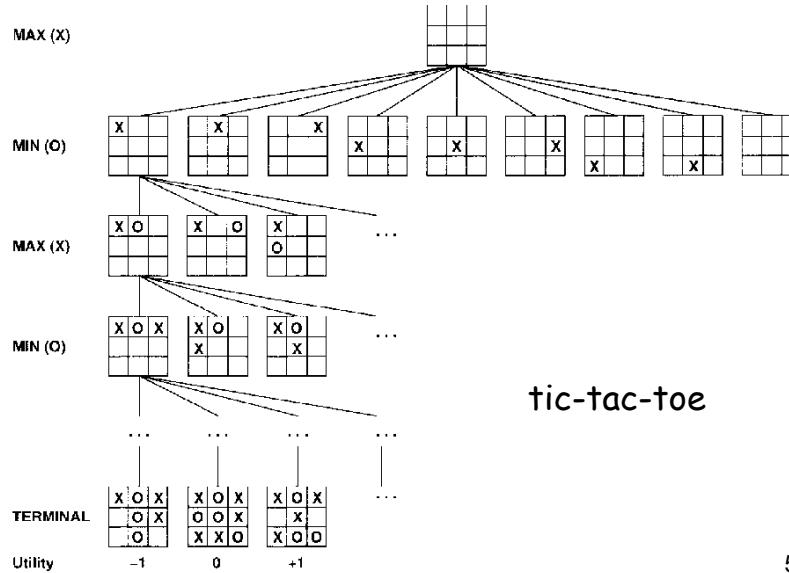
- **05-Constraint satisfaction.** [AIMA Ch 6] Node, arc, path, and k-consistency. Backtracking search. Local search using min-conflicts.



Course Overview (cont.)

Practical applications of search.

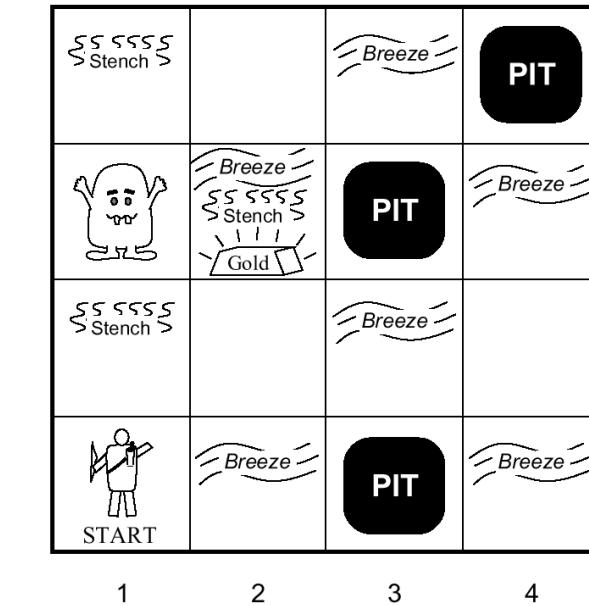
- **06-Game playing.** [AIMA Ch 5] The minimax algorithm. Resource limitations. Alpha-beta pruning. Elements of chance and non-deterministic games.
- **06-Advanced Game Playing** [AIMA Ch 16]
 - Reinforcement Q-Learning



Course Overview (cont.)

Towards intelligent agents

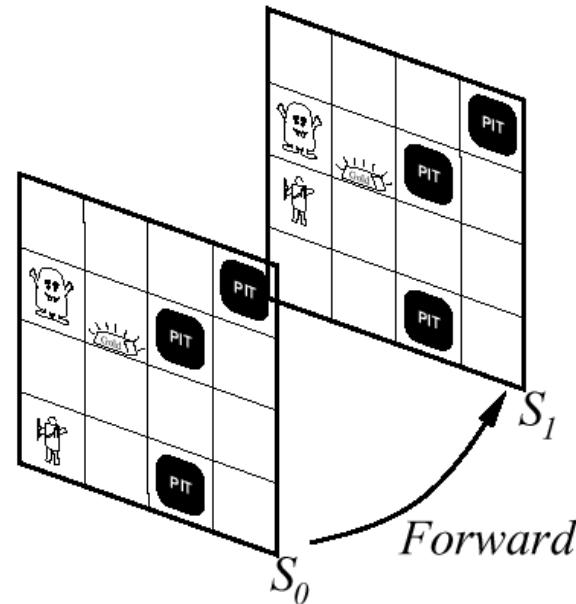
- **9-Agents that reason logically 1.**
[AIMA Ch 7] Knowledge-based agents.
Logic and representation. Propositional
(boolean) logic.
- **10-Agents that reason logically 2.**
[AIMA Ch 7] Inference in propositional
logic. Syntax. Semantics. Examples.



Course Overview (cont.)

Building knowledge-based agents: 1st Order Logic

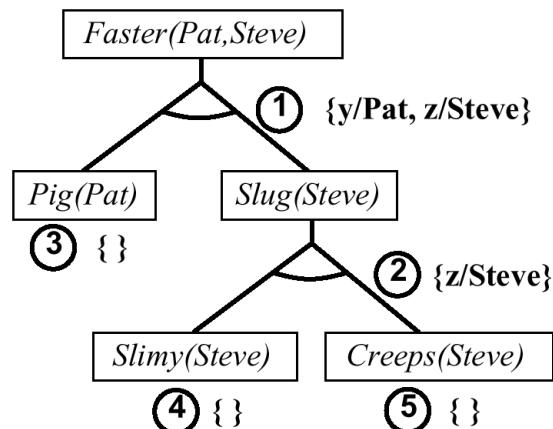
- **11-First-order logic 1.** [AIMA Ch 8] Syntax. Semantics. Atomic sentences. Complex sentences. Quantifiers. Examples. FOL knowledge base. Situation calculus.
- **12-First-order logic 2.**
[AIMA Ch 8] Describing actions.
Planning. Action sequences.



Course Overview (cont.)

Reasoning Logically

- **13/14-Inference in first-order logic.** [AIMA Ch 9] Proofs. Unification. Generalized modus ponens. Forward and backward chaining.



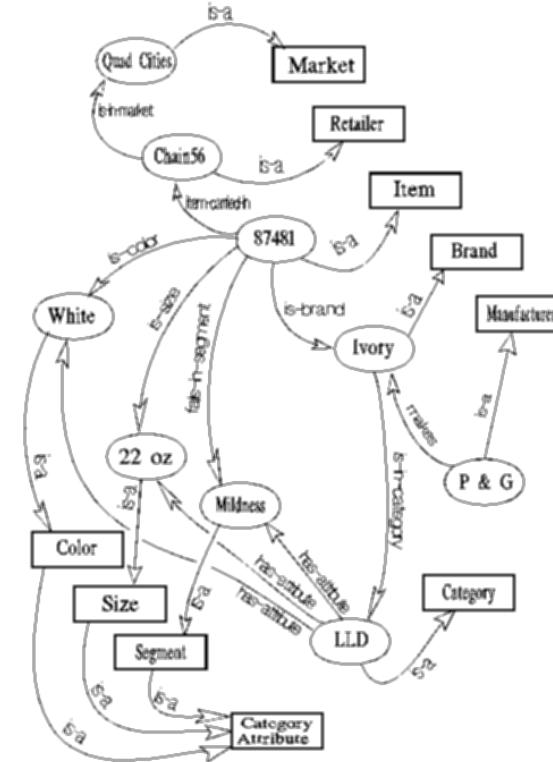
Example of
backward chaining

Course Overview (cont.)

Examples of Logical Reasoning Systems

- **15-Logical reasoning systems.**
[AIMA Ch 9] Indexing, retrieval and unification. The Prolog language. Theorem provers. Frame systems and semantic networks.

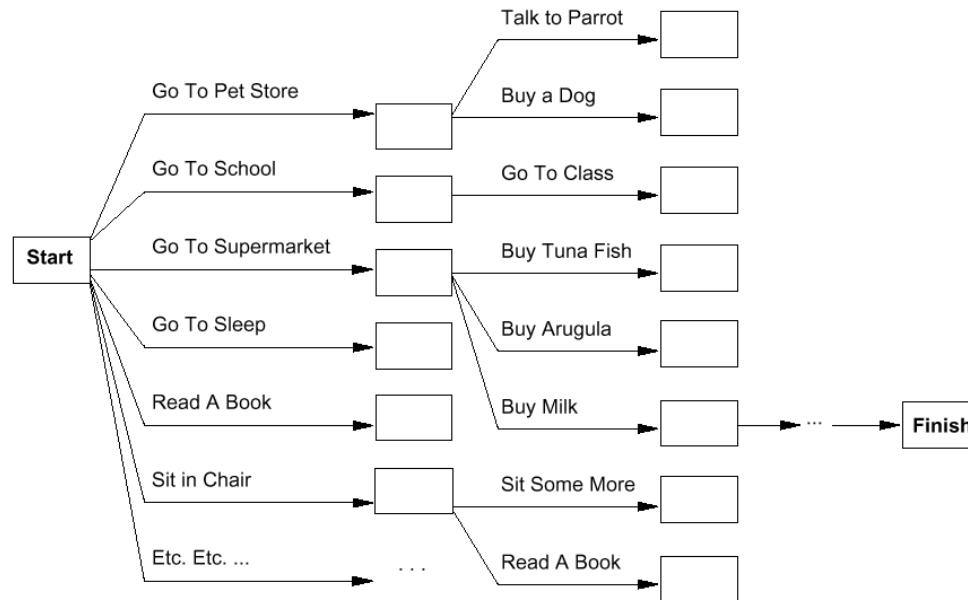
Semantic network used in an insight generator (Duke university)



Course Overview (cont.)

Systems that can Plan Future Behavior

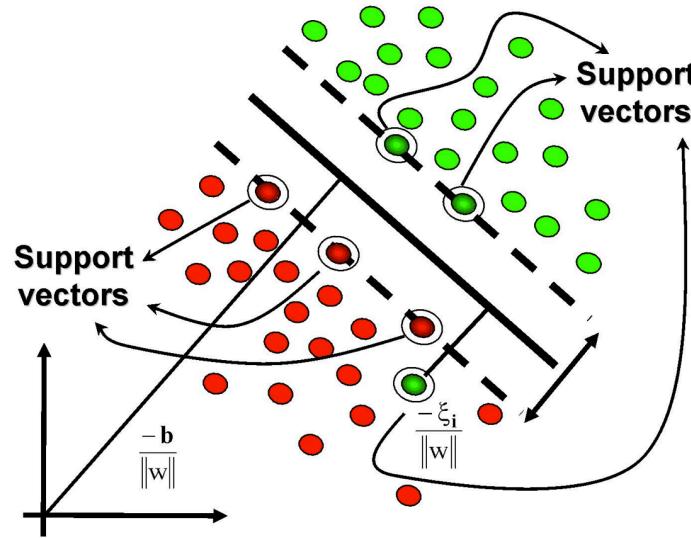
- **16-Planning.** [AIMA Ch 10] Definition and goals. Basic representations for planning. Situation space and plan space. Examples.



Course Overview (cont.)

Handling fuzziness, change, uncertainty.

18-Learning from examples. [AIMA 18 + handout]. Supervised learning, learning decision trees, support vector machines.

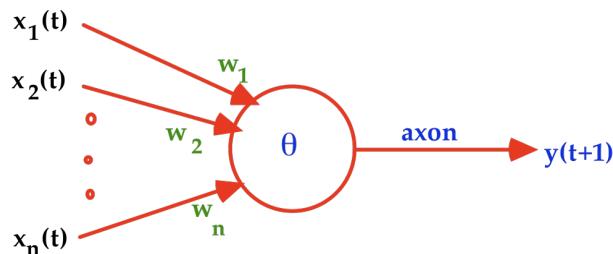
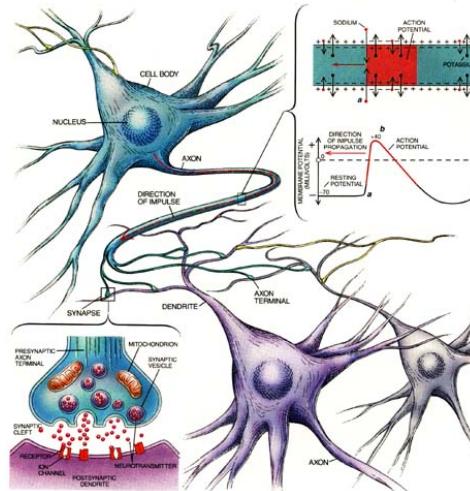


Course Overview (cont.)

Learning with Neural networks

- **19/20-Learning with Neural Networks.**

[Handout + AIMA 18] Introduction to perceptrons, Hopfield networks, self-organizing feature maps. How to size a network? What can neural networks achieve? Advanced concepts – convnets, deep learning, stochastic gradient descent, dropout learning, autoencoders, applications and state of the art.



Course Overview (cont.)

Handling uncertainties, fuzziness, and changes

- **21/22-Probabilistic Reasoning.** [AIMA Ch 13, 14, 15]

Reasoning under uncertainty – probabilities, conditional independence, Markov blanket, Bayes nets. Probabilistic reasoning in time. Hidden Markov Models, Kalman filters, dynamic Bayesian networks.

For assessing diagnostic probability from causal probability:

$$P(Cause|Effect) = \frac{P(Effect|Cause)P(Cause)}{P(Effect)}$$

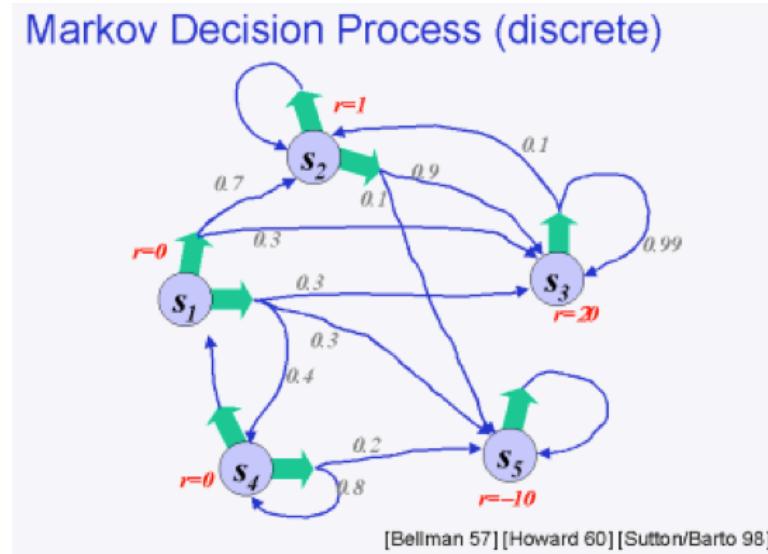
E.g., let M be meningitis, S be stiff neck:

$$P(M|S) = \frac{P(S|M)P(M)}{P(S)} = \frac{0.8 \times 0.0001}{0.1} = 0.0008$$

Course Overview (cont.)

Handling uncertainties, fuzziness, and changes

- **23-Probabilistic decision making.** [AIMA 16, 17] – utility theory, decision networks, value iteration, policy iteration, Markov decision processes (MDP), partially-observable MDP (POMDP).

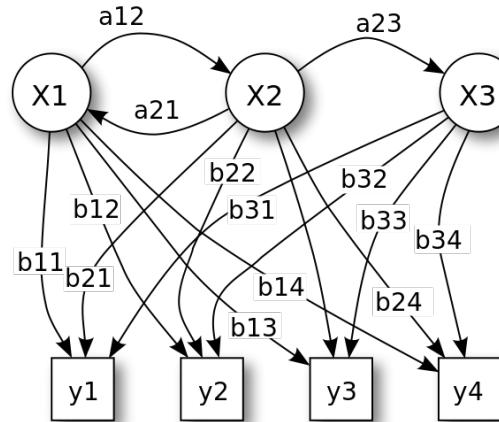


Course Overview (cont.)

Handling uncertainties, fuzziness, and changes

- **24-Probabilistic reasoning over time.** [AIMA 15]

Temporal models, Hidden Markov Models, Kalman filters, Dynamic Bayesian Networks, Automata theory.

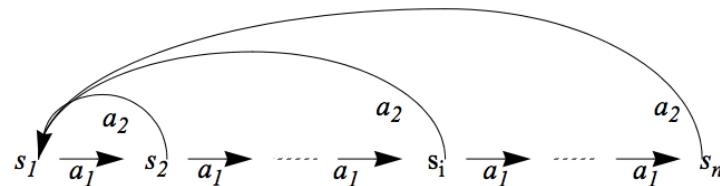


Course Overview (cont.)

Handling uncertainties, fuzziness, and changes

- **25-Probability-based learning.** [AIMA 20-21]

Probabilistic Models, Naïve Bayes Models, EM algorithm, Reinforcement Learning.



$$Q_{k+1}(s, a) \leftarrow \sum_{s'} T(s, a, s') \left[R(s, a, s') + \gamma \max_{a'} Q_k(s', a') \right]$$

Course Overview (cont.)

What challenges remain?

- **26-Towards intelligent machines.** [AIMA Ch 26, 27] The challenge of robots: with what we have learned, what hard problems remain to be solved? Different types of robots. Tasks that robots are for. Parts of robots. Architectures. Configuration spaces. Navigation and motion planning. Towards highly-capable robots. What have we learned. Where do we go from here?



More from the Discussion Session later today

INTELLIGENT AGENTS

Defining Intelligent Agents

- Intelligent Agents (IA)
- Environment types
- IA Behavior
- IA Structure
- IA Types

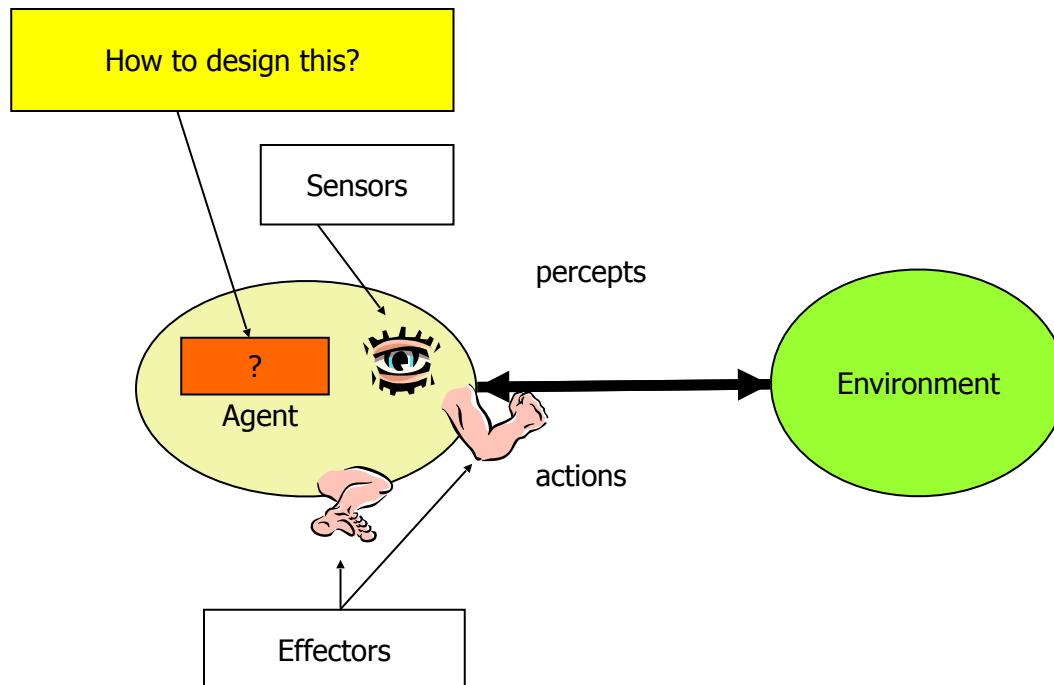
What is an (Intelligent) Agent?

- An over-used, over-loaded, and misused term.
- Anything that can be **viewed** as **perceiving** its **environment** through **sensors** and **acting** upon that environment through its **effectors** to maximize progress towards its **goals**.

What is an (Intelligent) Agent?

- **PAGE** (Percepts, Actions, Goals, Environment)
- Task-specific & specialized: well-defined goals and environment
- The notion of an agent is meant to be a tool for analyzing systems, It is not a different hardware or new programming languages

Agent and Environment Interactions



A Windshield Wiper Agent (Example)

How do we design a agent that can wipe the windshields when needed?

- Goals?
- Percepts?
- Sensors?
- Effectors?
- Actions?
- Environment?

A Windshield Wiper Agent (Cont'd)

- Goals: Keep windshields clean & maintain visibility
- Percepts: Raining, Dirty
- Sensors: Camera (moist sensor)
- Effectors: Wipers (left, right, back)
- Actions: Off, Slow, Medium, Fast
- Environment: Inner city, freeways, highways, weather ...

Interactions Among Agents

Collision Avoidance Agent (CAA)

- Goals: Avoid running into obstacles
- Percepts ?
- Sensors?
- Effectors ?
- Actions ?
- Environment: Freeway

Lane Keeping Agent (LKA)

- Goals: Stay in current lane
- Percepts ?
- Sensors?
- Effectors ?
- Actions ?
- Environment: Freeway

Interactions Among Agents

Collision Avoidance Agent (CAA)

- Goals: Avoid running into obstacles
- Percepts: Obstacle distance, velocity, trajectory
- Sensors: Vision, proximity sensing
- Effectors: Steering Wheel, Accelerator, Brakes, Horn, Headlights
- Actions: Steer, speed up, brake, blow horn, signal (headlights)
- Environment: Freeway

Lane Keeping Agent (LKA)

- Goals: Stay in current lane
- Percepts: Lane center, lane boundaries
- Sensors: Vision
- Effectors: Steering Wheel, Accelerator, Brakes
- Actions: Steer, speed up, brake
- Environment: Freeway

Conflict Resolution by Action Selection Agents

- **Override:** CAA overrides LKA
- **Arbitrate:** if Obstacle is Close then CAA else LKA
- **Compromise:** Choose action that satisfies both agents
- Any combination of the above
- **Challenges:** Doing the right thing at the right time

The Right Thing = The Rational Action

- **Rational Action:** The action that maximizes the expected value of the performance measure given the percept sequence to date
 - Rational = Best ?
 - Rational = Optimal ?
 - Rational = Omniscience ?
 - Rational = Clairvoyant (supernatural) ?
 - Rational = Successful ?

The Right Thing = The Rational Action

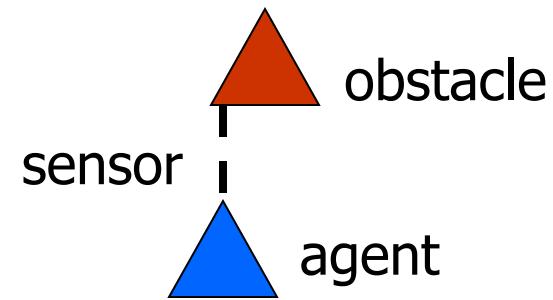
- **Rational Action:** The action that maximizes the expected value of the performance measure given the percept sequence to date
 - Rational = Best Yes, to the best of its knowledge
 - Rational = Optimal Yes, to the best of its abilities (incl. its constraints)
 - Rational ≠ Omniscience
 - Rational ≠ Clairvoyant (supernatural)
 - Rational ≠ Successful

Behavior and Performance of IAs

- **Perception (sequence) to Action Mapping:** $f : P^* \rightarrow A$
 - **Ideal mapping:** specifies which actions an agent ought to take at any point in time
 - **Implementation:** Look-Up-Table, Closed Form, Algorithm, etc.
- **Performance measure:** a subjective measure to characterize how successful an agent is (e.g., speed, power usage, accuracy, money, etc.)
- (degree of) **Autonomy:** to what extent is the agent able to make decisions and take actions on its own?

"Look-Up-Table" Agent

Distance	Action
10	No action
5	Turn left 30 degrees
2	Stop



Closed Form

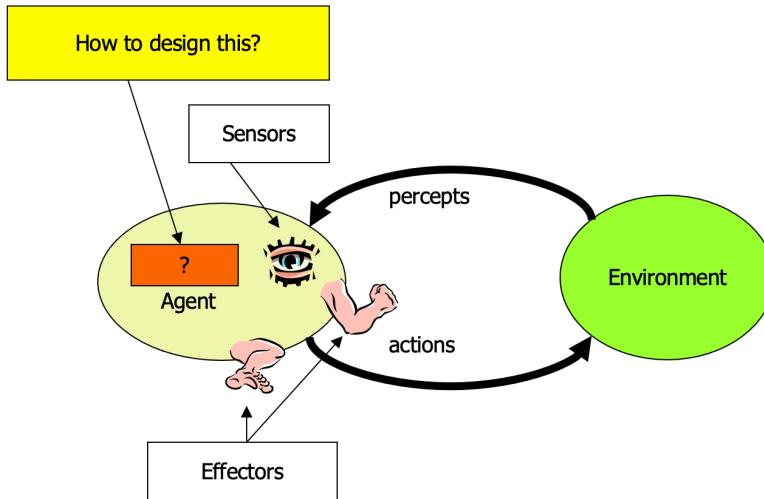
- Output (degree of rotation) = F (distance)
- E.g., $F(d) = 10/d$ (distance cannot be less than 1/10)

How is an Agent different from other software?

Is A/C Thermostats an Agent?
Is Zoom an agent?

- Agents are **autonomous**, that is, they act on behalf of the user
- Agents contain some level of **intelligence**, from fixed rules to learning engines that allow them to adapt to changes in the environment
- Agents don't only act **reactively**, but sometimes also **proactively**
- Agents have **social ability**, that is, they communicate with the user, the system, and other agents as required
- Agents may also **cooperate** with other agents to carry out more complex tasks than they themselves can handle
- Agents may **migrate** from one system to another to access remote resources or even to meet other agents

Types of Environments



- Characteristics

- Accessible vs. Inaccessible
- Deterministic vs. Nondeterministic
- Episodic vs. Non-episodic
- Hostile vs. Friendly
- Static vs. Dynamic
- Discrete vs. Continuous

Environment Types

- Characteristics
 - Accessible (observable) vs. inaccessible (partial observable)
 - Accessible: sensors give access to **complete** state of the environment.
 - Deterministic vs. nondeterministic
 - Deterministic: the next state can be determined based on the current state and the action.
 - Episodic (history-insensitive) vs. non-episodic (Sequential, history-sensitive)
 - Episode: each perceive and action pairs
 - In episodic environments, the quality of action does not depend on the previous episode and does not affect the next episode.
 - Example: Episodic: mail sorting system; non-episodic: chess (why? "castling"), Go?

Environment Types

- Characteristics
 - Hostile vs. friendly
 - Static vs. dynamic
 - Dynamic if the environment changes during deliberation
 - Discrete vs. continuous
 - Chess vs. driving

Environment types

Environment	Accessible	Deterministic	Episodic	Static	Discrete
Operating System					
Virtual Reality					
Office Environment					
Mars					

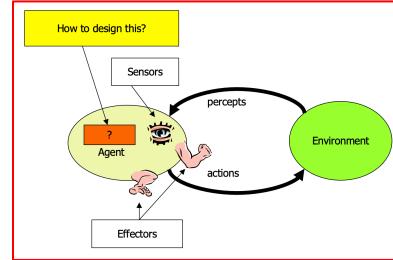
Environment types

Environment	Accessible	Deterministic	Episodic	Static	Discrete
Operating System	Yes	Yes	No	No	Yes
Virtual Reality	Yes	Yes	Yes/no	No	Yes/no
Office Environment	No	No	No	No	No
Mars	No	Semi	No	Semi	No

The environment types largely determine the agent design.

Structure of Intelligent Agents

- **Agent** = architecture + program



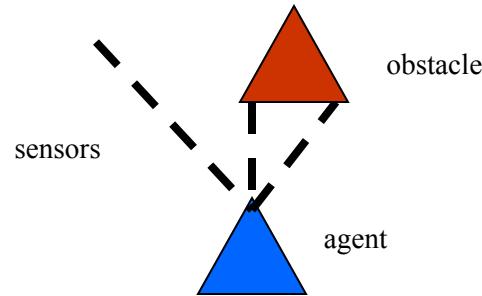
- **Agent Program:** the implementation of $f : P^* \rightarrow A$, the agent's perception-action mapping

```
function Skeleton-Agent(Percept) returns Action
    memory ← UpdateMemory(memory, Percept)
    Action ← ChooseBestAction(memory)
    memory ← UpdateMemory(memory, Action)
return Action
```

- **Architecture:** a device that can execute the agent program (e.g., general-purpose computer, specialized device, beobot, etc.)

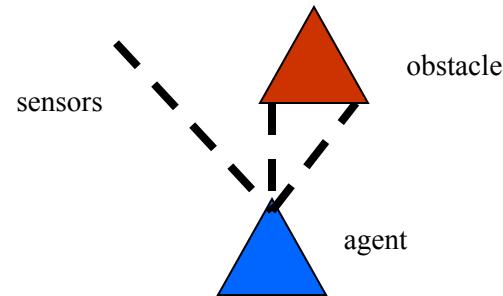
Using a look-up-table to encode $f : \mathcal{P}^* \rightarrow A$

- **Example:** Collision Avoidance
 - Sensors: 3 proximity sensors
 - Effectors: Steering Wheel, Brakes
- How to generate?
- How large?
- How to select action?



Using a look-up-table to encode $f : \mathcal{P}^* \rightarrow A$

- **Example:** Collision Avoidance
 - Sensors: 3 proximity sensors
 - Effectors: Steering Wheel, Brakes
- **How to generate:** for each $p \in \mathcal{P}_l \times \mathcal{P}_m \times \mathcal{P}_r$ generate an appropriate action, $a \in \mathcal{S} \times \mathcal{B}$
- **How large:** size of table = # possible percepts times # possible actions = $|\mathcal{P}_l| |\mathcal{P}_m| |\mathcal{P}_r| |\mathcal{S}| |\mathcal{B}|$
E.g., $\mathcal{P} = \{\text{close, medium, far}\}^3$
 $\mathcal{A} = \{\text{left, straight, right}\} \times \{\text{on, off}\}$
then size of table = 27 rows
- Total possible combinations (ways to fill table): $27 * 3 * 2 = 162$
- **How to select action?** Search.



TYPES OF AGENT

Agent Types

- Reflex agents
- Reflex agents with internal states
- Goal-based agents
- Utility-based agents
- Learning agents

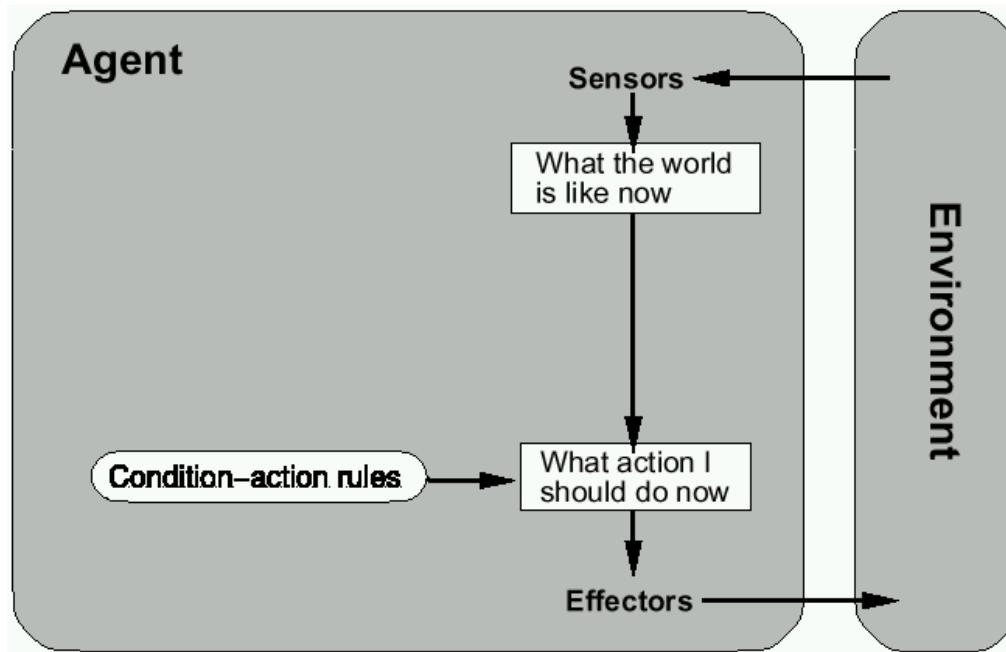
Agent Types

- Reflex agents
 - Reactive: No memory
- Reflex agents with internal states
 - W/o previous state, may not be able to make decision
 - E.g. brake lights at night.
- Goal-based agents
 - Goal information needed to make decision

Agent Types

- Utility-based agents
 - How well can the goal be achieved (degree of happiness)
 - What to do if there are conflicting goals?
 - Speed and safety
 - Which goal should be selected if several can be achieved?
- Learning agents
 - How can I adapt to the environment?
 - How can I learn from my mistakes?

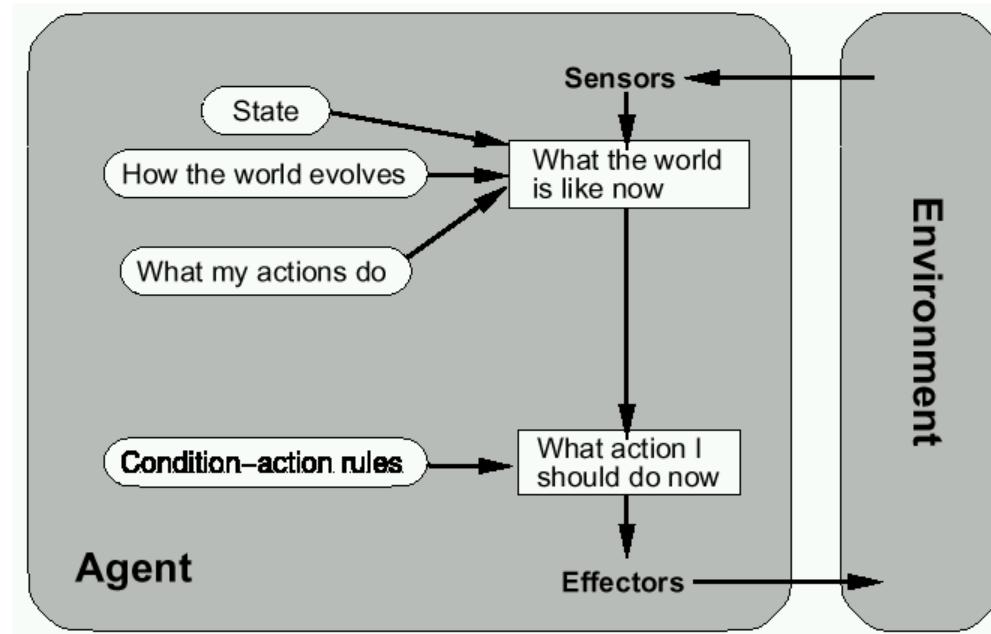
Reflex Agents



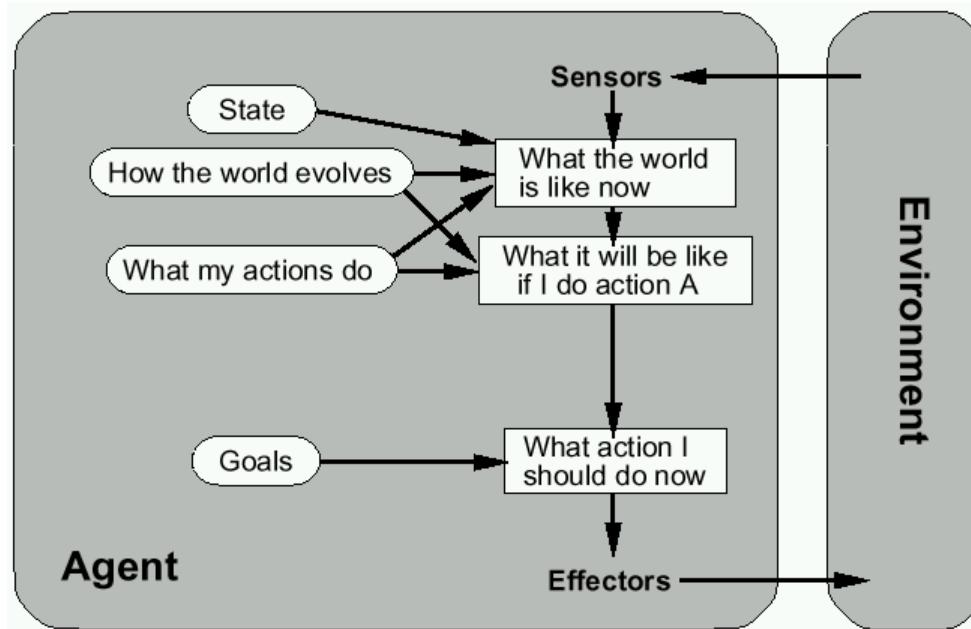
Reactive Agents

- Reactive agents do not have internal symbolic models.
 - Act by stimulus-response to the current state of the environment.
 - Each reactive agent is simple and interacts with others in a basic way.
 - Complex patterns of behavior emerge from their interaction.
-
- **Benefits:** robustness, fast response time
 - **Challenges:** scalability, how intelligent? how do you debug them?

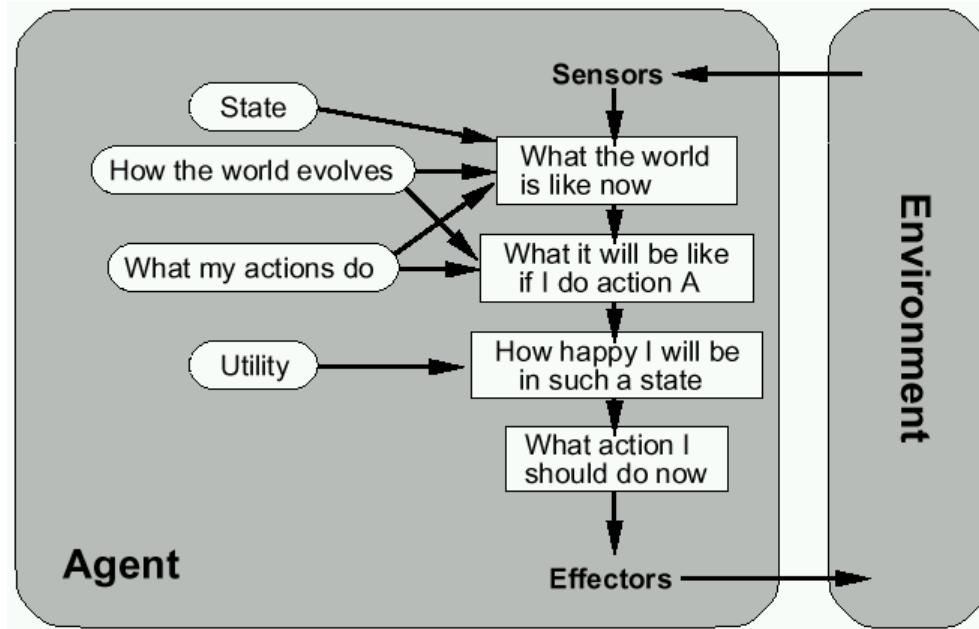
Reflex Agents with Internal States



Goal-Based Agents



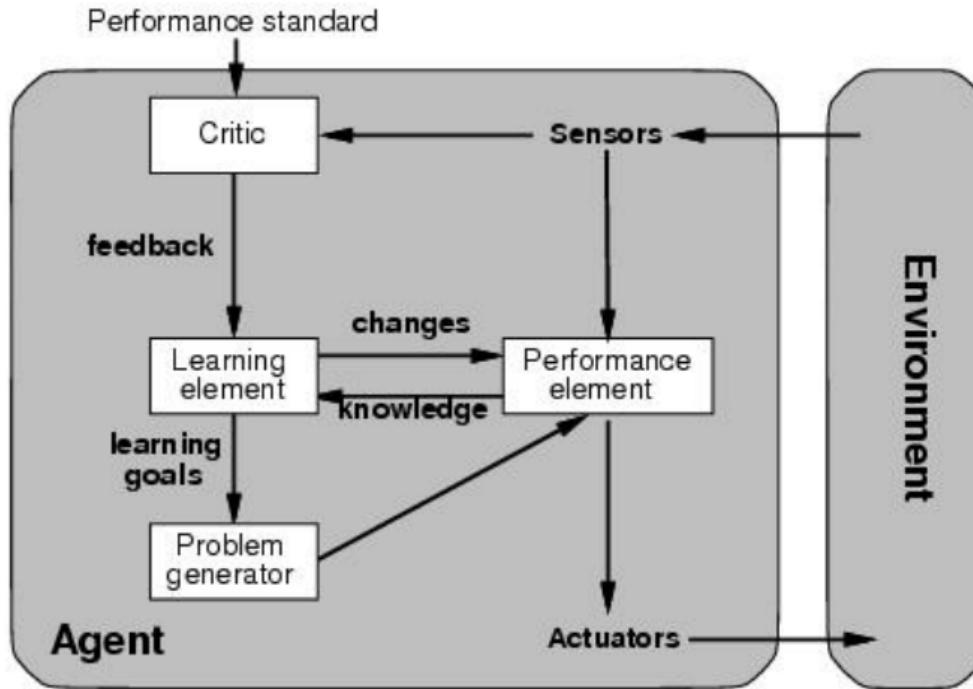
Utility-Based Agents



Learning Agents

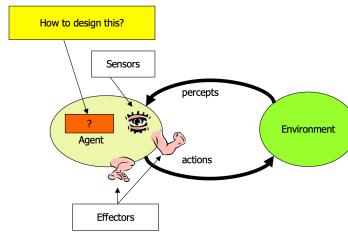
Critic: Determines outcomes of actions and gives feedback

Learning element:
Takes feedback from
critic and improves
performance element



Problem generator: creates new experiences
to promote learning

Summary on Intelligent Agents



- **Intelligent Agents:**

- Anything that can be **viewed** as **perceiving** its **environment** through **sensors** and **acting** upon that environment through its **effectors** to maximize progress towards its **goals**.
- PAGE (Percepts, Actions, Goals, Environment)
- Described as a Perception (sequence) to Action Mapping: $f : p^* \rightarrow A$
- Using look-up-table, closed form, etc.

- **Agent Types:** Reflex, state-based, goal-based, utility-based, learning

- **Rational Action:** The action that maximizes the expected value of the performance measure given the percept sequence to date

CSCI 561 - Foundation for Artificial Intelligence

02. Problem Solving & Search

Professor Wei-Min Shen
University of Southern California

Outline

Problem Solving and Search

- Search Space

Formulation of “Problem Solving” and Search

Complexity of Problems

- Search Algorithms (uninformed vs informed)

- Uninformed Search

Search strategies: breadth-first, uniform-cost, depth-first, bi-directional, ...

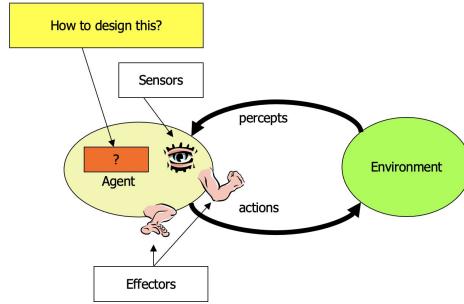
- Informed Search (next lecture)

Search strategies: best-first, A*

Heuristic functions

Review of General AI

- **Definition of AI?**
- **Turing Test?**
- **Intelligent Agents:**



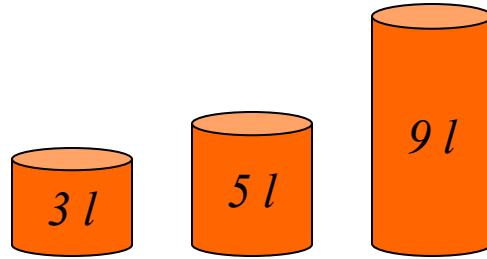
- Anything that can be **viewed** as **perceiving** its **environment** through **sensors** and **acting** upon that environment through its **effectors** to maximize progress towards its **goals**.
- PAGE (Percepts, Actions, Goals, Environment)
- Described as a Perception (sequence) to Action Mapping: $f : P^* \rightarrow A$
- Using look-up-table, closed form, etc.
- **Agent Types:** Reflex, state-based, goal-based, utility-based
- **Rational Action:** The action that maximizes the expected value of the performance measure given the percept sequence to date

PROBLEM SOLVING

WHAT IS A “PROBLEM”?

HOW TO DEFINE A “PROBLEM”?

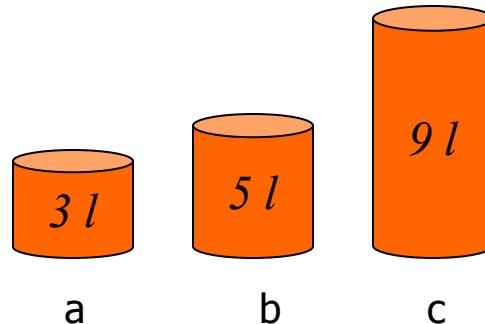
An Example of “Problem” (Measuring Water)



Problem: Using these three buckets,
measure 7 liters of water.

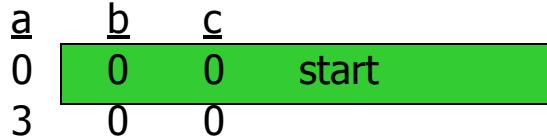
Example: Measuring problem!

- **(one possible) Solution:**

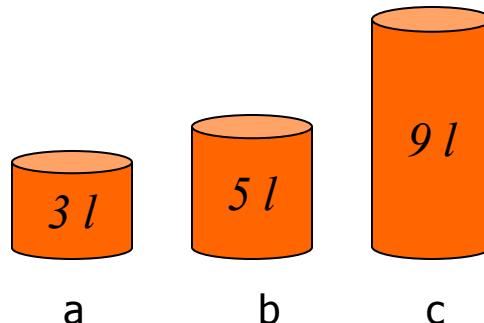


Example: Measuring problem!

- **(one possible) Solution:**



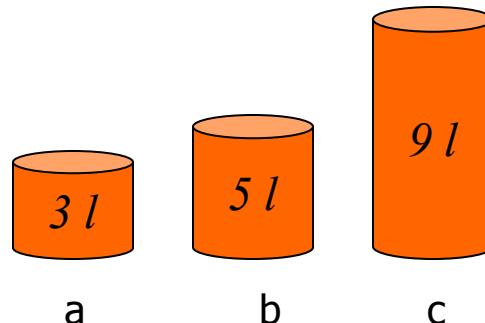
C
C
C
C
C
C
1
C



Example: Measuring problem!

- (one possible) Solution:

a	b	c	
0	0	0	start
3	0	0	
0	0	3	
1			
2			
3			



Example: Measuring problem!

- (one possible) Solution:

a	b	c	
0	0	0	start
3	0	0	

0 0 3

3 0 3

- -

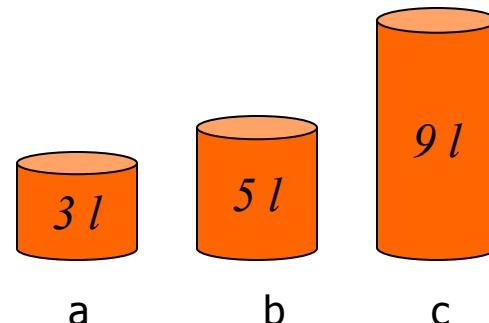
C C

C C

C C

1

C



Example: Measuring problem!

- (one possible) Solution:

a	b	c	
0	0	0	start
3	0	0	

0 0 3

3 0 3

0 0 6

3 0 3

0 0 6

3 0 3

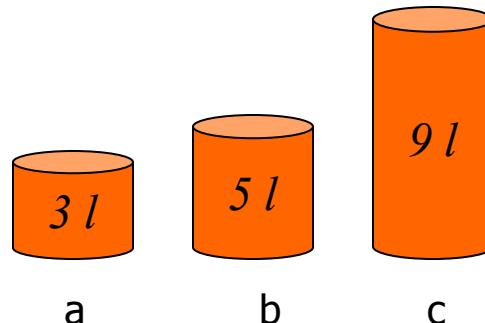
0 0 6

3 0 3

0 0 6

3 0 3

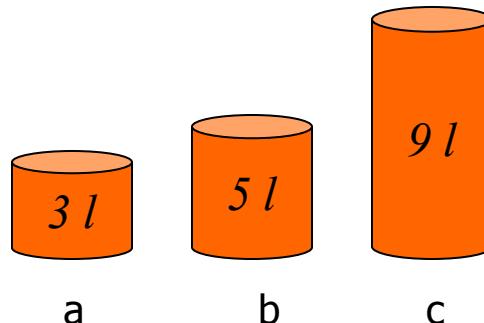
0 0 6



Example: Measuring problem!

- (one possible) Solution:

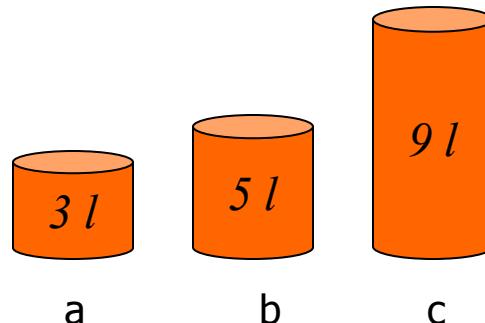
a	b	c	
0	0	0	start
3	0	0	
0	0	3	
3	0	3	
0	0	6	
3	0	6	
c	-	-	
:			
1			
c			



Example: Measuring problem!

- **(one possible) Solution:**

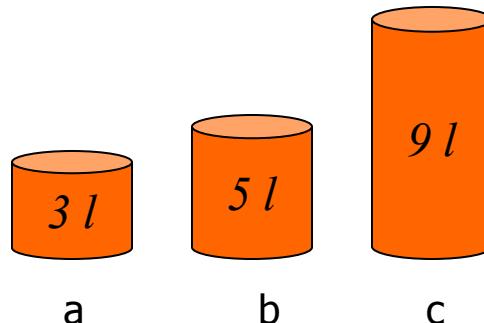
a	b	c	
0	0	0	start
3	0	0	
0	0	3	
3	0	3	
0	0	6	
3	0	6	
0	3	6	
-	-	-	
1			
0			



Example: Measuring problem!

- (one possible) Solution:

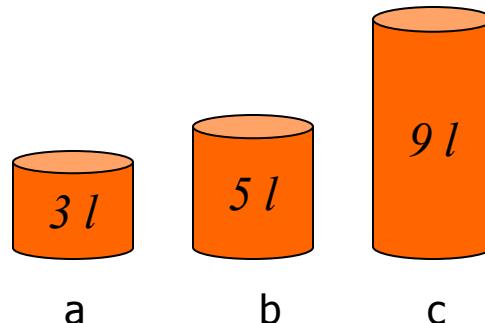
a	b	c	
0	0	0	start
3	0	0	
0	0	3	
3	0	3	
0	0	6	
3	0	6	
0	3	6	
3	3	6	
1			
C			



Example: Measuring problem!

- (one possible) Solution:

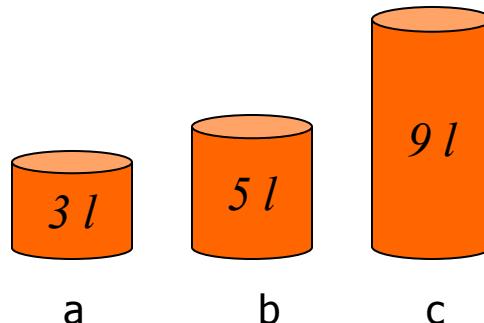
a	b	c	
0	0	0	start
3	0	0	
0	0	3	
3	0	3	
0	0	6	
3	0	6	
0	3	6	
3	3	6	
1	5	6	
0	5	7	---



Example: Measuring problem!

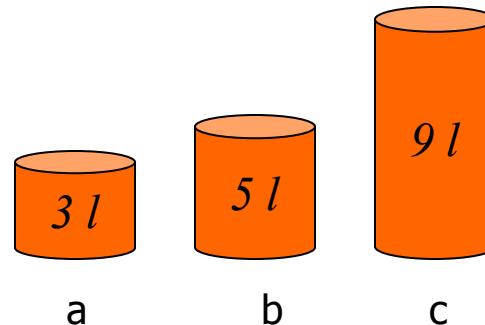
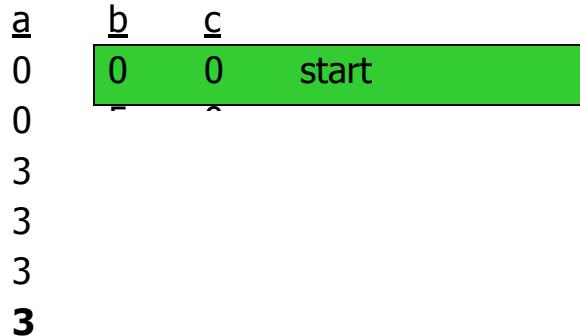
- (one possible) Solution:

a	b	c	
0	0	0	start
3	0	0	
0	0	3	
3	0	3	
0	0	6	
3	0	6	
0	3	6	
3	3	6	
1	5	6	
0	5	7	goal



Example: Measuring problem!

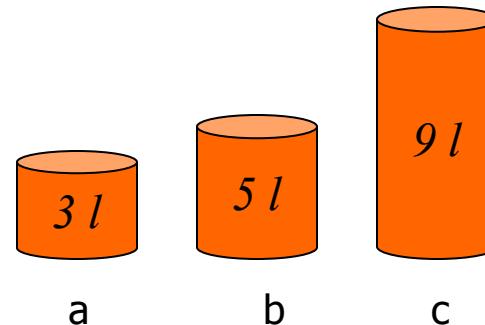
- Another Solution:



Example: Measuring problem!

- Another Solution:

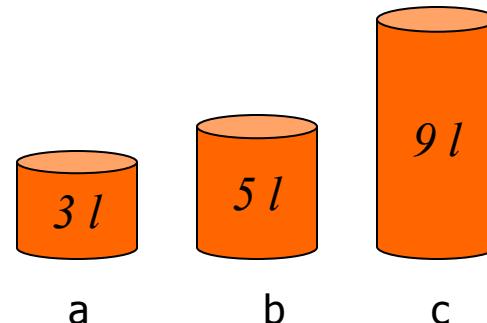
a	b	c	
0	0	0	start
0	5	0	
3			
3			
3			
3			



Example: Measuring problem!

- Another Solution:

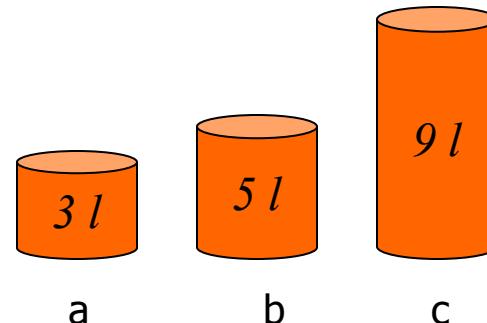
a	b	c	
0	0	0	start
0	5	0	
3	2	0	
3	~	~	
3			
3			



Example: Measuring problem!

- Another Solution:

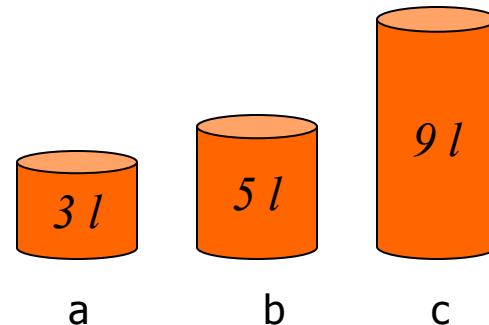
a	b	c	
0	0	0	start
0	5	0	
3	2	0	
3	0	2	
3			
3			



Example: Measuring problem!

- Another Solution:

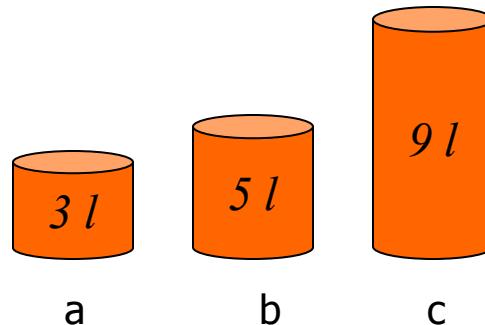
a	b	c	
0	0	0	start
0	5	0	
3	2	0	
3	0	2	
3	5	2	
3			



Example: Measuring problem!

- Another Solution:

a	b	c	
0	0	0	start
0	5	0	
3	2	0	
3	0	2	
3	5	2	
3	0	7	goal



Which solution do we prefer?

- **Solution 1:**

<u>a</u>	<u>b</u>	<u>c</u>	
0	0	0	start
3	0	0	
0	0	3	
3	0	3	
0	0	6	
3	0	6	
0	3	6	
3	3	6	
1	5	6	
0	5	7	goal

- **Solution 2:**

<u>a</u>	<u>b</u>	<u>c</u>	
0	0	0	start
0	5	0	
3	2	0	
3	0	2	
3	5	2	
3	0	7	goal

How to Define a “Problem”?

Measure 7 liters of water using a 3-liter, a 5-liter, and a 9-liter buckets.

- **Formulate Problem:**

1. States: amount of water in the buckets
2. Operators/Actions: Fill bucket from source, empty bucket
3. Initial State: three empty buckets
4. Goal State: Have 7 liters of water in 9-liter bucket

- **Find solution:** a sequence of operators/actions that bring your agent from the initial/current state to the goal state

Problem-Solving Agent

```
function SIMPLE-PROBLEM-SOLVING-AGENT(p) returns an action
    inputs: p, a percept
    static: s, an action sequence, initially empty
            state, some description of the current world state
            g, a goal, initially null
            problem, a problem formulation

    state  $\leftarrow$  UPDATE-STATE(state, p) // What is the current state?
    if s is empty then
        g  $\leftarrow$  FORMULATE-GOAL(state) // From LA to San Diego (given curr. state)
        problem  $\leftarrow$  FORMULATE-PROBLEM(state, g) // e.g., Gas usage
        s  $\leftarrow$  SEARCH(problem)
    action  $\leftarrow$  RECOMMENDATION(s, state)
    s  $\leftarrow$  REMAINDER(s, state)      // If fails to reach goal, update
    return action
```

Note: This is *offline* problem-solving. *Online* problem-solving involves acting w/o complete knowledge of the problem and environment

Remember: Environment Types

Environment	Accessible	Deterministic	Episodic	Static	Discrete
Operating System	Yes	Yes	No	No	Yes
Virtual Reality	Yes	Yes	Yes/No	No	Yes/No
Office Environment	No	No	No	No	No
Mars	No	Semi	No	Semi	No

The environment types largely determine the design of agent

Types of “Problems”

- **Single-state problem:** deterministic, accessible (totally observable)
Agent knows everything about world, thus can calculate optimal action sequence to reach goal state.
- **Multiple-state problem:** deterministic, inaccessible (partially observable)
Agent must reason about sequences of actions and states assumed while working towards goal state.
- **Contingency problem:** nondeterministic, inaccessible
 - *Must use sensors during execution*
 - *Solution is a tree or policy*
 - *Often interleave search and execution*
- **Exploration problem:** unknown state space
Discover and learn about environment while taking actions.

Problem types

- **Single-state problem:** deterministic, accessible
 - Agent knows everything about world (the exact state),
 - Can calculate optimal action sequence to reach goal state.
- E.g., playing chess. Any action will result in an exact state
- Why is Chess or Go “accessible”? (what about “casting”?)

Problem types

- **Multiple-state problem:** deterministic, inaccessible
 - Agent does not know the exact state (could be in any of the possible states)
 - May not have sensors at all
 - Assume states while working towards goal state.
- E.g., walking in a dark room, or playing the poker game
 - If you are at the door, going straight will lead you to the kitchen
 - If you are at the kitchen, turning left leads you to the bedroom
 - ...

Problem types

- **Contingency problem:** nondeterministic, inaccessible
 - Must use sensors during execution
 - Solution is a tree or policy
 - Often interleave search and execution
- E.g., a new skater in an arena
 - Sliding problem.
 - Many skaters around

Problem types

- **Exploration problem:** unknown state space

Discover and learn about environment while taking actions.

- *E.g., Maze, or Mars*

Example 1: Vacuum world

Simplified world: 2 locations, each may or not contain dirt,
each may or not contain vacuuming agent.

Goal of agent: clean up the dirt.

Single-state, start in #5. Solution??

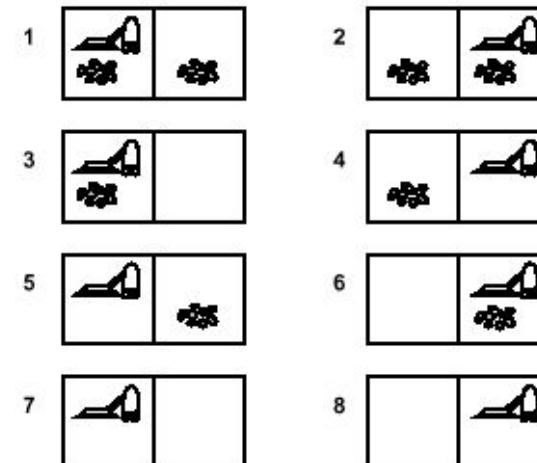
Multiple-state, start in $\{1, 2, 3, 4, 5, 6, 7, 8\}$
e.g., Right goes to $\{2, 4, 6, 8\}$. Solution??

Contingency, start in #5

Murphy's Law: *Suck* can dirty a clean carpet

Local sensing: dirt, location only.

Solution??







CS 561, Sessions 2-3



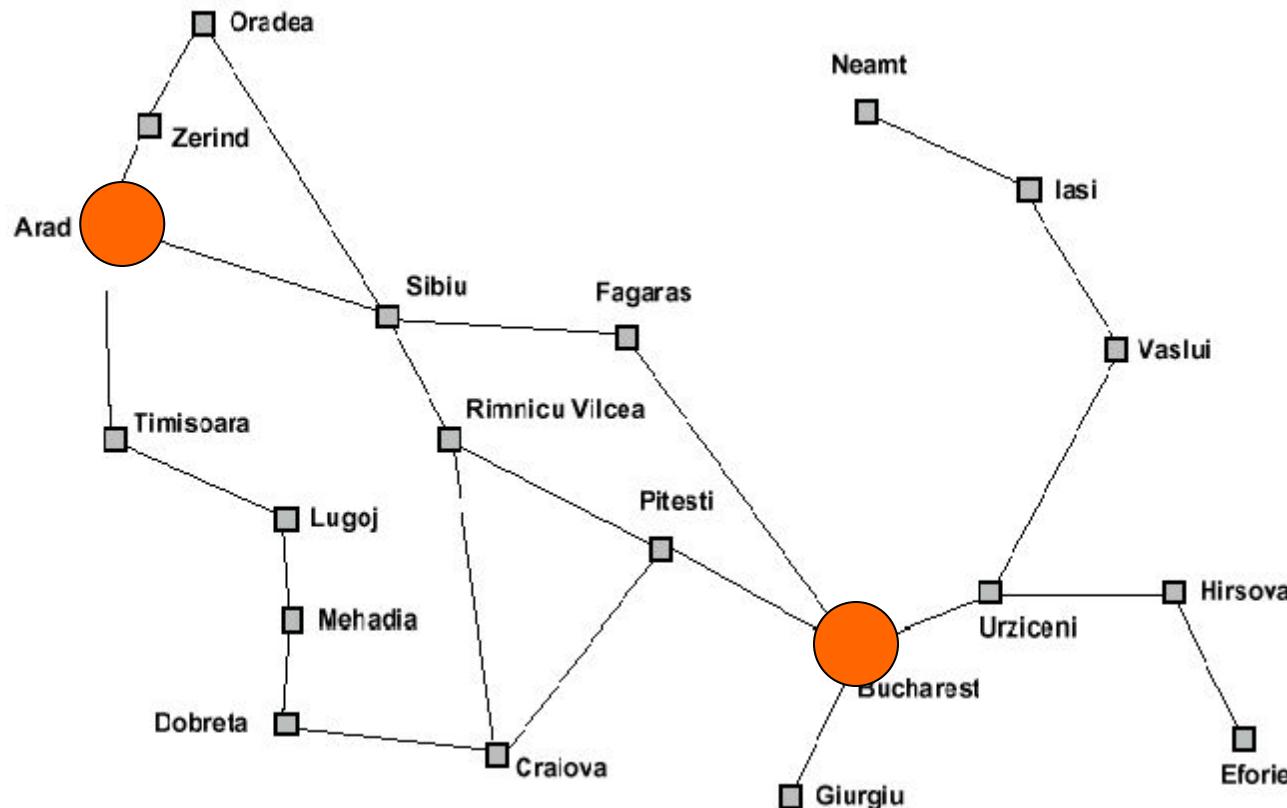
CS 561, Sessions 2-3

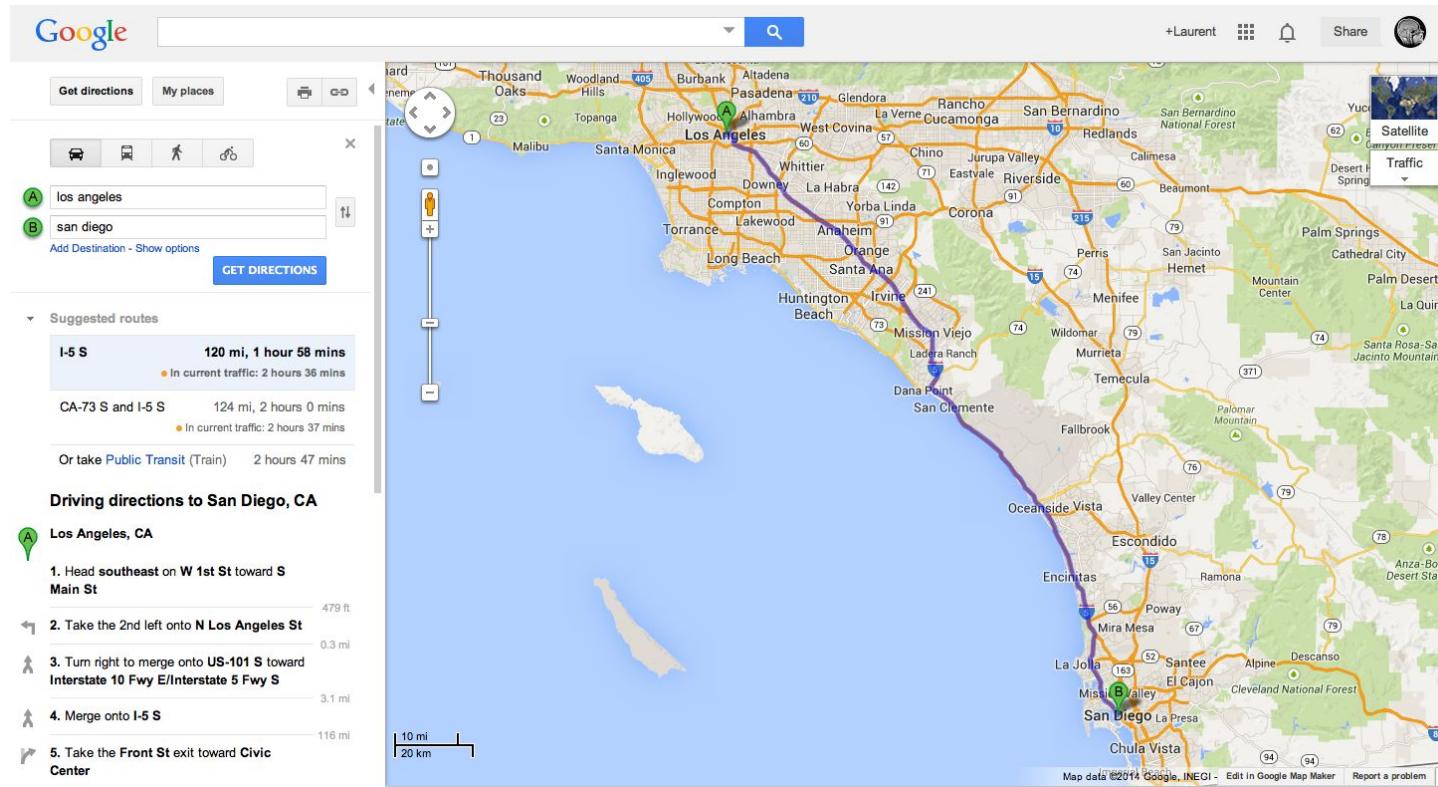


Example 2: Traveling in Romania

- In Romania, on vacation. Currently in Arad.
- Flight leaves tomorrow from Bucharest.
- **Formulate goal:**
 - Be in Bucharest
- **Formulate problem:**
 - States: various cities
 - Operators: drive between cities
- **Find solution:**
 - Sequence of cities, such that total driving distance is minimized.

Example: Traveling from Arad To Bucharest





The General Formulation of “Problem”

A *problem* is defined by four items:

initial state e.g., “at Arad”

operators (or successor function $S(x)$)

e.g., Arad → Zerind Arad → Sibiu etc.

goal test, can be

explicit, e.g., $x = \text{“at Bucharest”}$

implicit, e.g., $NoDirt(x)$

path cost (additive)

e.g., sum of distances, number of operators executed, etc.

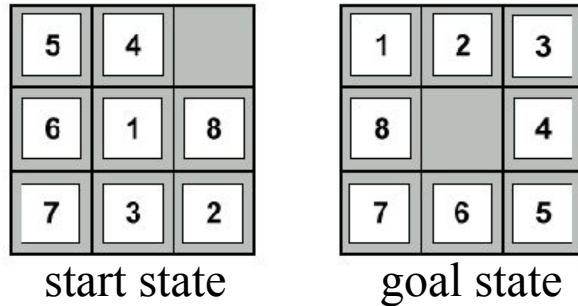
A *solution* is a sequence of operators

leading from the initial state to a goal state

Selecting a Space of States

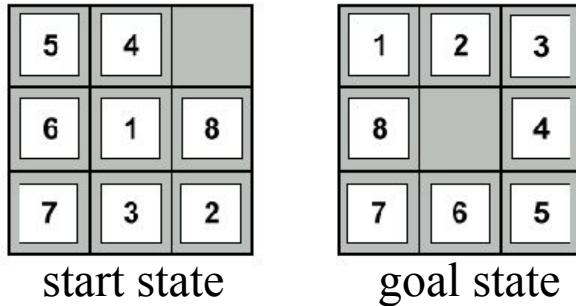
- Real world is absurdly complex; some abstraction is necessary to allow us to reason on it...
- Selecting the correct abstraction and resulting state space is a difficult problem!
- Abstract states \Leftrightarrow real-world states
- Abstract operators \Leftrightarrow sequences or real-world actions
(e.g., going from city i to city j costs L_{ij} \Leftrightarrow actually drive from city i to j)
- Abstract solution \Leftrightarrow set of real actions to take in the
real world such as to solve problem

Example 3: 8-puzzle



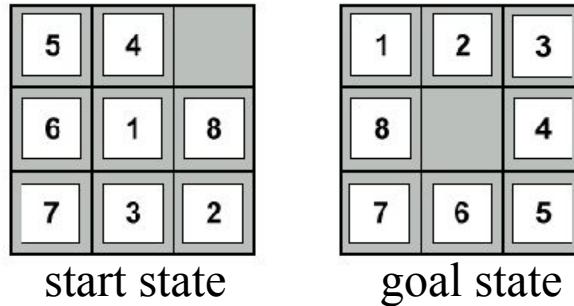
- State:
- Operators:
- Goal test:
- Path cost:

Example: 8-puzzle



- State: integer location of tiles (ignore intermediate locations)
- Operators: moving blank left, right, up, down (ignore jamming)
- Goal test: does state match goal state?
- Path cost: 1 per move

Example: 8-puzzle

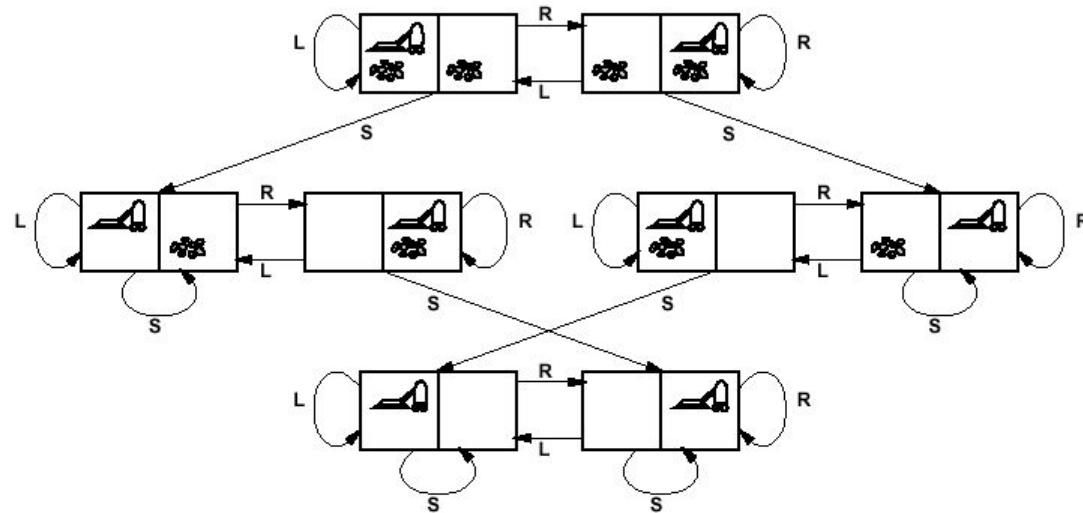


Why do we need search algorithms?

- 8-puzzle has $(9 \times 8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2) = 9! = 362,880$ states
- 15-puzzle has $15!$ approximately $\sim 10^{12}$ states
- 24-puzzle has $24!$ approximately $\sim 10^{25}$ states

So, we need a principled way to look for a solution in these huge search spaces...

Back to Vacuum World



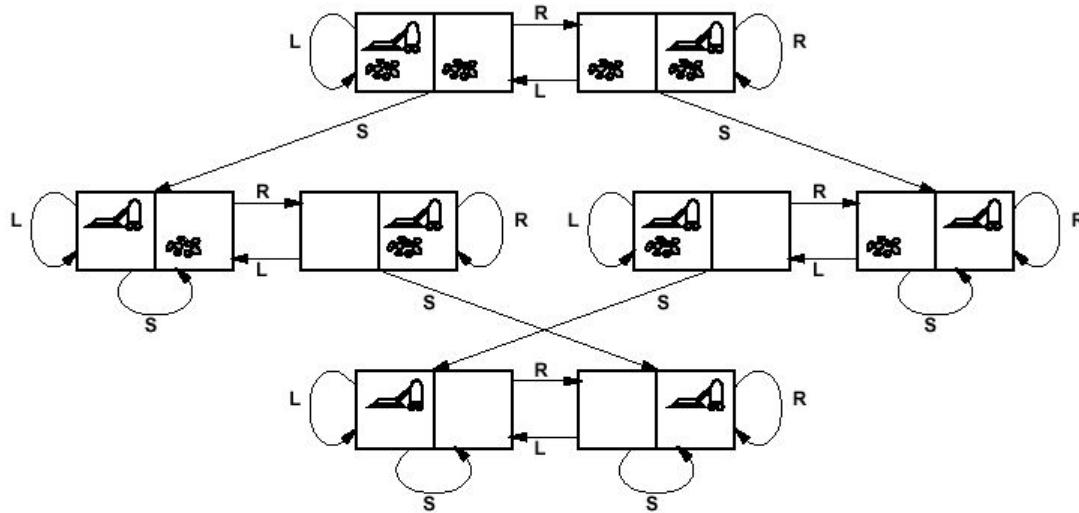
states??

operators??

goal test??

path cost??

Back to Vacuum World



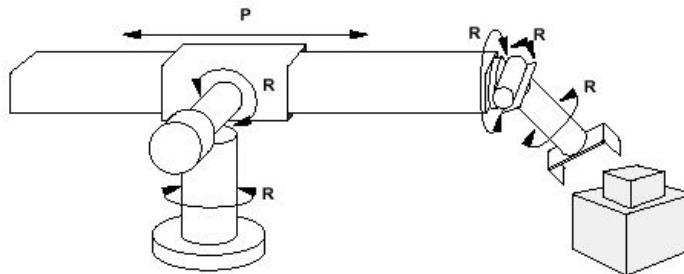
states??: integer dirt and robot locations (ignore dirt *amounts*)

operators??: *Left, Right, Suck*

goal test??: no dirt

path cost??: 1 per operator

Example: Robotic Assembly



states??: real-valued coordinates of
robot joint angles
parts of the object to be assembled

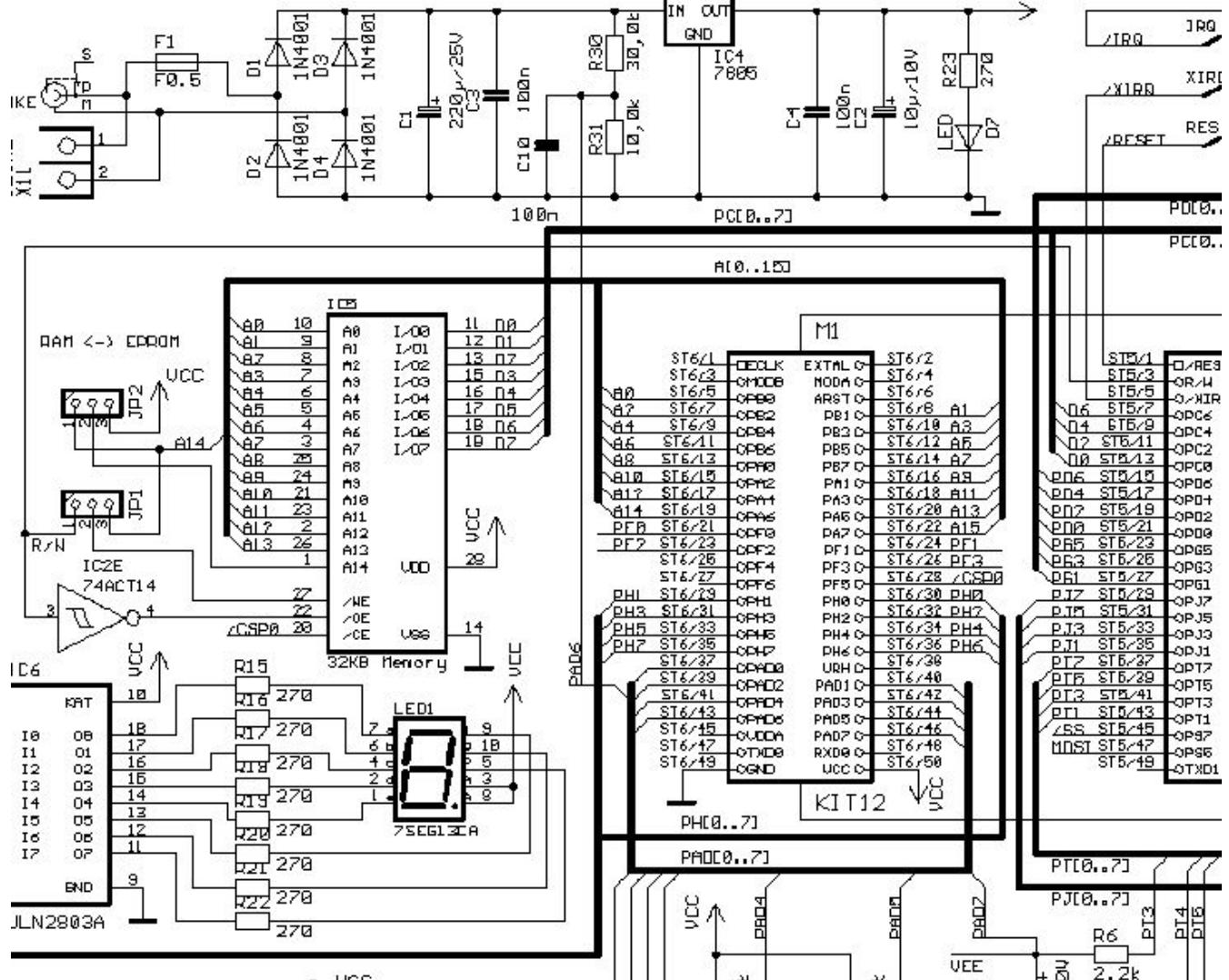
operators??: continuous motions of robot joints

goal test??: complete assembly *with no robot included!*

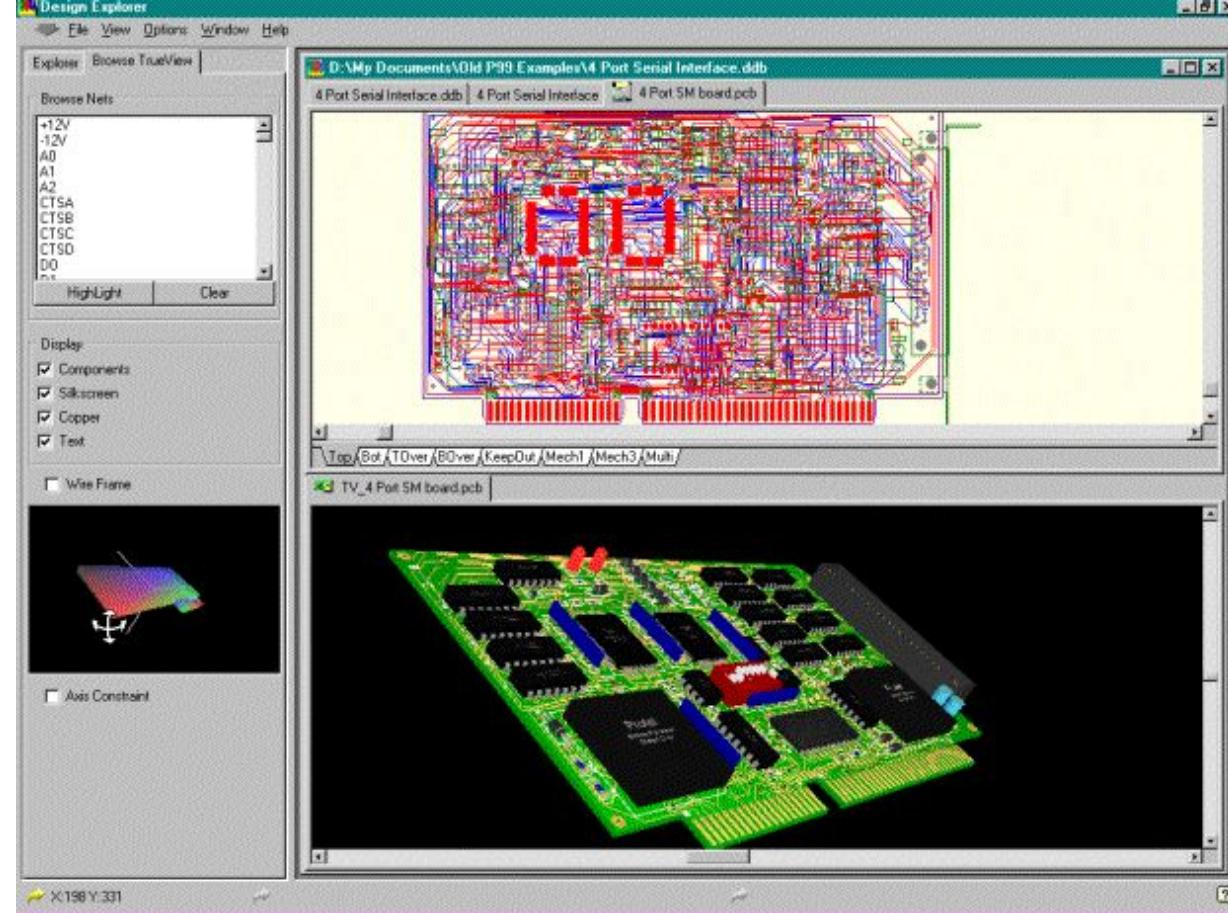
path cost??: time to execute

A Real-life Example: Circuit Board Layout

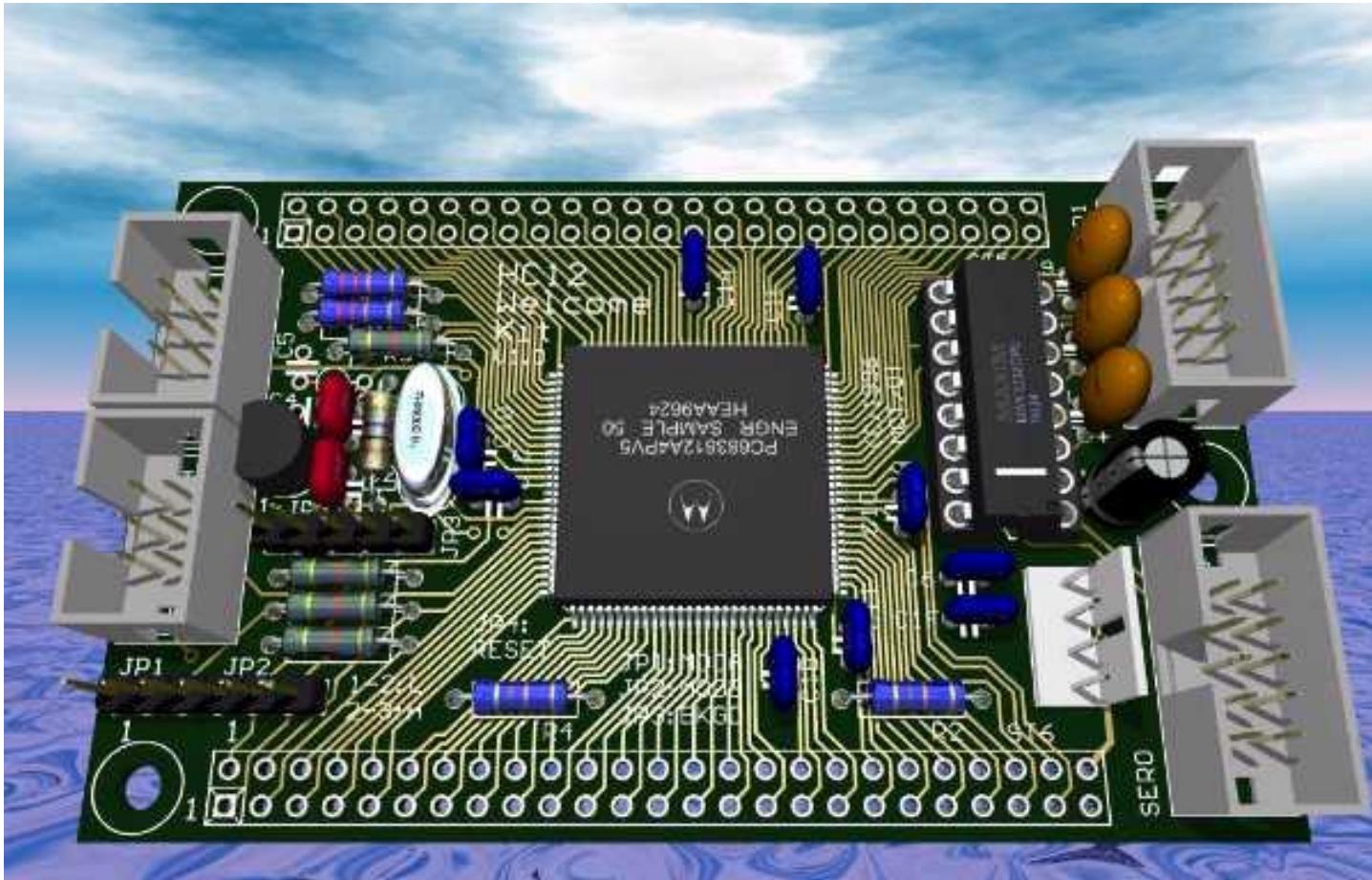
- Given schematic diagram comprising components (chips, resistors, capacitors, etc) and interconnections (wires), find optimal way to place components on a printed circuit board, under the constraint that only a small number of wire layers are available (and wires on a given layer cannot cross!)
- “optimal way”??
 - minimize surface area
 - minimize number of signal layers
 - minimize number of vias (connections from one layer to another)
 - minimize length of some signal lines (e.g., clock line)
 - distribute heat throughout board
 - etc.



Use automated tools to place components and route wiring.



Protel 99 SE's unique 3D visualization feature lets you see your finished board before it leaves your desktop. Sophisticated 3D modeling and extrusion techniques render your board in stunning 3D without the need for additional height information. Rotate and zoom to examine every aspect of your board.



SEARCH FOR SOLUTIONS

Search Algorithms

Basic idea:

offline, systematic exploration of simulated state-space by generating successors of explored states (expanding)

Function General-Search(*problem, strategy*) returns a *solution*, or failure

 initialize the search tree using the initial state problem

loop do

if there are no candidates for expansion **then return** failure

 choose a leaf node for expansion according to strategy

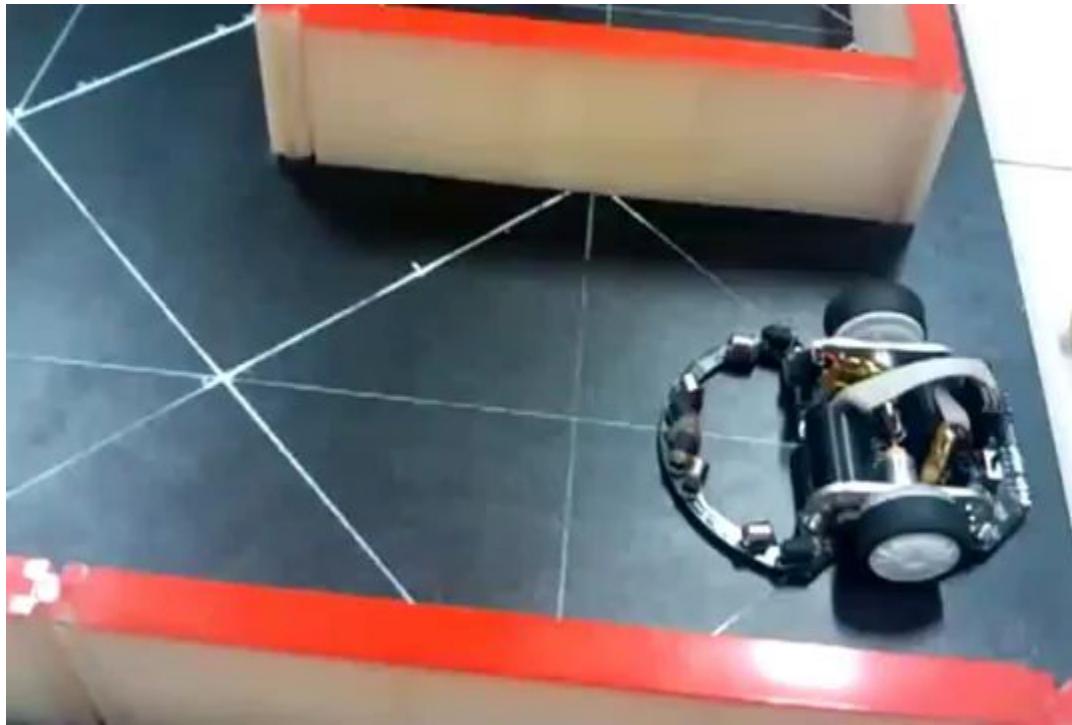
if the node contains a goal state **then**

return the corresponding solution

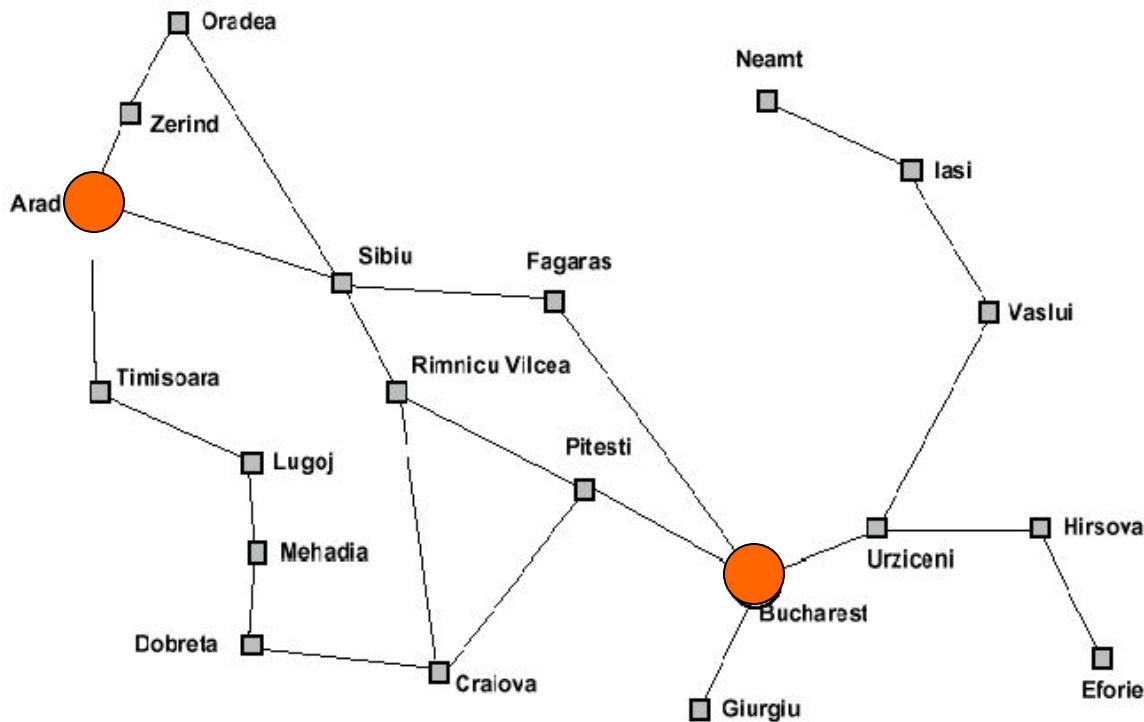
else expand the node and add resulting nodes to the search tree

end

Example: A micro_mouse searches in a maze



Example: Traveling from Arad To Bucharest

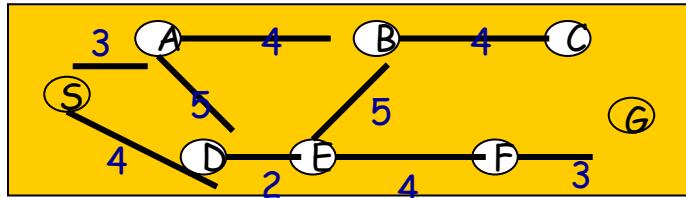


From Problem Space to Search Tree

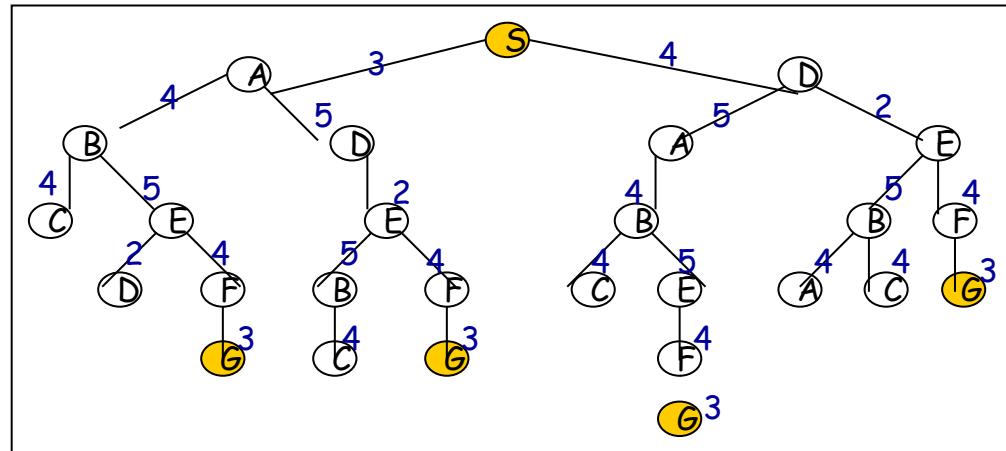
- Some material in this and following slides is from

<http://www.cs.kuleuven.ac.be/~dannyd/FAI/> check it out!

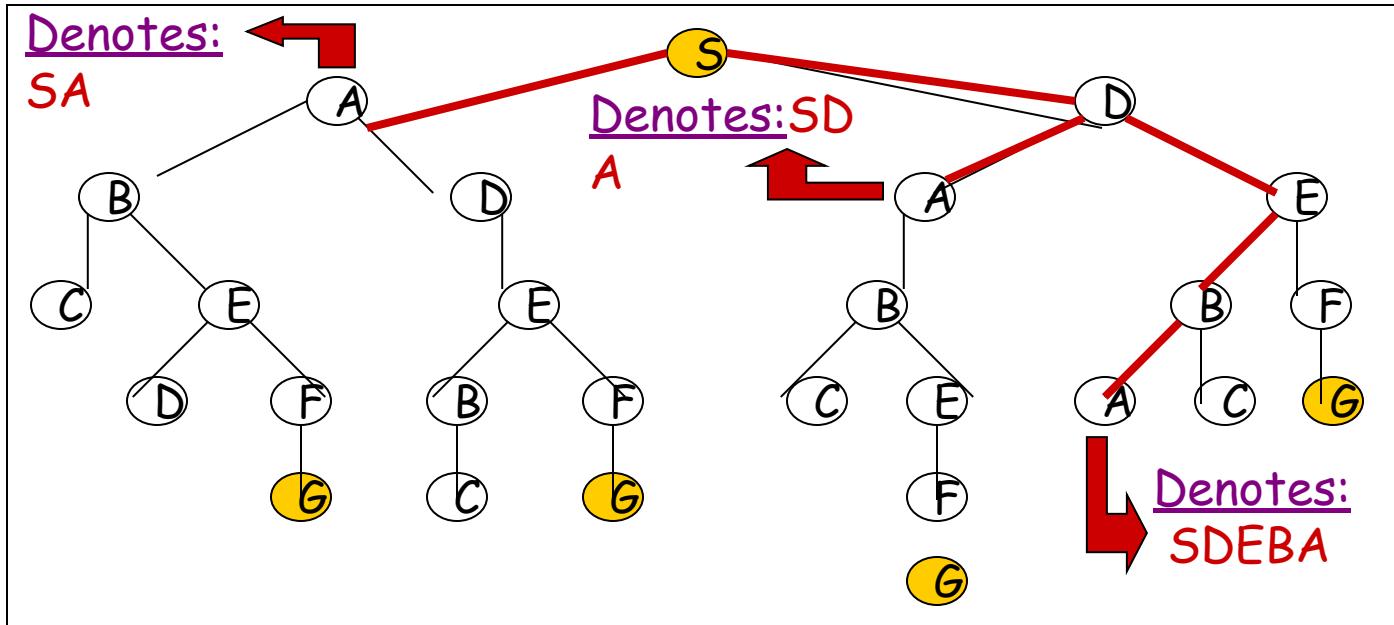
Problem space



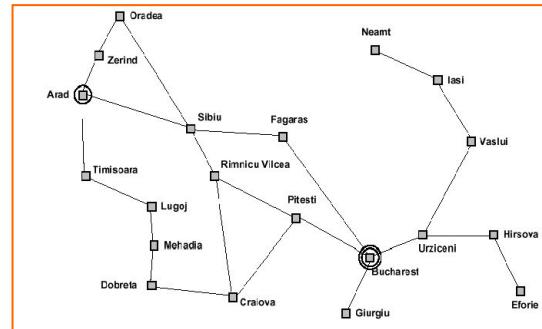
Associated
loop-free
search tree



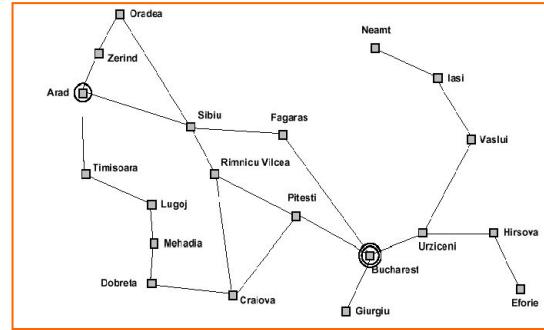
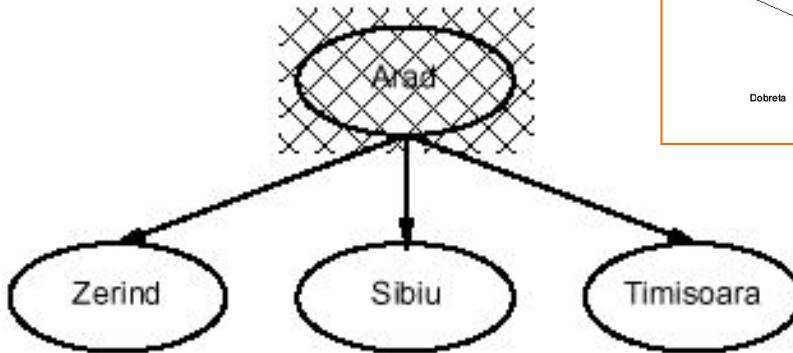
Paths in Search Trees



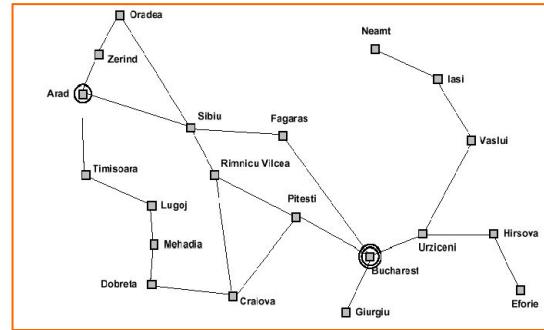
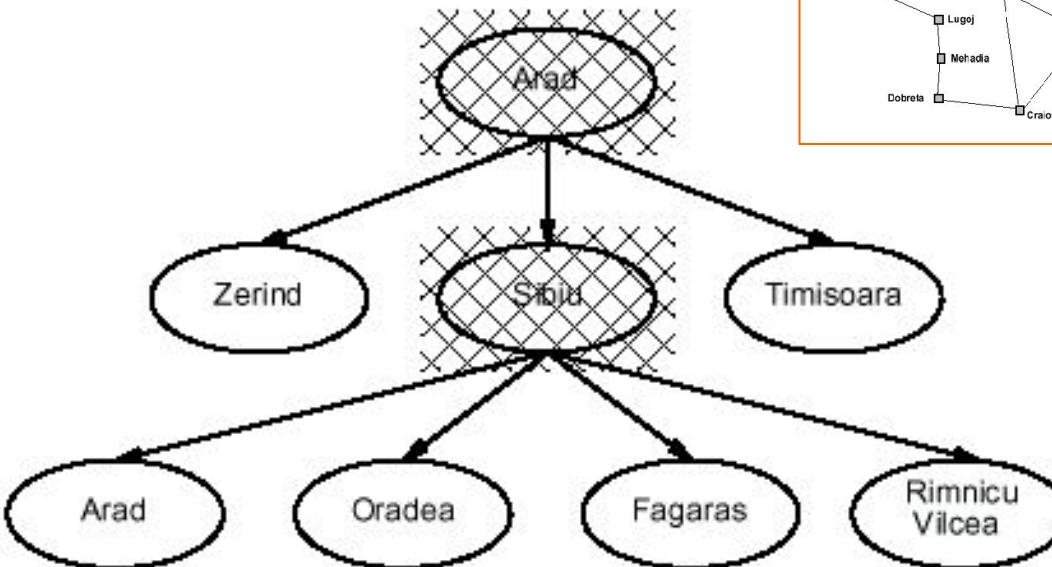
A General Search Example



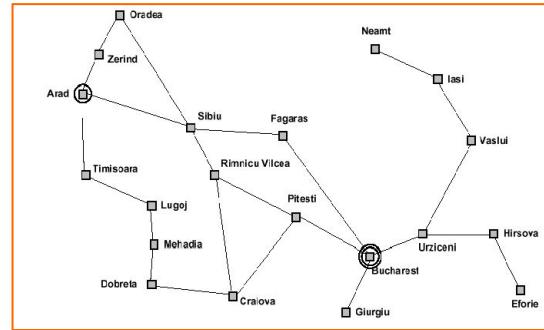
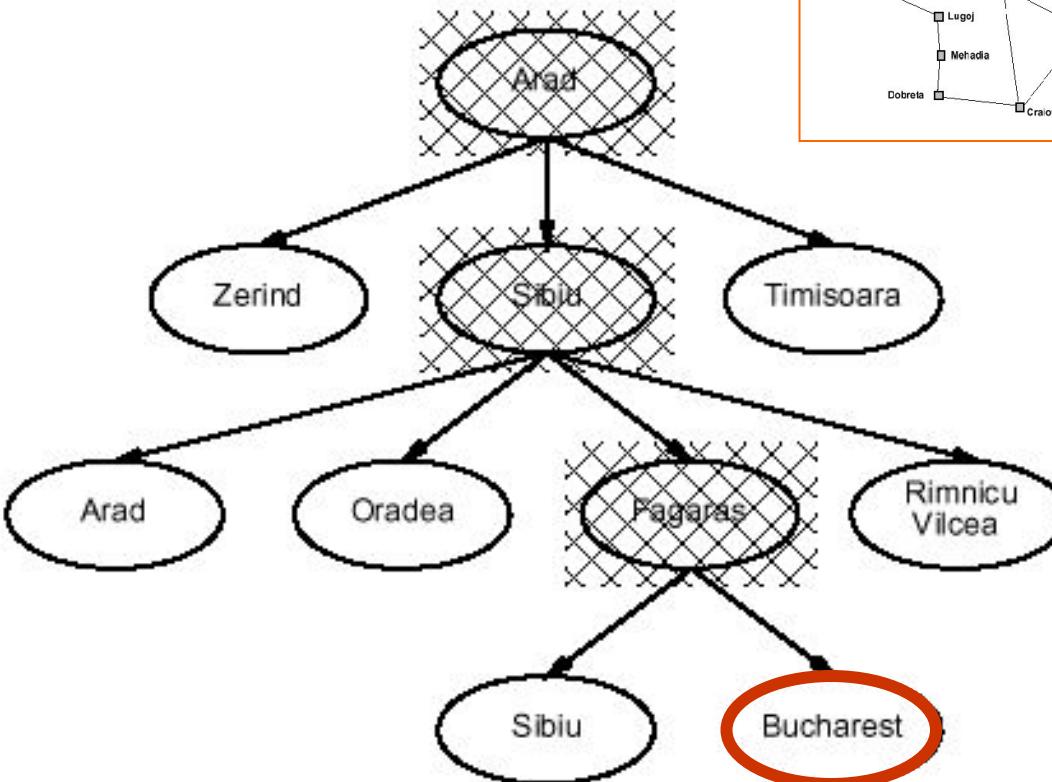
A General Search Example



A General Search Example



A General Search Example



An Implementation of Search Algorithms

```
Function General-Search(problem, Queuing-Fn) returns a solution, or failure
  nodes □ make-queue(make-node(initial-state[problem]))
  loop do
    if nodes is empty then return failure
    node □ Remove-Front(nodes)
    if Goal-Test[problem] applied to State(node) succeeds then return node
    nodes □ Queuing-Fn(nodes, Expand(node, Operators[problem]))
  end
```

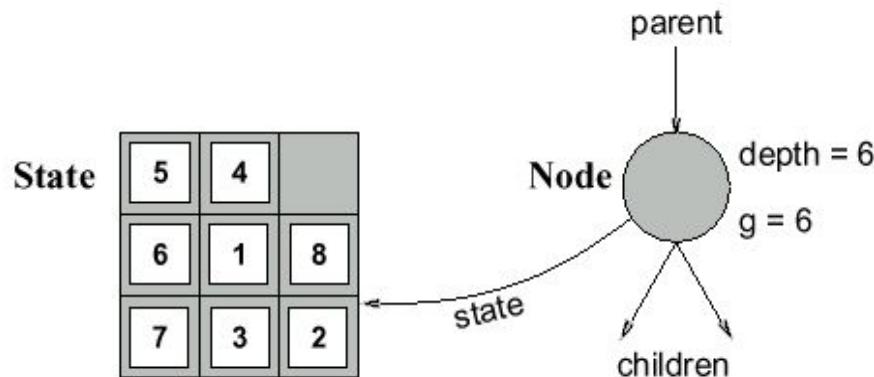
Queuing-Fn(*queue, elements*) is a queuing function that inserts a set of elements into the queue and determines the order of node expansion. Varieties of the queuing function produce varieties of the search algorithm.

Encapsulating *state* information in *nodes*

A *state* is a (representation of) a physical configuration

A *node* is a data structure constituting part of a search tree
includes *parent*, *children*, *depth*, *path cost* $g(x)$

States do not have parents, children, depth, or path cost!



The EXPAND function creates new nodes, filling in the various fields and using the OPERATORS (or SUCCESSORFn) of the problem to create the corresponding states.

Paths in search trees

1	3	4
7		6
5	8	2

1		4
7	3	6
5	8	2

states

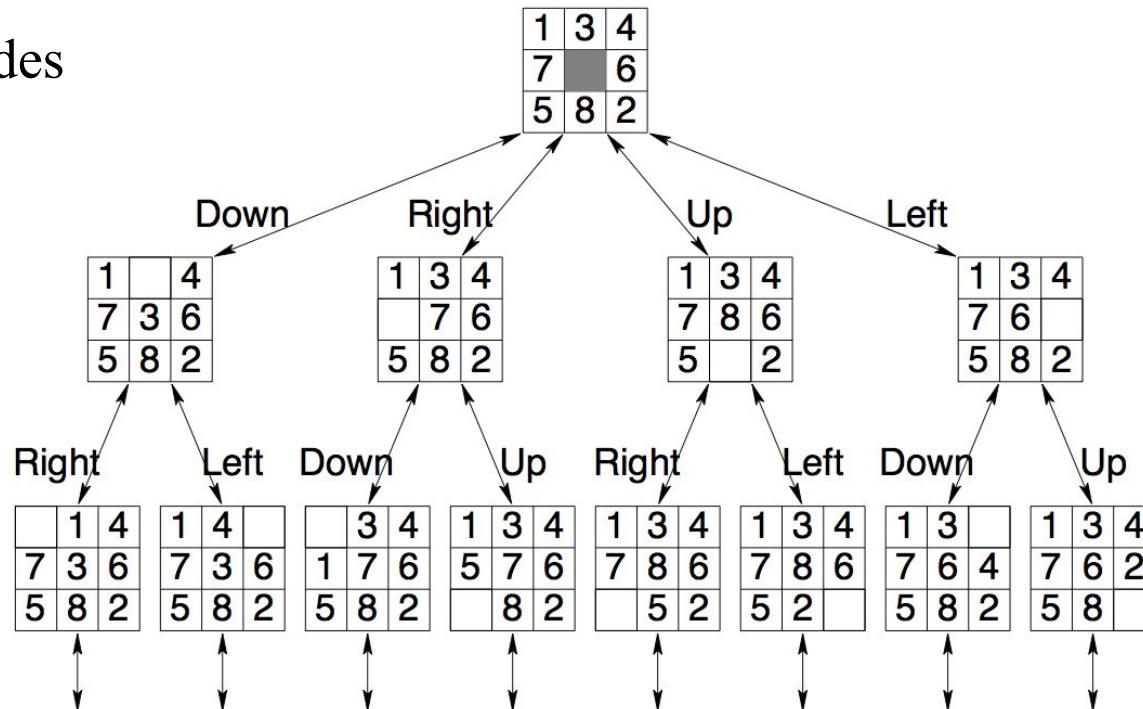
1	3	4
	7	6
5	8	2

1	3	4
7	8	6
5		2

...

Paths in search trees

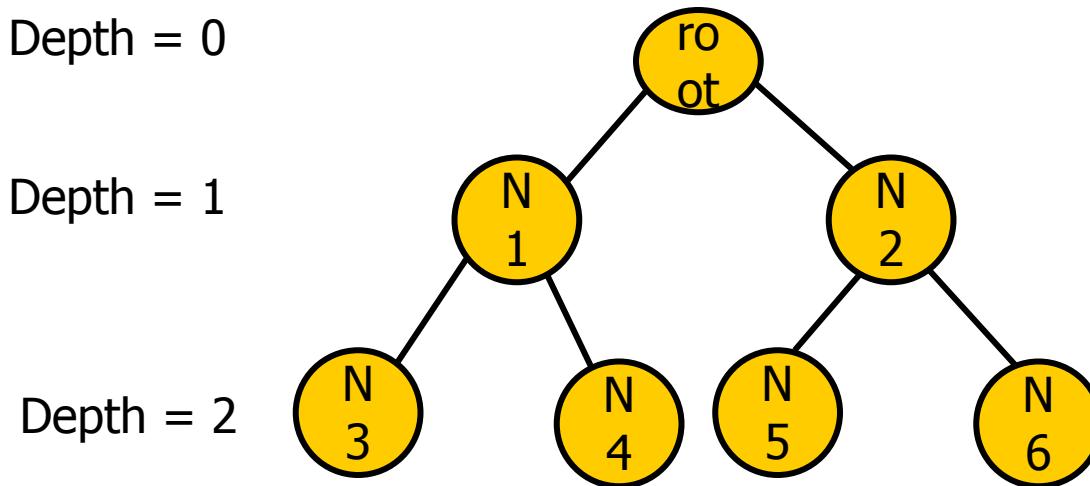
Search tree nodes



Evaluation of search strategies

- A search strategy is defined by picking the order of node expansion
- Search algorithms are commonly evaluated according to the following four criteria:
 - **Completeness:** does it always find a solution if one exists?
 - **Time Complexity:** how long does it take as function of num. of nodes?
 - **Space Complexity:** how much memory does it require?
 - **Optimality:** does it guarantee the least-cost solution?
- Time and space complexity are measured in terms of:
 - b – max branching factor of the search tree
 - d – depth of the least-cost solution
 - m – max depth of the search tree (may be infinity)

Binary Tree Example



Number of nodes at max depth: $n = 2^{\text{max depth}}$

Number of levels (given n at max depth) = $\log_2(n)$

COMPLEXITY OF PROBLEMS

Complexity

- Why worry about complexity of algorithms?
 - because a problem may be solvable in principle but may take too long to solve in practice

Complexity: Tower of Hanoi

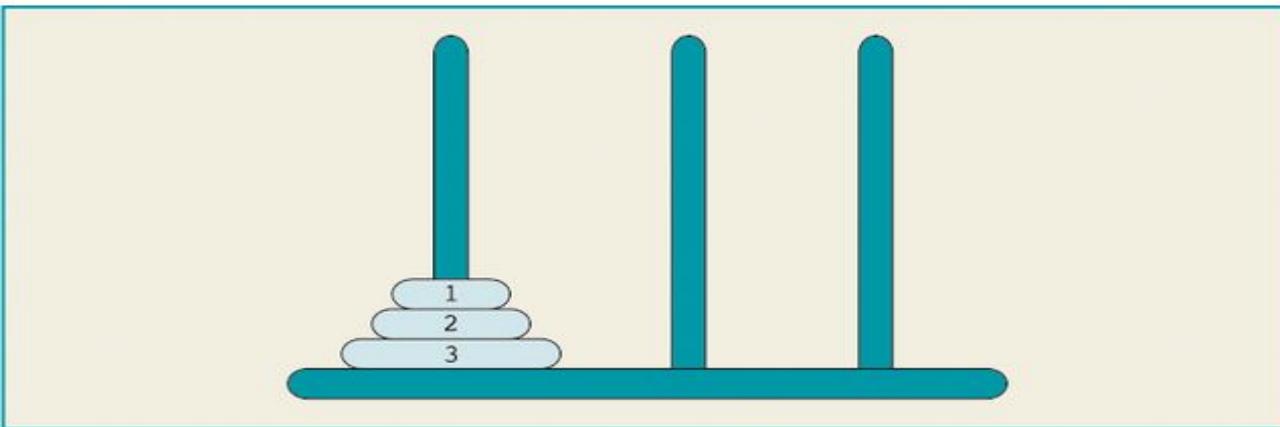


Figure 11-6 Tower of Hanoi problem with three disks

Complexity: Tower of Hanoi

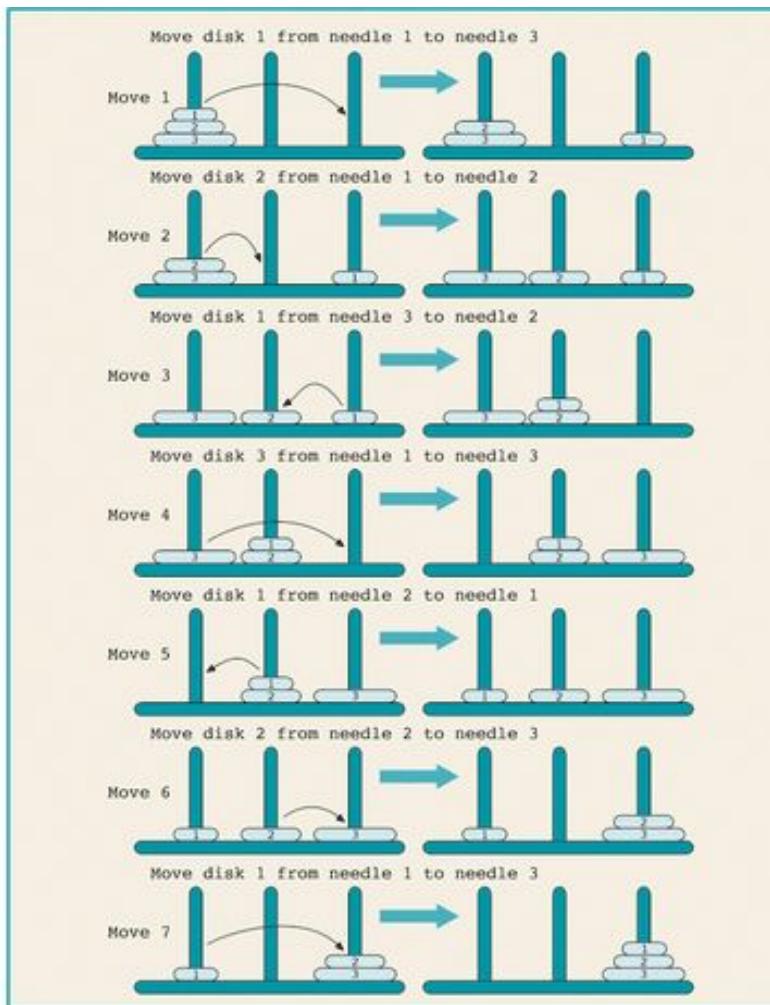


Figure 11-7 Solution of Tower of Hanoi problem with three disks

Complexity: Tower of Hanoi

- 3-disk problem: $2^3 - 1 = 7$ moves
- 64-disk problem: $2^{64} - 1$.
 - $2^{10} = 1024 \approx 1000 = 10^3$,
 - $2^{64} = 2^4 * 2^{60} \approx 2^4 * 10^{18} = 1.6 * 10^{19}$
- One year $\approx 3.2 * 10^7$ seconds

Complexity: Tower of Hanoi

- The wizard's speed = one disk / second

$$1.6 * 10^{19} = 5 * 3.2 * 10^{18} =$$

$$5 * (3.2 * 10^7) * 10^{11} =$$

$$(3.2 * 10^7) * (5 * 10^{11})$$

500 billion years

Complexity: Tower of Hanoi

- The time required to move all 64 disks from needle 1 to needle 3 is roughly $5 * 10^{11}$ years.
- It is estimated that our universe is about $15 \text{ billion} = 1.5 * 10^{10}$ years old.

$$5 * 10^{11} = 50 * 10^{10} \approx \textcolor{red}{33} * (1.5 * 10^{10}).$$

Complexity: Tower of Hanoi

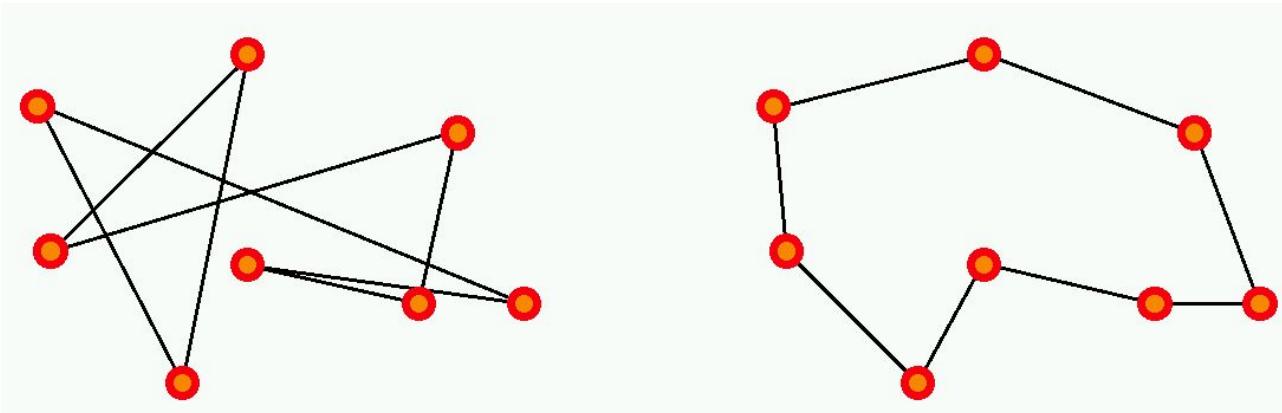
- Assume: a computer with 1 billion = 10^9 moves/second.
 - $\text{Moves/year} = (3.2 \times 10^7) \times 10^9 = 3.2 \times 10^{16}$
- To solve the problem for 64 disks:
 - $2^{64} \approx 1.6 \times 10^{19} = 1.6 \times 10^{16} \times 10^3 = (3.2 \times 10^{16}) \times 500$
 - 500 years for the computer to generate 2^{64} moves at the rate of 1 billion moves per second.

Complexity

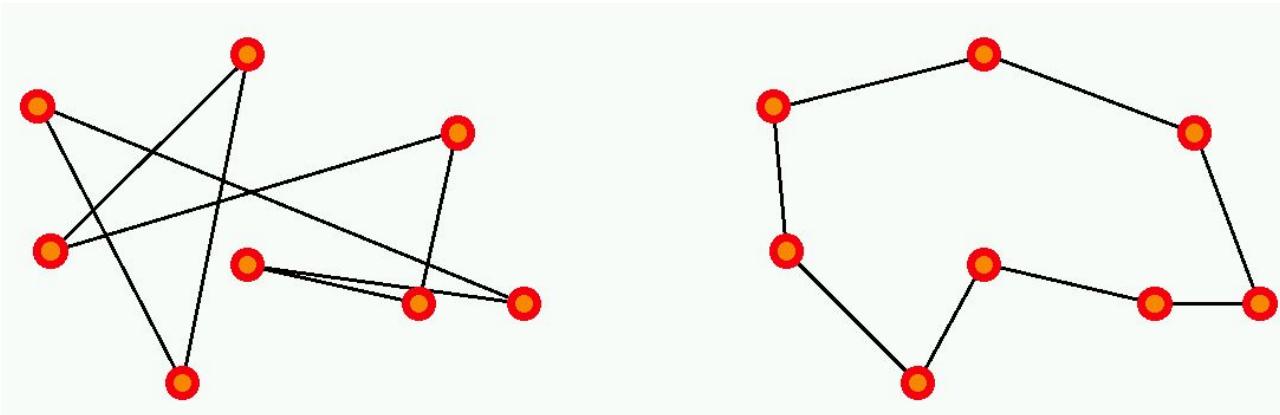
- Why worry about complexity of algorithms?
 - because a problem may be solvable in principle but may take too long to solve in practice
- How can we evaluate the complexity of algorithms?
 - through asymptotic analysis, i.e., estimate time (or number of operations) necessary to solve an instance of size n of a problem when n tends towards infinity
 - See AIMA, Appendix A.

Complexity example: Traveling Salesman Problem

- There are n cities, with a road of length L_{ij} joining city i to city j .
- The salesman wishes to find a way to visit all cities that is optimal in two ways:
each city is visited only once, and
the total route is as short as possible.



Complexity example: Traveling Salesman Problem



This is a *hard* problem: the only known algorithms (so far) to solve it have exponential complexity, that is, the number of operations required to solve it grows as $\exp(n)$ for n cities.

Why is exponential complexity “hard”?

It means that the number of operations necessary to compute the exact solution of the problem grows exponentially with the size of the problem (here, the number of cities).

- $\exp(1) = 2.72$
- $\exp(10) = 2.20 \cdot 10^4$ (daily salesman trip)
- $\exp(100) = 2.69 \cdot 10^{43}$ (monthly salesman planning)
- $\exp(500) = 1.40 \cdot 10^{217}$ (music band worldwide tour)
- $\exp(250,000) = 10^{108,573}$ (fedex, postal services)
- Fastest computer = 10^{12} operations/second

So...

In general, exponential-complexity problems *cannot be solved for any but the smallest instances!*

Complexity

- **Polynomial-time (P) problems:** we can find algorithms that will solve them in a time (=number of operations) that grows polynomially with the size of the input.
- **for example:** sort n numbers into increasing order: poor algorithms have n^2 complexity, better ones have $n \log(n)$ complexity.

Complexity

- Since we did not state what the order of the polynomial is, it could be very large! Are there algorithms that require more than polynomial time?
- Yes (until proof of the contrary); for some algorithms, we do not know of any polynomial-time algorithm to solve them. These belong to the class of **nondeterministic-polynomial-time (NP)** algorithms (which includes P problems as well as harder ones).
 - **for example:** traveling salesman problem.
- In particular, exponential-time algorithms are believed to be NP.

Note on NP-hard problems

- The formal definition of NP problems is:

A problem is **nondeterministic polynomial** if there exists some algorithm that can guess a solution and then verify whether or not the guess is correct in polynomial time.

(one can also state this as these problems being solvable in polynomial time on a nondeterministic Turing machine.)

In practice, until proof of the contrary, this means that known algorithms that run on known computer architectures will take more than polynomial time to solve the problem.

Complexity: $O()$ and $o()$ measures (Landau symbols)

- How can we represent the complexity of an algorithm?
- Given: Problem input (or instance) size: n
Number of operations to solve problem: $f(n)$
- If, for a given function $g(n)$, we have: // for some k

$$\exists k \in \mathbb{R}, \exists n_0 \in \mathbb{N}, \forall n \in \mathbb{N}, n \geq n_0, f(n) \leq kg(n)$$

then

$$f \in O(g)$$

- If, for a given function $g(n)$, we have: // for all k

$$\forall k \in \mathbb{R}, \exists n_0 \in \mathbb{N}, \forall n \in \mathbb{N}, n \geq n_0, f(n) \leq kg(n)$$

then

$$f \in o(g)$$

Landau symbols

$$f \in O(g) \Leftrightarrow \exists k, \underline{\lim_{n \rightarrow \infty}} f(n) \leq kg(n) \Leftrightarrow \frac{f}{g} \text{ is bounded}$$

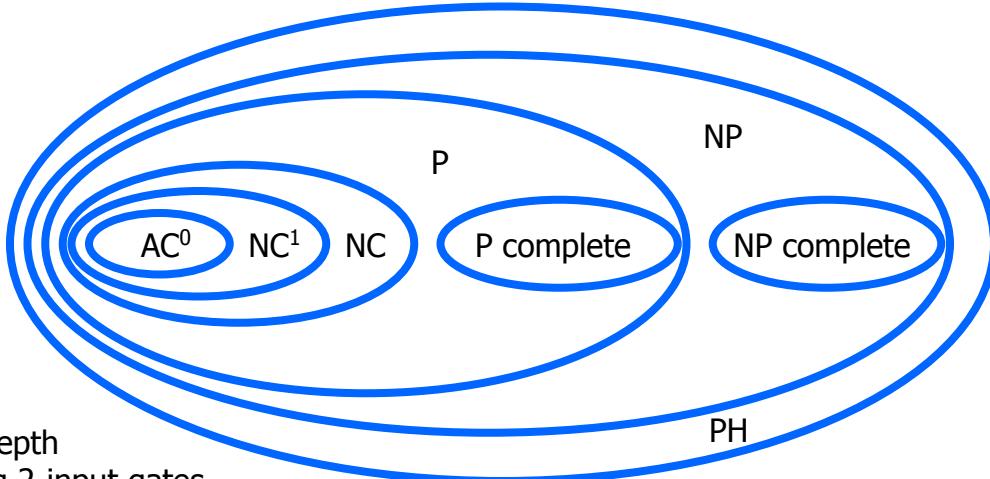
$$f \in o(g) \Leftrightarrow \forall k, \underline{\lim_{n \rightarrow \infty}} f(n) \leq kg(n) \Leftrightarrow \frac{f(n)}{g(n)} \xrightarrow{n \rightarrow \infty} 0$$

Examples, Properties

- $f(n)=n, g(n)=n^2$:
 n is $o(n^2)$, because $n/n^2 = 1/n \rightarrow 0$ as $n \rightarrow \infty$
similarly, $\log(n)$ is $o(n)$
 n^C is $o(\exp(n))$ for any C
- if f is $O(g)$, then for any K , $K*f$ is also $O(g)$; idem for $o()$
- if f is $O(h)$ and g is $O(h)$, then for any K, L : $(K*f + L*g)$ is $O(h)$
idem for $o()$
- if f is $O(g)$ and g is $O(h)$, then f is $O(h)$
- if f is $O(g)$ and g is $o(h)$, then f is $o(h)$
- if f is $o(g)$ and g is $O(h)$, then f is $o(h)$

Polynomial-time hierarchy

See Handbook of Brain Theory & Neural Networks
(Arbib, ed.; MIT Press 1995).



AC⁰: can be solved using gates of constant depth

NC¹: can be solved in logarithmic depth using 2-input gates

NC: can be solved by small, fast parallel computer

P: can be solved in polynomial time

P-complete: hardest problems in P; if one of them can be proven to be NC, then P = NC

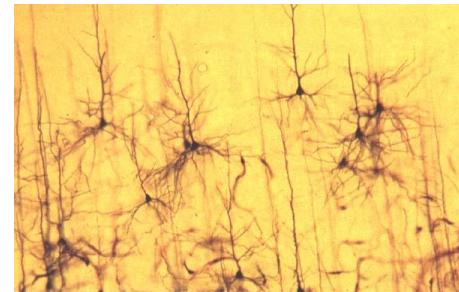
NP: nondeterministic-polynomial algorithms

NP-complete: hardest NP problems; if one of them can be proven to be P, then NP = P

PH: polynomial-time hierarchy

Complexity and Human Brain

- Are computers close to human brain power?
- Current computer chip (CPU):
 - 10^3 inputs (pins)
 - 10^7 processing elements (gates)
 - 2 inputs per processing element (fan-in = 2)
 - processing elements compute boolean logic (OR, AND, NOT, etc)
- Typical human brain:
 - 10^7 inputs (sensors)
 - 10^{10} processing elements (neurons)
 - fan-in = 10^3
 - processing elements compute complicated functions



Still a lot of improvement needed for computers; but
computer clusters come close!

Summary

- This Week:
 - Problem formulation usually requires **abstracting away real-world details** to define a **state space** that can be explored using computer algorithms.
 - Once problem is formulated in abstract form, **complexity analysis** helps us picking out best algorithm to solve problem.
- Next Week:
 - Variety of uninformed search strategies; difference lies in method used to **pick node that will be further expanded**.
 - **Iterative deepening** search only uses linear space and not much more time than other uninformed search strategies.

CSCI 561 - Foundation for Artificial Intelligence

DISCUSSION SECTION (WEEK 1)

PROF WEI-MIN SHEN WMSHEN@USC.EDU

OUTLINE

Introduction of myself

Overview of Course Discussion

History (Success) of Artificial Intelligence

Agents and Environments

Evaluation Criteria of AI

Applications of AI

What You Should Know This Week

USC POLYMORPHIC ROBOTICS

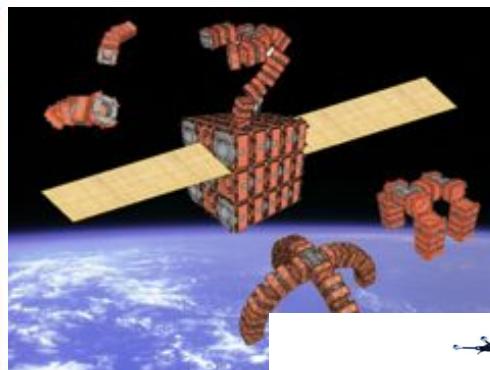
LAB

<https://robots.isi.edu/>

Transformer Robots: Self-Reconfigurable & Modular



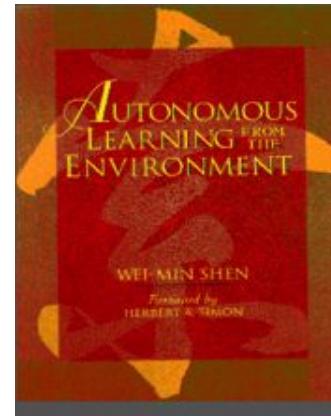
Welcome



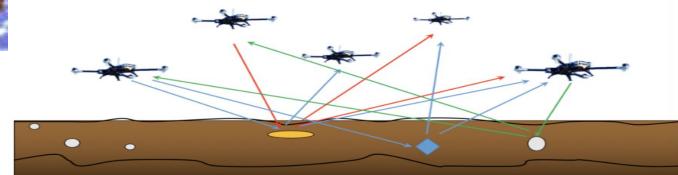
Projects

We conduct research in **adaptive, self-reconfigurable, autonomous robots and systems**, including **StarCell**, modular, multifunctional and self-reconfigurable **SuperBot**, Hormone-

Surprise-Based Learning



Teaching
CS561 (AI)
CS360 (AI)
INF552
INF553



SURPRISE-BASED LEARNING

Herbert A. Simon (1916-2001)

- Nobel Price Winner (AI, Machine Discovery)

Wei-Min Shen (1983 – now)

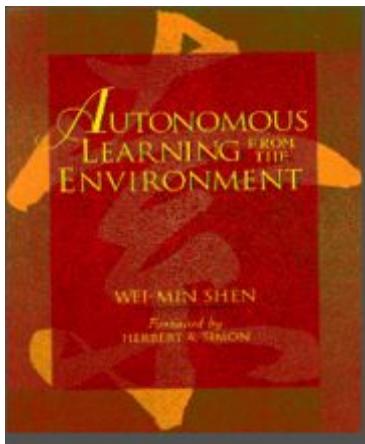
- Autonomous Learning from the Environment

Nadeesha Ranasinghe (2005 – now)

- Learn and predict unexpected changes

Thomas J. Collins (2013 – now)

- Discovery of hidden/latent structures



Surprise-Based Learning

by
Nadeesha Oliver Ranasinghe

Ph.D. Dissertation Proposal Guidance Committee:

Dr. Yu-Han Chang

Dr. Laurent Itti

Dr. Ramakant Nevatia

Dr. Michael Safronov

Dr. Wei-Min Shen (Chair)

USCViterbi
School of Engineering
Information Sciences

Active State Learning from Surprises in Stochastic and Partially-observable Environments

Thomas Joseph Collins

Defense Committee:

Prof. Wei-Min Shen (Chair)

Prof. Paul Rosenbloom

Prof. John Carlsson (Outside member)

A portrait of Professor Herbert A. Simon is shown on the left, and a quote from his work "Forecasting the Future or Shaping it?" is on the right.

AI magazine
VOLUME 21, NUMBER 4
October 2000

Herb A. Simon
1916 - 2001

Forecasting the Future or Shaping it?
October 19, 2000

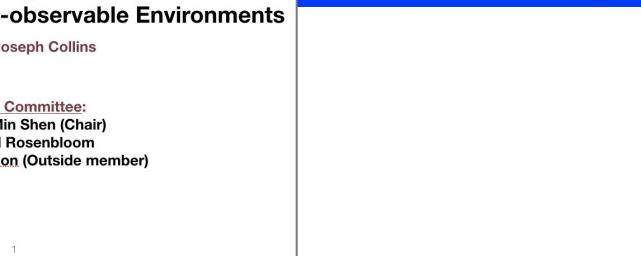
Our task is not to *predict* the future; our task is to *design* a future for a sustainable and acceptable world, and then to devote our efforts to bringing that future about.

Professor Herbert A. Simon
Nobel Price Laureate
A Founder of Artificial Intelligence

Prof Wei-Min Shen <http://www.isi.edu/robots>

Self-Reconfigurable Modular Robots
(System of Systems)

Surprise-Based Learning



CS561: ARTIFICIAL INTELLIGENCE

Course overview: foundations of symbolic intelligent systems. Agents, search, problem solving, logic, representation, reasoning, symbolic programming, and robotics.

Prerequisites: Good programming and algorithm analysis skills. Basic probability theory desirable.

Textbook:

Russell & Norvig, *Artificial Intelligence: A Modern Approach 4th Edition*

Wei-Min Shen, *Autonomous Learning from the Environment, (selected readings)*.

CS561: ARTIFICIAL INTELLIGENCE

Discussion Sections

- Provide more details, discussion and new material to augment lectures
- Run algorithms on more complex examples than during lectures
- Relate lecture concepts to latest research topics
- Showcase cool demos of recent A.I. achievements

NOTE:

- You will be responsible for material presented in lecture *and* discussion sections

AI SUCCESS: DEEP BLUE

In 1997 Deep Blue became the first machine to win a match against a reigning world chess champion (by 3.5-2.5)



Game viewer - Kasparov vs. Deep Blue

File Help

GAME 6

MATCH SCORE	3.5	2.5
Deep Blue	Kasparov	
00:08:25	00:08:20	

white black

0	Prelude	
1	e4	c6
2	d4	d5
3	Nc3	dxe4
4	Nxe4	Nd7
5	Ng5	Ngf6
6	Bd3	e6
7	Nf3	h6
8	Nxe6	Qe7
9	O-O	fxe6

IBM



◀ ▶ ▶ ▶

DEEP BLUE COMBINED

Parallel and special purpose hardware



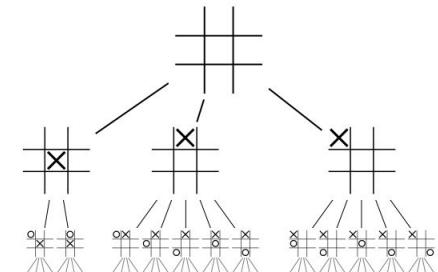
- A 30-node IBM RS/6000, enhanced with 480 special purpose VLSI chess chips

A heuristic game-tree search algorithm

- Capable of searching 200M positions/sec
- Searched 6-12 moves deep on average, sometimes to 40

Chess knowledge

- An opening book of 4K positions and 700K GM games
- An endgame database for when only 5-6 pieces left
- A positional evaluation function with 8K parts and many parameters that were tuned by learning over thousands of Master games



FIRST ROBOCUP (1997)



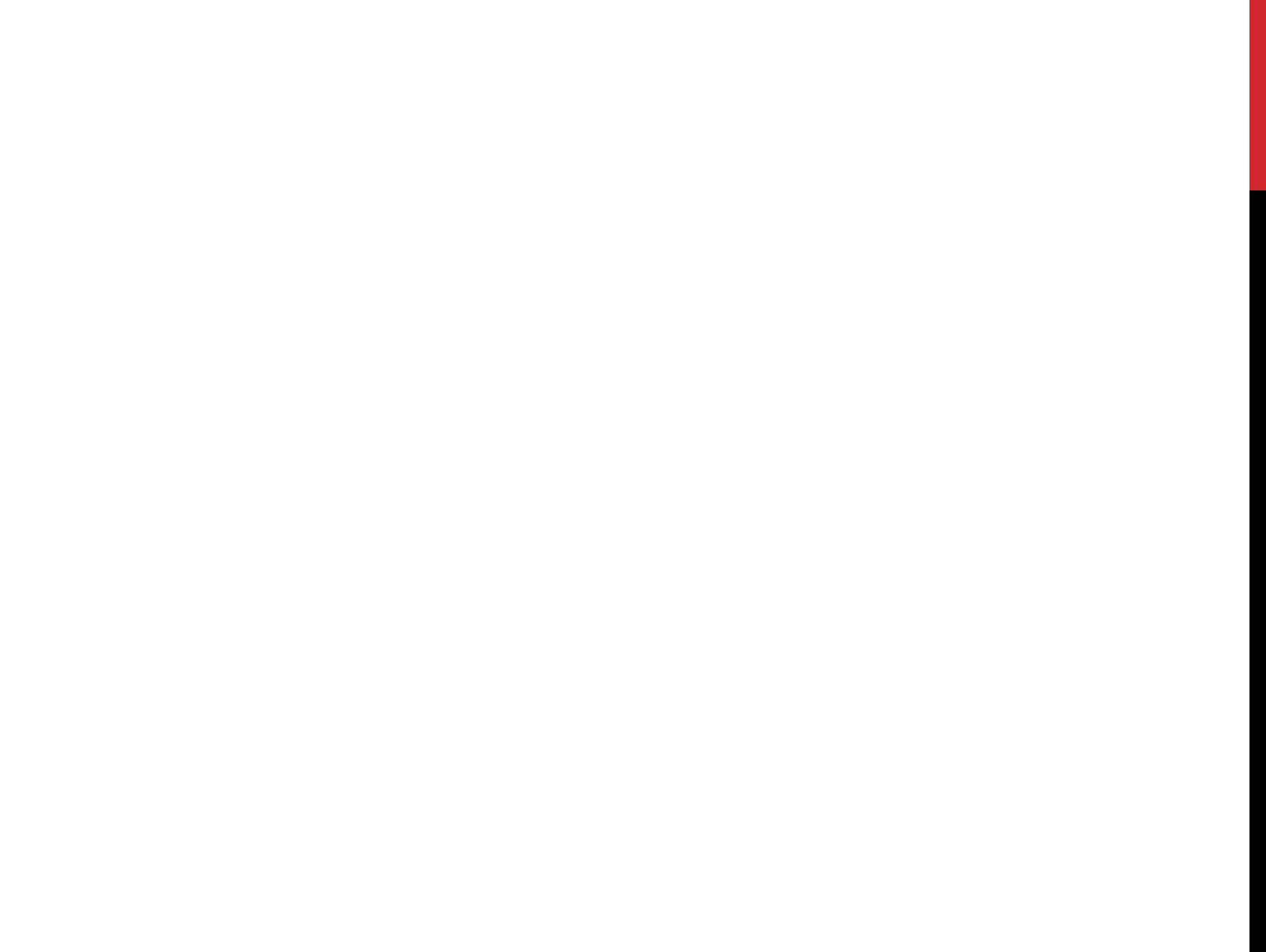


SONY AIBO ROBOT DOG LEAGUE



aibo

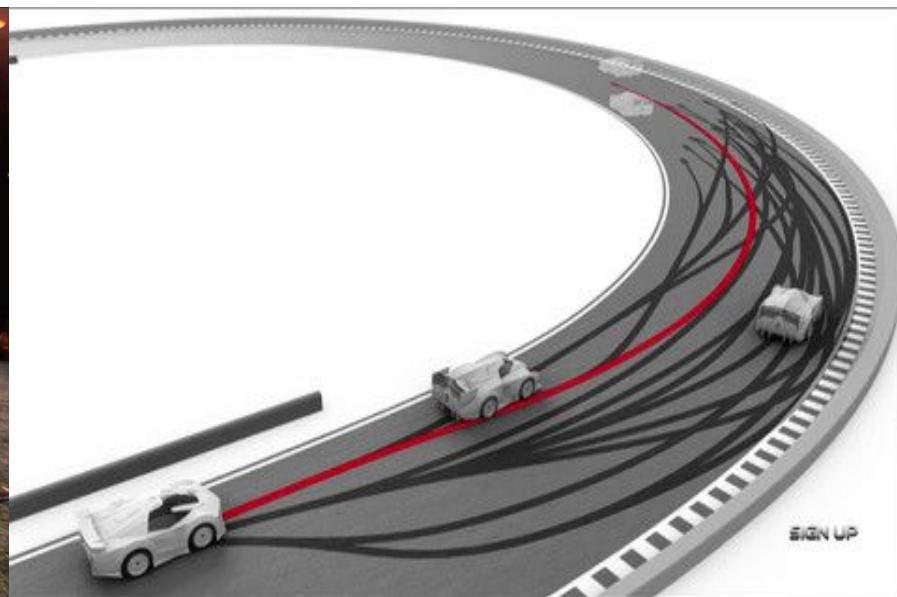
SONY



DARPA URBAN CHALLENGE (2007)



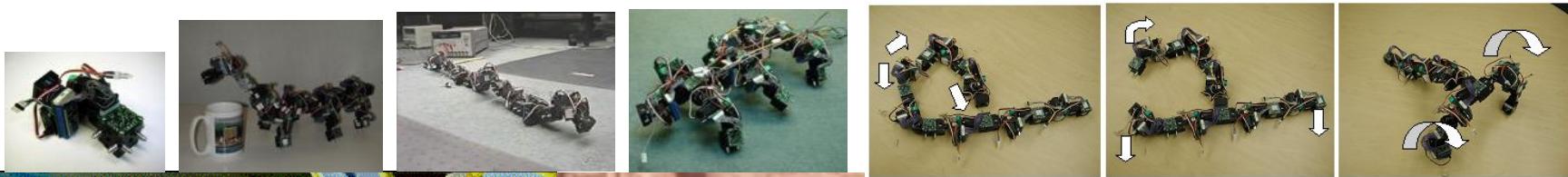
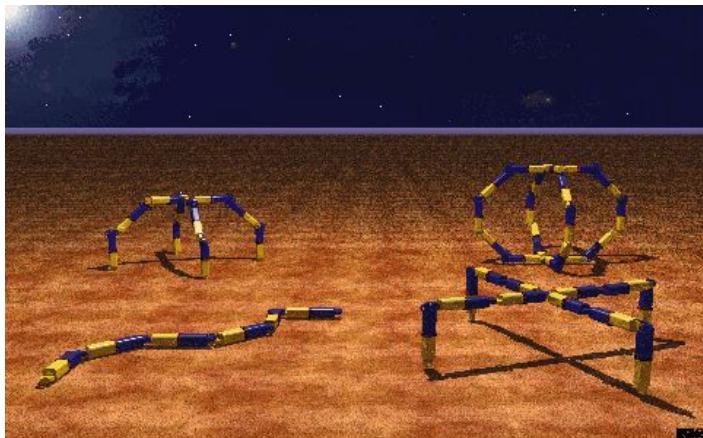
SELF-DRIVING CAR



DARPA Grand Challenge for Robotics



USC SELF-RECONFIGURABLE SUPERBOT



SUPERBOT VISION FOR SPACE

MODULAR, MULTIFUNCTION, SELF-RECONFIGURATION



USC SELF-RECONFIGURABLE “SUPERBOT”



USC Polymorphic Robotics Lab Openings for Students

Time: 2024 Fall through 2025 Spring

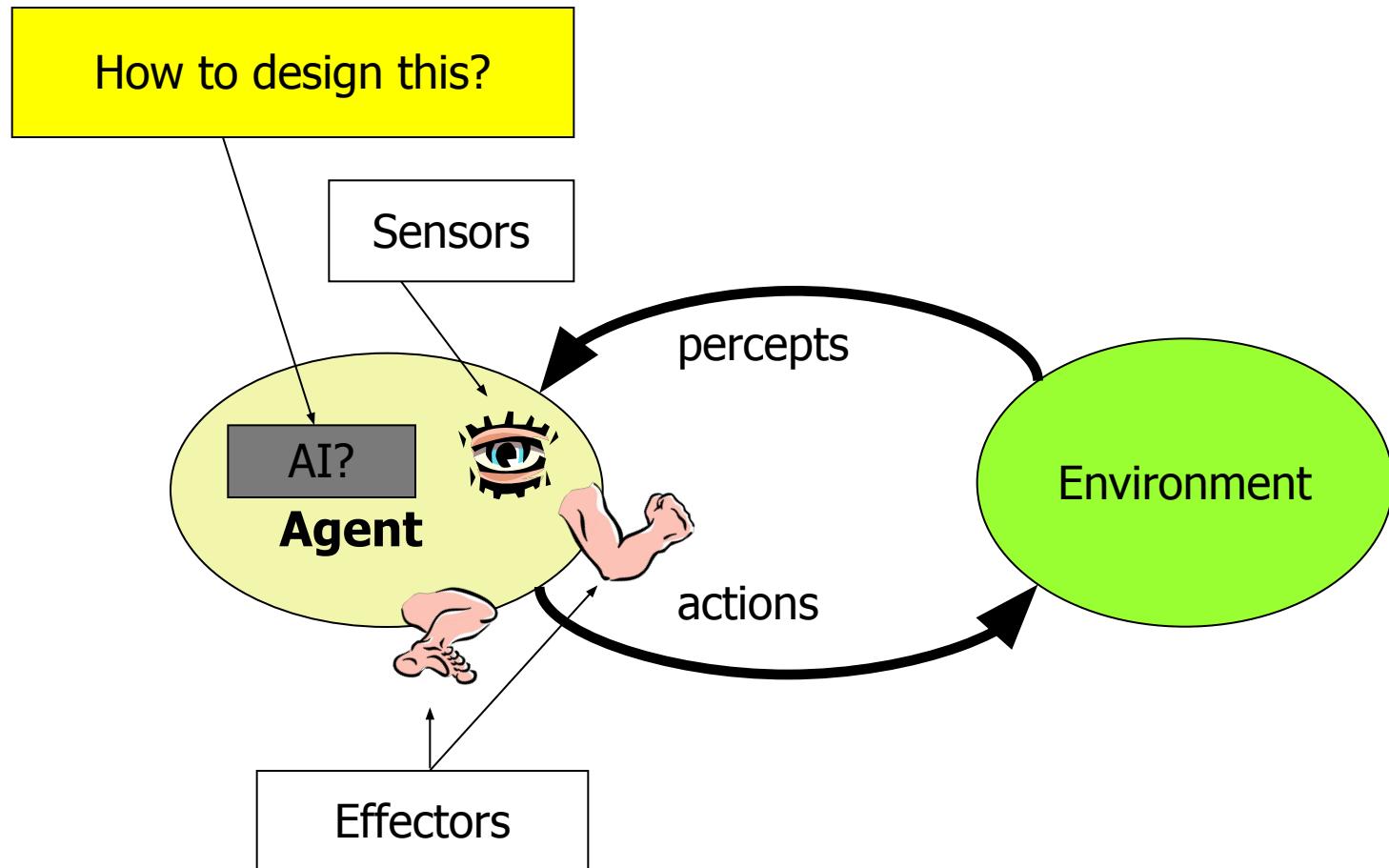
What: The USC Polymorphic Robotics Lab may have some openings for student volunteers/workers

Applications: US Citizenship required.
Please email and talk me if you are interested

Where: USC Information Sciences Institute
Marina del Rey,

Commute by USC Buses

AGENT ENVIRONMENT INTERACTIONS



PLEASE CONSIDER

How to test if a system is “intelligent” or not?

What is a (basic) Turing test?

What is a (full) Total Turing test?

What is rational?

What is the most simple environment?

What is the most complex environment?

How many types of Agents?

What is an agent function? (optional 2 views)

WHAT IS AI?

The exciting new effort to make computers think ... machine with minds, in the full and literal sense"
(Haugeland 1985)

"The study of mental faculties through the use of computational models"
(Charniak et al. 1985)

"The art of creating machines that perform functions that require intelligence when performed by people" (Kurzweil, 1990)

A field of study that seeks to explain and emulate intelligent behavior in terms of computational processes"
(Schalkol, 1990)

How to Measure Intelligence?

Systems that think like humans

Systems that act like humans

Systems that think rationally

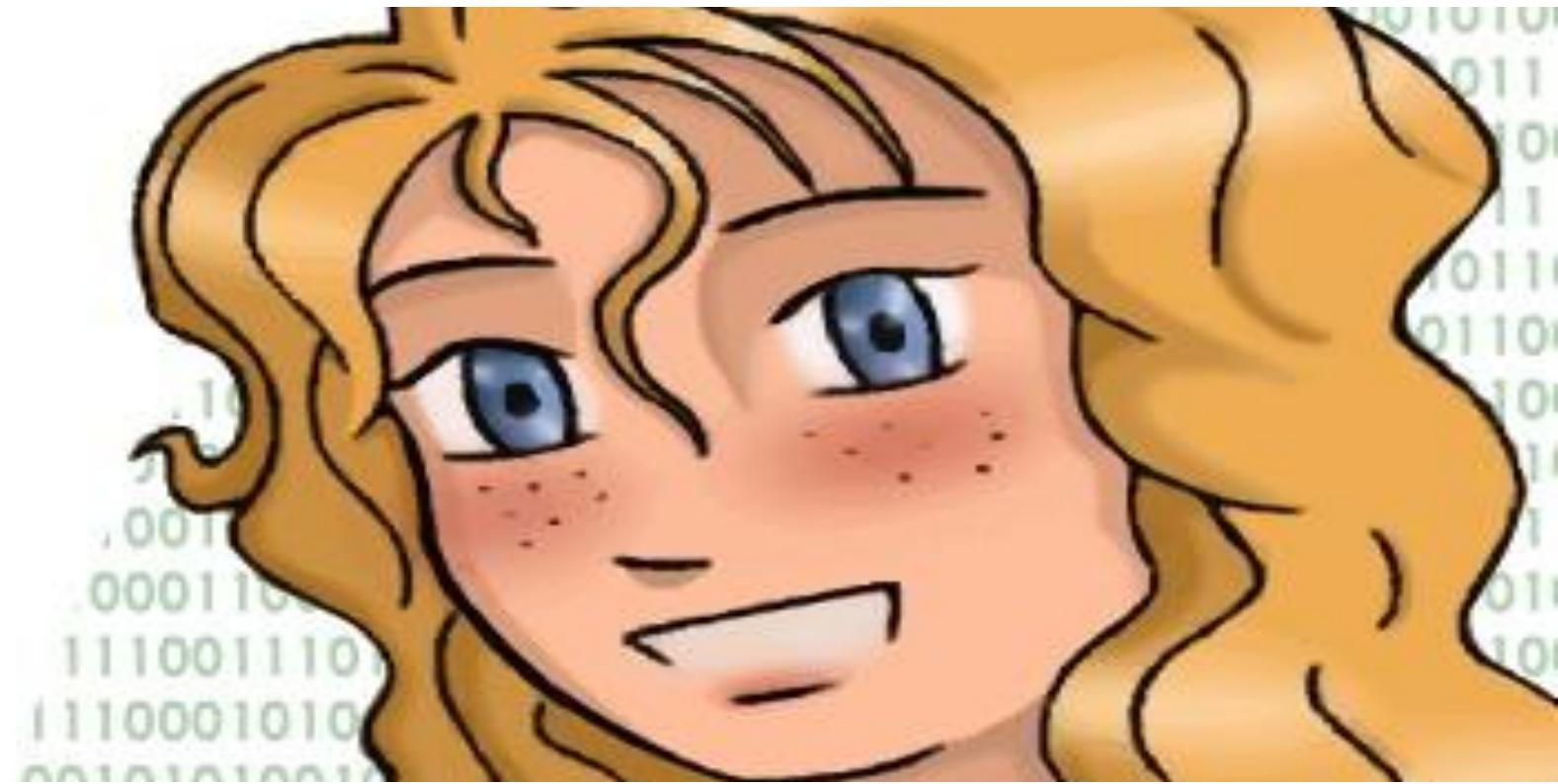
Systems that act rationally

EUGENE GOOSTMAN – TURING TEST 2014

- [Interview](#)
- What do you think? Was the test really passed?
- What effect did context, that is, saying that the computer was playing a 13-year-old Ukrainian boy for whom English was a second language, have on the results?
- What does that tell us about the Turing test as a test of artificial intelligence?
- What would be a better test?
- <http://www.princetonai.com/bot/>



IMITATION CHAMPION MITSUKU – 2013 LOEBENER WINNER



<http://www.loebner.net/Prizef/loebner-prize.htm>

Did “She” Passed Turning Tests?

‘Ex Machina’



MAJOR ISSUES FOR AI APPLICATIONS

- How to represent knowledge about the world?
- How to react to new perceived events?
- How to integrate new percepts to past experience?
- How to understand the user?
- How to optimize balance between user goals & environment constraints?
- How to use reasoning to decide on the best course of action?
- How to communicate back with the user?
- How to plan ahead?
- How to learn from experience?

IS AI SCIENCE OR ENGINEERING?

A **science** is a field of study that leads to the acquisition of empirical knowledge by the scientific method, which involves falsifiable hypotheses about what is.

A pure **engineering** field can be thought of as taking a fixed base of empirical knowledge and using it to solve problems of interest to society.

What are examples of AI systems that support your answer to this question?

ALPHA ZERO



WHAT YOU SHOULD KNOW

- **What is AI? Why study AI?**
- **What is a performance measure? Rational action? Why are they important for an agent?**
- **What is the relationship between the agent and task environment? How does the environment affect the agent design?**
- **Why is the boundary between agent and environment important?**
- **How is this demonstrated by the recent AI successes?**
- **What is the Turing Test? How has it shaped the field of AI?**

WANT MORE?

Chapter 1 Problem# 1.14

Chapter 2 Problem# 2.2, 2.5

Chapter 3 Problem# 3.9, 3.14

CSCI 561 - Foundation for Artificial Intelligence

03 Search Algorithms

Uninformed and Informed

Professor Wei-Min Shen
University of Southern California

Outline

Search Algorithms

- **Uninformed Search (this lecture)**

Search strategies: breadth-first, uniform-cost, depth-first, bi-directional, ...

- **Informed Search (next lecture)**

Search strategies: best-first, A*

Heuristic functions

TYPES OF SEARCH ALGORITHMS

Concept Review

How to define a “problem”?

1. ?
2. ?
3. ?
4. ?

What is a “solution”?

What is a “problem space”?

What is a “search tree”?

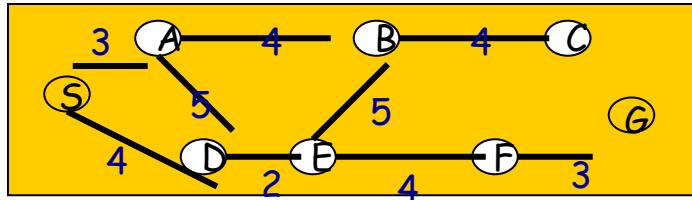
How to find a solution?

Where is your **frontier** in your search?

How to manage your “soldiers” during the search?

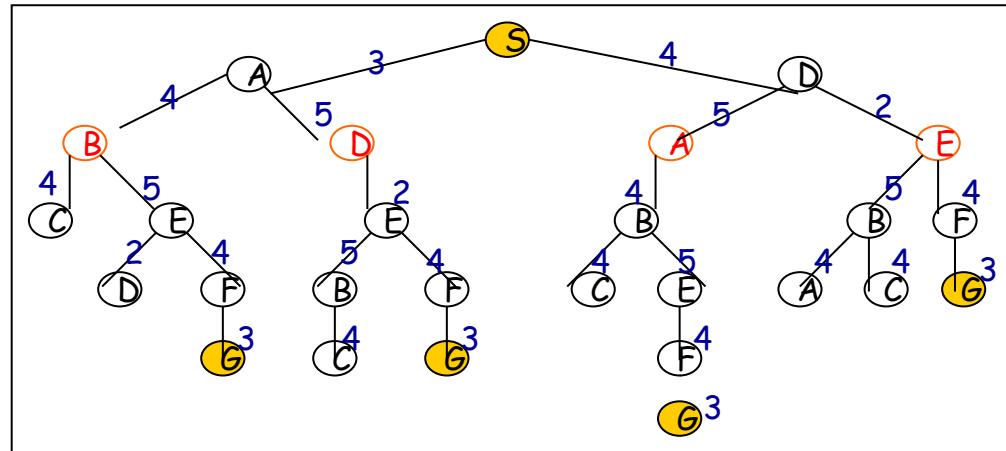
From Problem Space to Search Tree

Problem Space



Associated
loop-free
search tree

frontier



Remember: Implementation of search algorithms

```
Function General-Search(problem, Queuing-Fn) returns a solution, or failure
  nodes <- make-queue(make-node(initial-state[problem])) // the frontier
  loop do
    if nodes is empty then return failure
    node <- Remove-Front(nodes) // command your soldiers
    if Goal-Test[problem] applied to State(node) succeeds then return node
    nodes <- Queuing-Fn(nodes, Expand(node, Operators[problem])) // organize your soldiers
  end
```

Organizing your soldiers at the frontier:

Queuing-Fn(*queue, elements*) is a queuing function that inserts a set of elements into the queue and determines the order of node expansion. Varieties of the queuing function produce varieties of the search algorithm.

Remember: Implementation of search algorithms

```
Function General-Search(problem, Queuing-Fn) returns a solution, or failure
  nodes <- make-queue(make-node(initial-state[problem])) // the frontier
  loop do
    if nodes is empty then return failure
    node <- Remove-Front(nodes)           // command your soldiers
    if Goal-Test[problem] applied to State(node) succeeds then return node
    nodes <- Queuing-Fn(nodes, Expand(node, Operators[problem])) // organize your soldiers
  end
```

Organizing your soldiers at the frontier:

Breadth-first search: Enqueue expanded (children) nodes to the **back** of the queue (FIFO order)

Depth-first search: Enqueue expanded (children) nodes to the **front** of the queue (LIFO order)

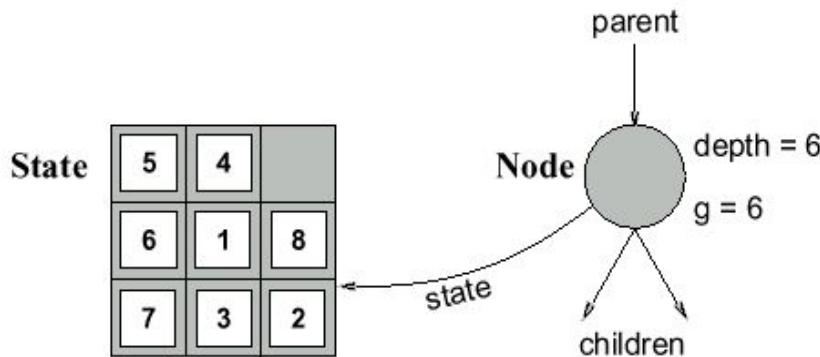
Uniform cost search: Enqueue expanded (children) nodes so that queue is **ordered by the path (past) cost of the nodes** (priority queue order).

Encapsulating *state* information in *nodes*

A *state* is a (representation of) a physical configuration

A *node* is a data structure constituting part of a search tree
includes *parent*, *children*, *depth*, *path cost* $g(x)$

States do not have parents, children, depth, or path cost!



The EXPAND function creates new nodes, filling in the various fields and using the OPERATORS (or SUCCESSORFn) of the problem to create the corresponding states.

How to Compare Different Search Strategies?

- A search strategy is defined by picking the order of node expansion.
- Search algorithms are commonly evaluated according to the following four criteria:
 - **Completeness:** does it always find a solution if one exists?
 - **Time complexity:** how long does it take as function of num. of nodes?
 - **Space complexity:** how much memory does it require?
 - **Optimality:** does it guarantee the least-cost solution?
- Time and space complexity are measured in terms of:
 - b – max branching factor of the search tree
 - d – depth of the least-cost solution
 - m – max depth of the search tree (may be infinity)

A Note for Approximations

- In our complexity analysis, we do not take the built-in loop-detection into account.
- The results only 'formally' apply to the variants of our algorithms **WITHOUT** loop-checks.
- Studying the effect of the loop-checking on the complexity is hard:
 - overhead of the checking MAY or MAY NOT be compensated by the reduction of the size of the tree.
- Also: our analysis **DOES NOT** take the length (space) of representing paths into account !!

<http://www.cs.kuleuven.ac.be/~dannyd/FAI/>

Uninformed search strategies

Use only information available in the problem formulation

- Breadth-first
- Uniform-cost
- Depth-first
- Depth-limited
- Iterative deepening

BREADTH-FIRST SEARCH

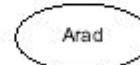
Breadth-first search

Expand shallowest unexpanded node

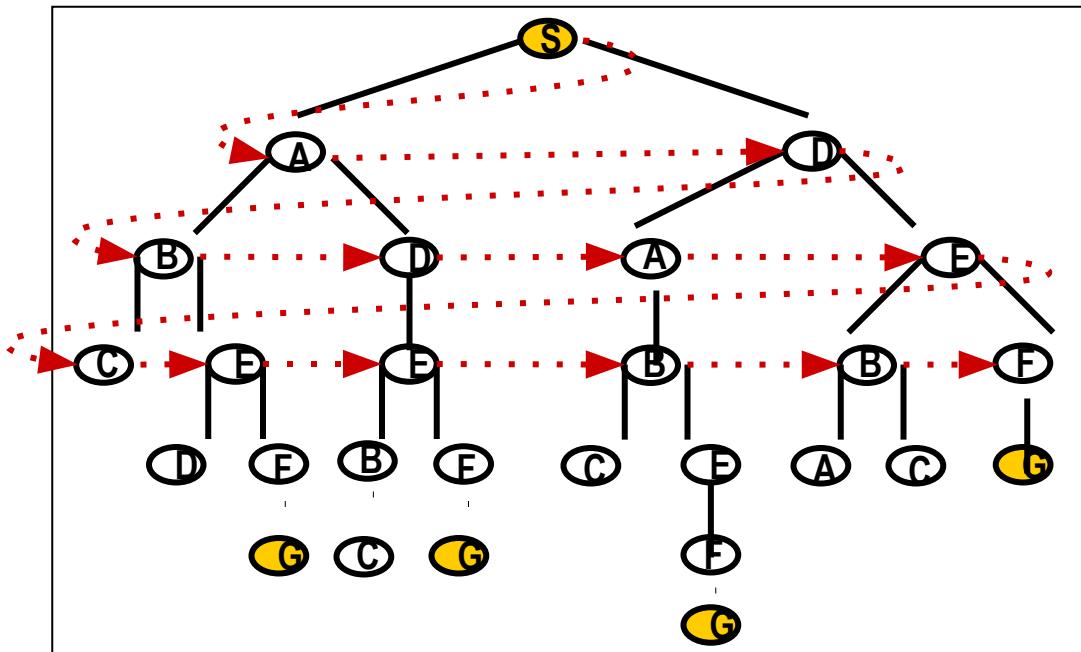
“First-in, first-out”

Implementation:

QUEUEINGFN = put successors at end of queue

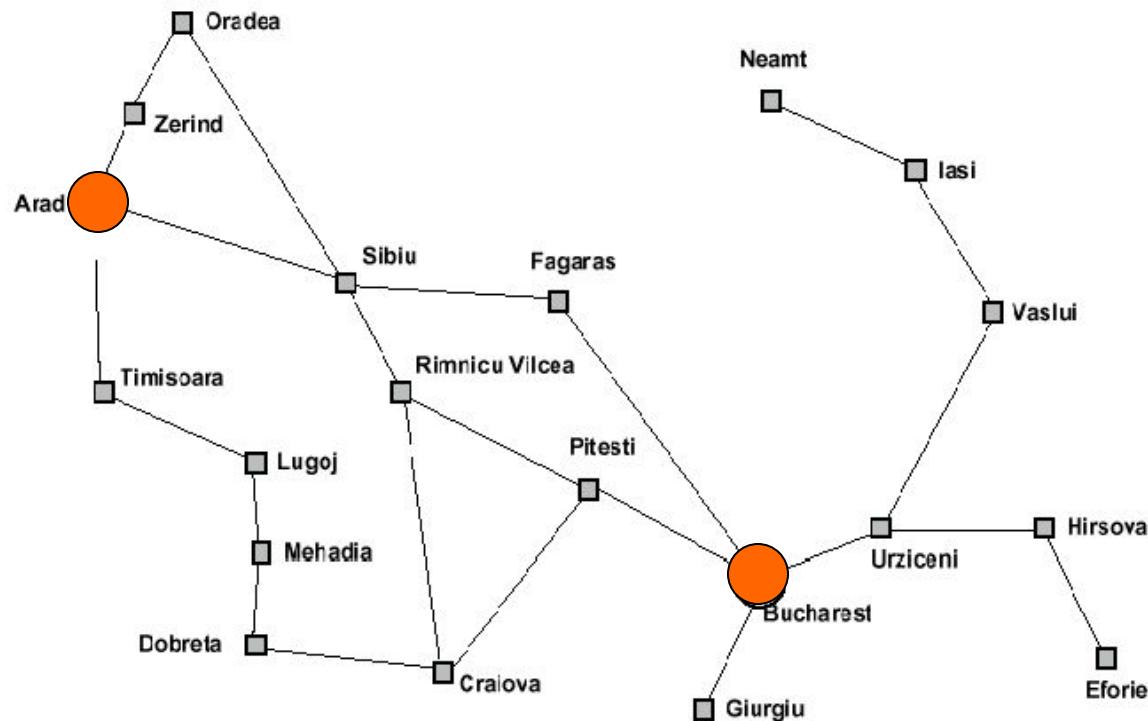


Breadth-first search

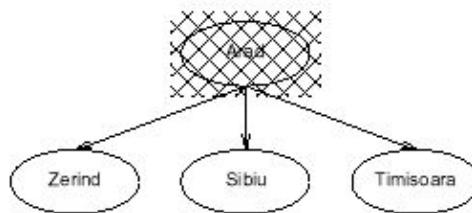


Move downwards,
level by level,
until goal is
reached.

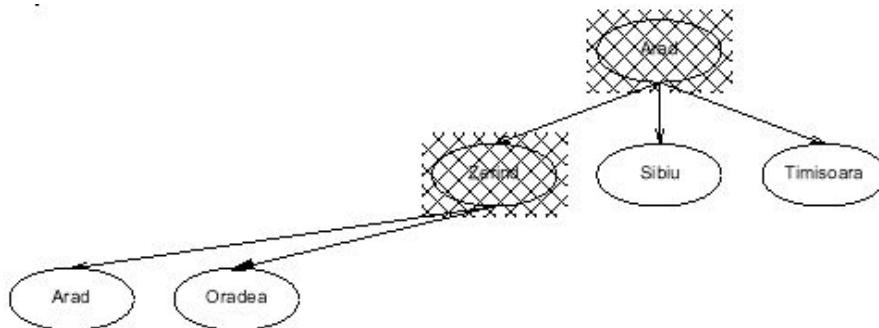
Example: Traveling from Arad To Bucharest



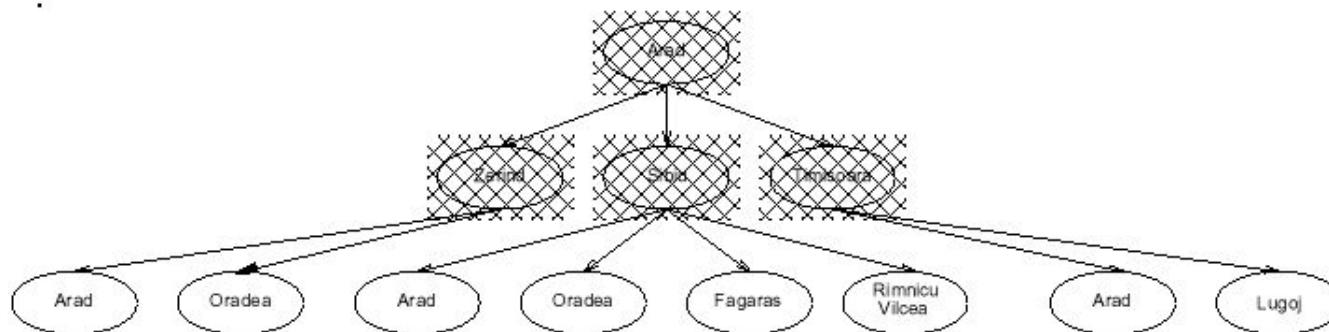
Breadth-first search



Breadth-first search



Breadth-first search



Properties of breadth-first search

- Completeness:
- Time complexity:
- Space complexity:
- Optimality:

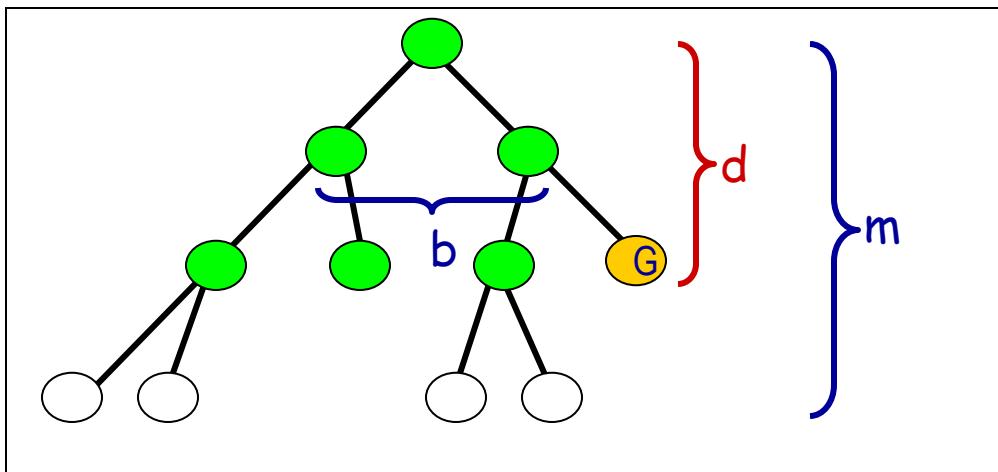
- Search algorithms are commonly evaluated according to the following four criteria:
 - **Completeness:** does it always find a solution if one exists?
 - **Time complexity:** how long does it take as function of num. of nodes?
 - **Space complexity:** how much memory does it require?
 - **Optimality:** does it guarantee the least-cost solution?
- Time and space complexity are measured in terms of:
 - b – max branching factor of the search tree
 - d – depth of the least-cost solution
 - m – max depth of the search tree (may be infinity)

Properties of breadth-first search

- Completeness: Yes, if b is finite
- Time complexity: $1+b+b^2+\dots+b^d = O(b^d)$, i.e., exponential in d
- Space complexity: $O(b^d)$ (see following slides)
- Optimality: Yes (assuming cost = 1 per step)

Time complexity of breadth-first search

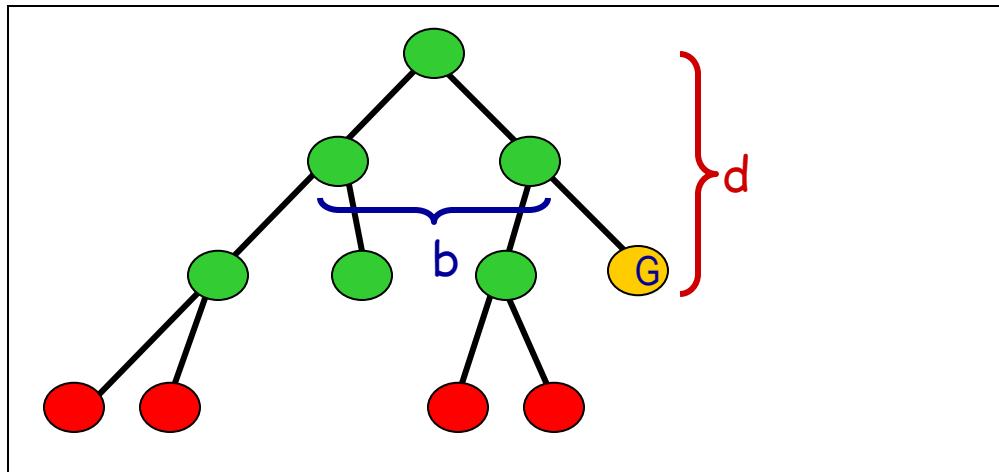
- If a goal node is found on depth d of the tree, all nodes up till that depth are created and examined (note: and the children of nodes at depth d are created and enqueued, but not yet examined).



- Thus: $O(b^d)$

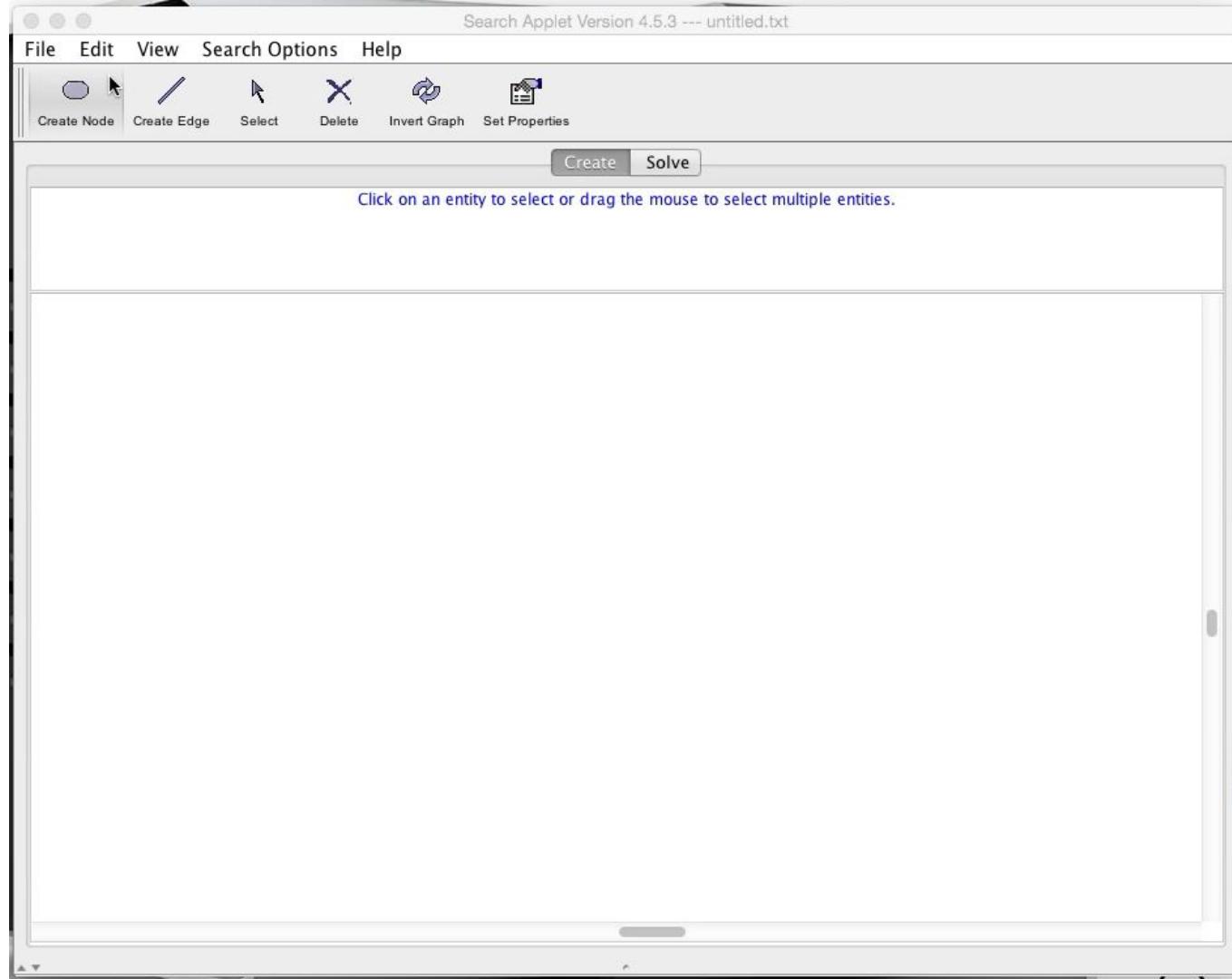
Space complexity of breadth-first

- Largest number of nodes in QUEUE is reached on the level $d+1$ just beyond the goal node.

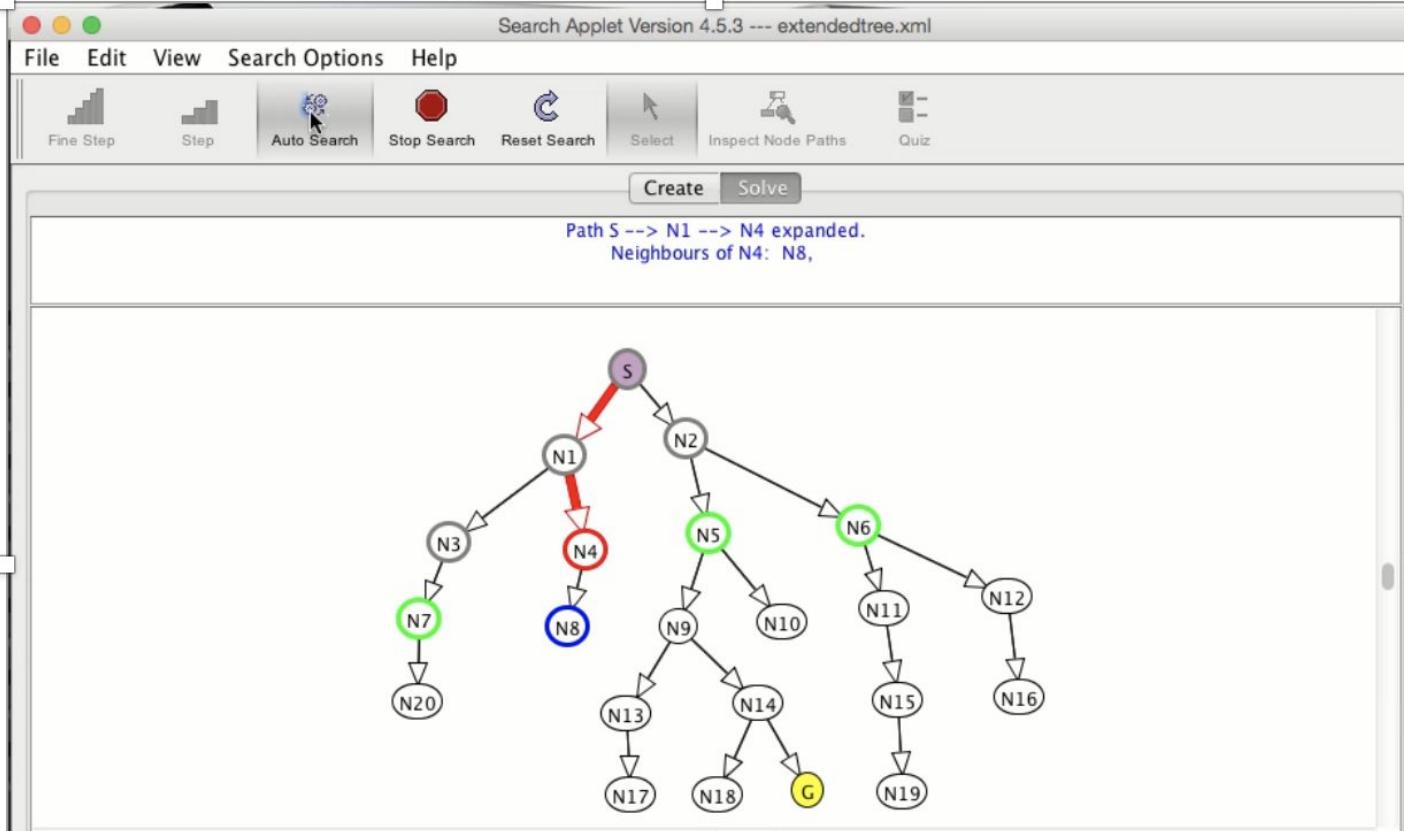


- QUEUE contains all nodes. (Thus: 4) .
- In General: $b^{d+1} - b \sim b^d$

Demo



Demo



Algorithm Selected: Breadth First

CURRENT PATH:

S --> N1 --> N4

NEW FRONTIER:

Node: N5 Path Cost: 19.7

Node: N6 Path Cost: 23.4

Node: N7 Path Cost: 38.4

Path: S --> N2 --> N5

Path: S --> N2 --> N6

Path: S --> N1 --> N3 --> N7

UNIFORM-COST SEARCH

Uniform-cost search

Expand least-cost unexpanded node

Implementation:

QUEUEINGFN = insert in order of increasing path cost

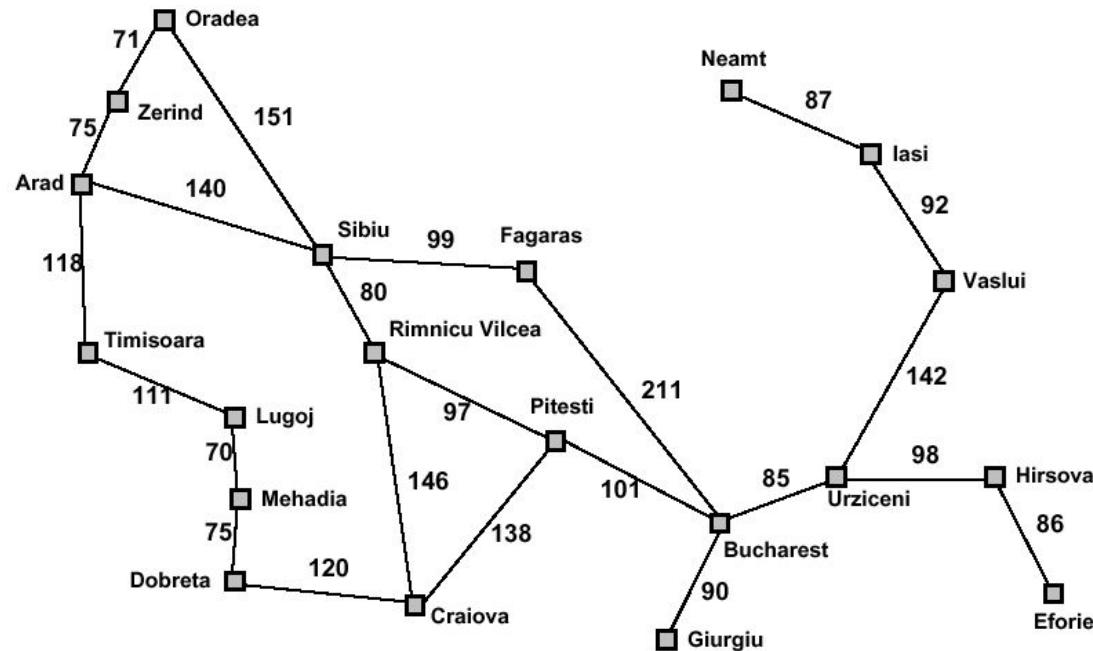


So, the queueing function keeps the node list sorted by increasing path cost, and we expand the first unexpanded node (hence with smallest path cost)

A refinement of the breadth-first strategy:

Breadth-first = uniform-cost with path cost = node depth

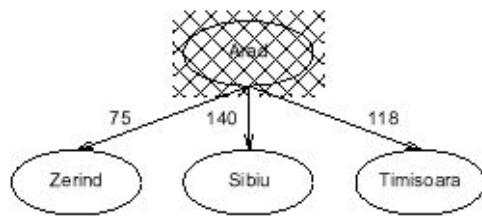
Romania with step costs in km



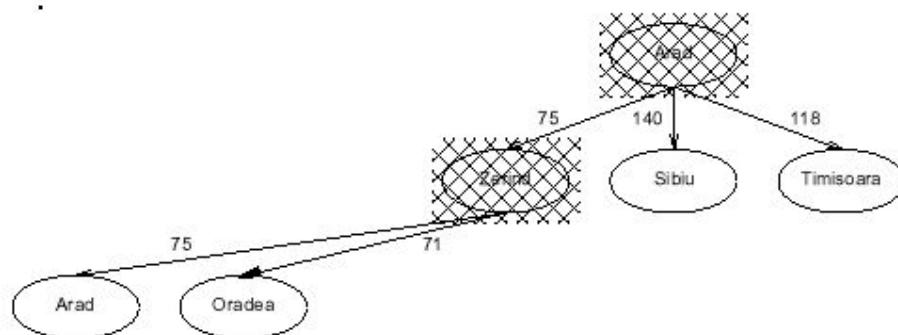
Straight-line distance
to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

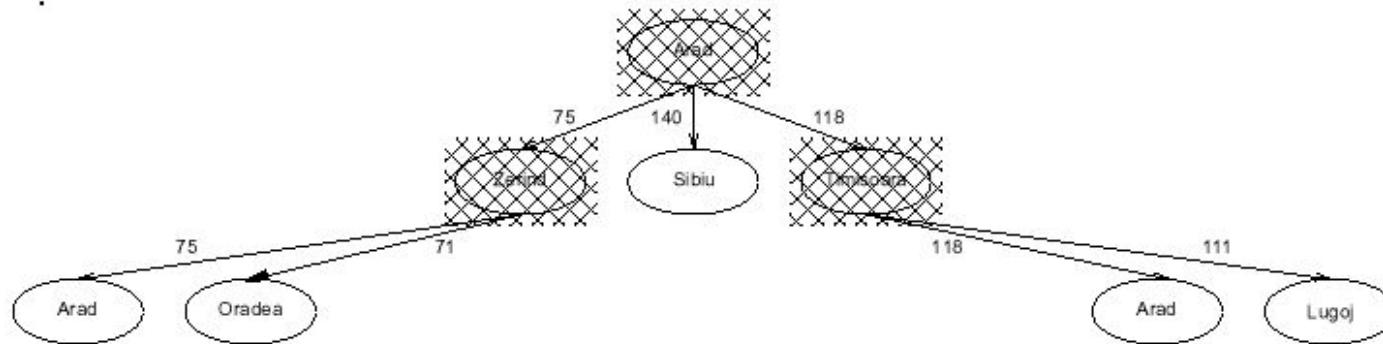
Uniform-cost search



Uniform-cost search



Uniform-cost search



Properties of uniform-cost search

- Completeness: Yes, if step cost $\geq \varepsilon > 0$
- Time complexity: # nodes with $g \leq$ cost of optimal solution, $\leq O(b^d)$
- Space complexity: # nodes with $g \leq$ cost of optimal solution, $\leq O(b^d)$
- Optimality: Yes, as long as path cost never decreases

$g(n)$ is the path cost to node n

Remember:

b = branching factor

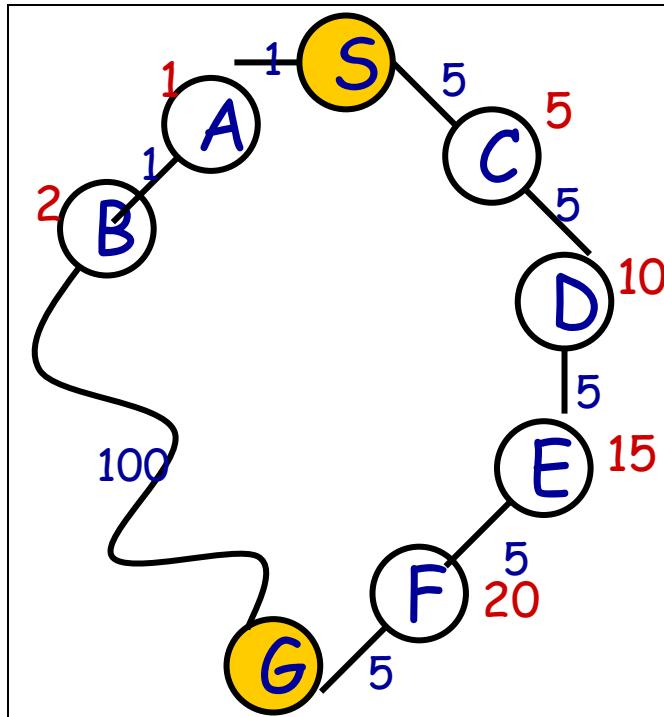
d = depth of least-cost solution

Implementation of uniform-cost search

- Initialize Queue with root node (built from start state)
- Repeat until (Queue empty) or (first node has Goal state):
 - Remove first node from front of Queue
 - Expand node (find its children)
 - Reject those children that have already been considered, to avoid loops
 - Add remaining children to Queue, *in a way that keeps entire queue sorted by increasing path cost*
- If Goal was reached, return success, otherwise failure

Caution! Don't Terminate the Search Pre-maturely

- Uniform-cost search **would not be optimal** if it is terminated when **any** node in the queue has goal state. **But is optimal** if you do goal-check when a node pops up from the frontier.



- Uniform cost returns the path with cost 102 (if any goal node is considered a solution), while there is a path with cost 25.

Note: Loop Detection

- In class, we saw that the search may fail or be sub-optimal if:
 - no loop detection: then algorithm runs into infinite cycles
(A -> B -> A -> B -> ...)
 - not queuing-up a node that has a state which we have already visited: may yield suboptimal solution
 - simply avoiding to go back to our parent: looks promising, but we have not proven that it works

Solution? do not enqueue a node if its state matches the state of any of its parents (assuming path costs > 0).

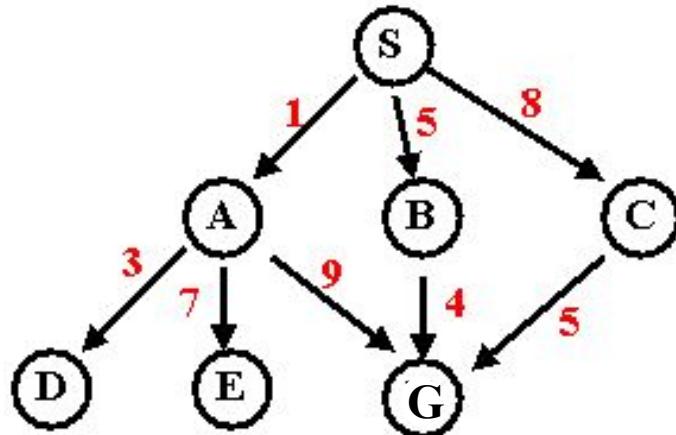
Indeed, if path costs > 0, it will always cost us more to consider a node with that state again than it had already cost us the first time.

Is that enough??

Example

From: <http://www.csee.umbc.edu/471/current/notes/uninformed-search/>

Example Illustrating Uninformed Search Strategies



Breadth-First Search Solution

From: <http://www.csee.umbc.edu/471/current/notes/uninformed-search/>

Breadth-First Search

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

exp. node nodes list

	(S)
S	(A B C)
A	(B C D E G)
B	(C D E G G')
C	(D E G G' G")
D	(E G G' G")
E	(G G' G")
G	(G' G")

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

Number of nodes expanded (including goal node) = 7

Uniform-Cost Search Solution

From: <http://www.csee.umbc.edu/471/current/notes/uninformed-search/>

Uniform-Cost Search

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

exp. node nodes list

	{ S }
S	(A(1) B(5) C(8))
A	(D(4) B(5) C(8) E(8) G(10))
D	(B(5) C(8) E(8) G(10))
B	(C(8) E(8) G(9) G(10))
C	(E(8) G(9) G(10) G(13))
E	(G(9) G(10) G(13))
G	()

(NB, we don't return G)

Solution path found is S B G <-- this G has cost 9, not 10
Number of nodes expanded (including goal node) = 7

Note: Queueing in Uniform-Cost Search

In the previous example, it is wasteful (but not incorrect) to queue-up three nodes with G state, if our goal is to find the least-cost solution:

Although they represent different paths, we know for sure that the one with smallest path cost (9 in the example) will yield a solution with smaller total path cost than the others.

So we can refine the queueing function by:

- queue-up node if

1) its state does not match the state of any parent

// it is new

and

2) path cost smaller than path cost of any

// it is better

unexpanded node with same state in the queue

(and

in this case, replace old node with same

state by our new node)

Is that it??

A Clean Robust Algorithm

```

Function UniformCost-Search(problem, Queuing-Fn) returns a solution, or failure
  open □ make-queue(make-node(initial-state[problem])) // frontier
  closed □ [empty] // already visited

loop do
  if open is empty then return failure
  currnode □ Remove-Front(open)
  if Goal-Test[problem] applied to State(currnode) then return currnode
  children □ Expand(currnode, Operators[problem])
  while children not empty

    [... see next slide ...]

end
  closed □ Insert(closed, currnode) // already visited
  open □ Sort-By-PathCost(open) // frontier
end

```

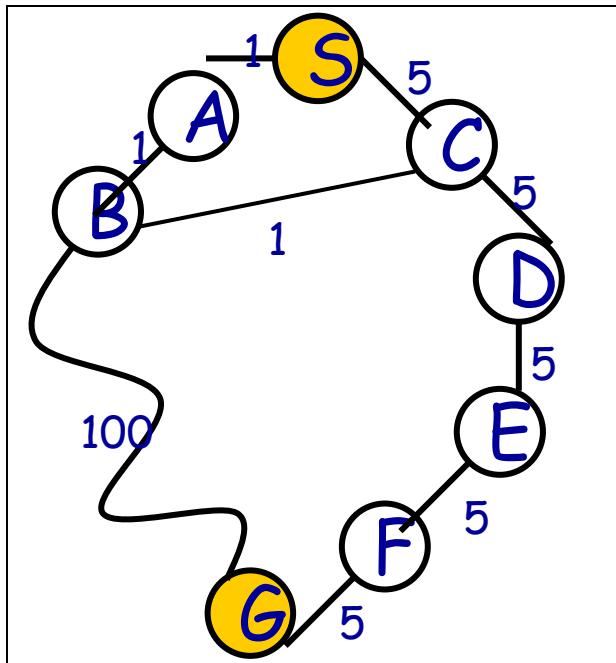
A Clean Robust Algorithm

[... see previous slide ...]

```
children □ Expand(currnode, Operators[problem])
while children not empty
    child □ Remove-Front(children)
    if no node in open or closed has child's state
        open □ Queuing-Fn(open, child)
    else if there exists node in open that has child's state
        if PathCost(child) < PathCost(node)
            open □ Delete-Node(open, node)
            open □ Queuing-Fn(open, child)
    else if there exists node in closed that has child's state
        if PathCost(child) < PathCost(node)
            closed □ Delete-Node(closed, node)
            open □ Queuing-Fn(open, child)
end
```

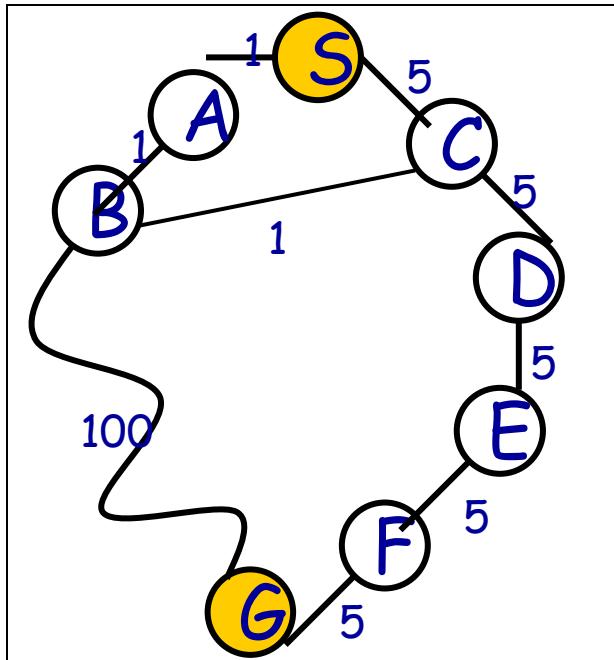
[... see previous slide ...]

Example



#	State	Depth	Cost	Parent
1	S	0	0	-

Example

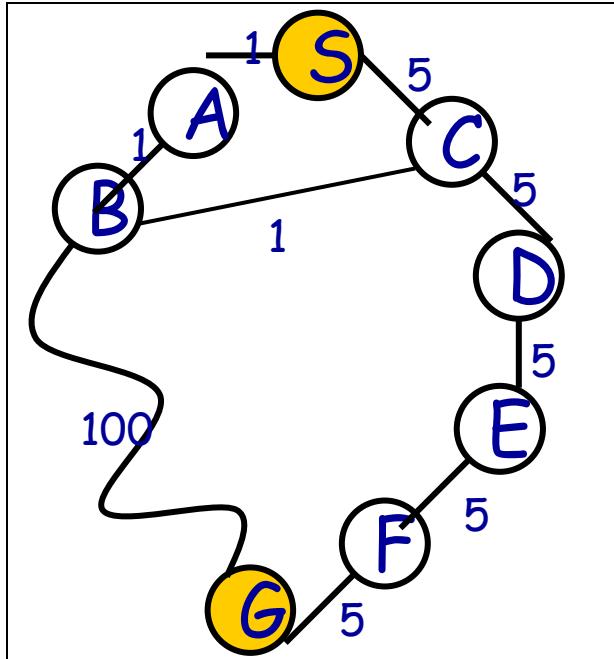


#	State	Depth	Cost	Parent
1	S	0	0	-
2	A	1	1	1
3	C	1	5	1

Black = open queue
Grey = closed queue

Insert expanded nodes
Such as to keep *open* queue sorted

Example

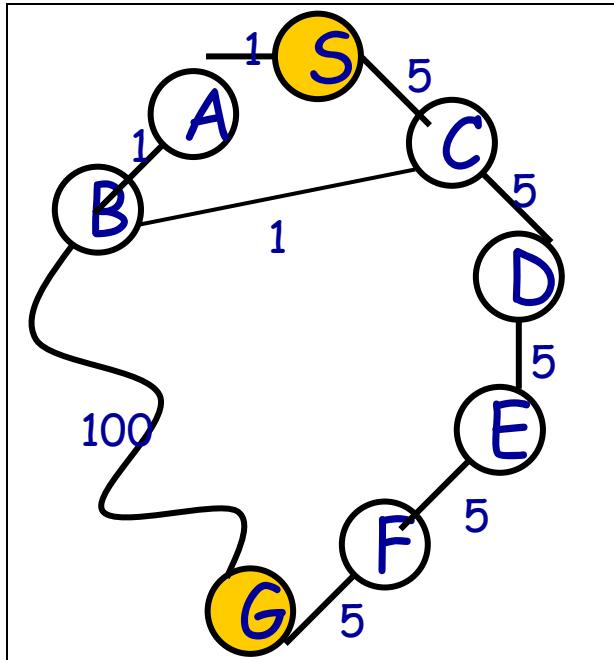


#	State	Depth	Cost	Parent
1	S	0	0	-
2	A	1	1	1
4	B	2	2	2
3	C	1	5	1

Node 2 has 2 successors: one with state B and one with state S.

We have node #1 in *closed* with state S; but its path cost 0 is smaller than the path cost obtained by expanding from A to S. So we do not queue-up the successor of node 2 that has state S.

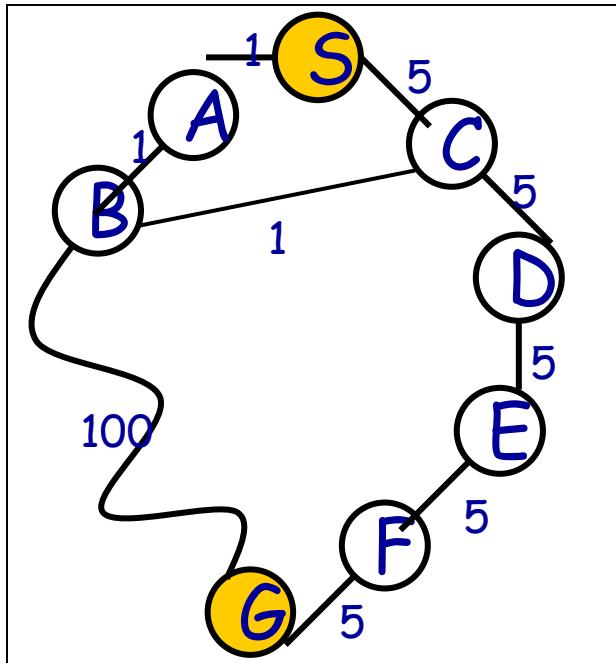
Example



#	State	Depth	Cost	Parent
1	S	0	0	-
2	A	1	1	1
4	B	2	2	2
5	C	3	3	4
6	G	3	102	4

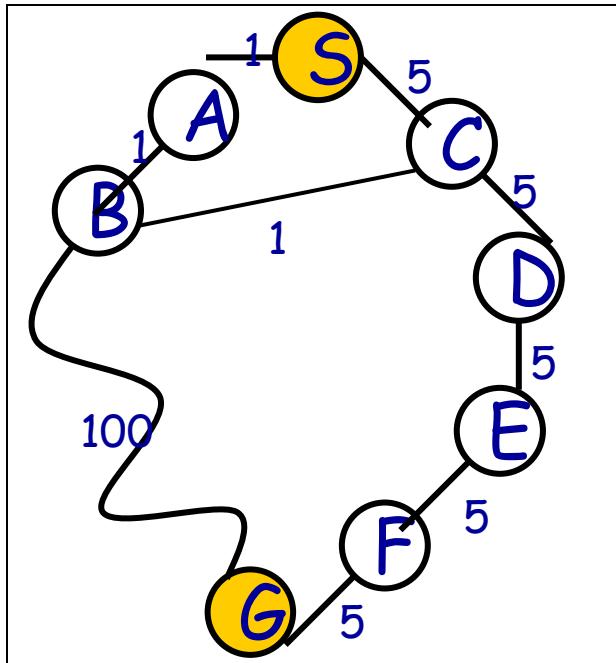
Node 4 has a successor with state C and Cost smaller than node #3 in *open* that Also had state C; so we update *open* To reflect the shortest path.

Example



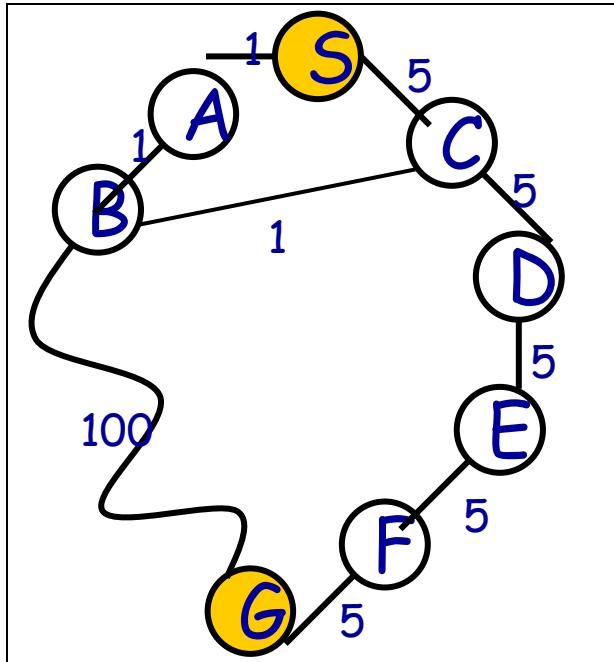
#	State	Depth	Cost	Parent
1	S	0	0	-
2	A	1	1	1
4	B	2	2	2
5	C	3	4	
7	D	4	5	
6	G	3	102	4

Example



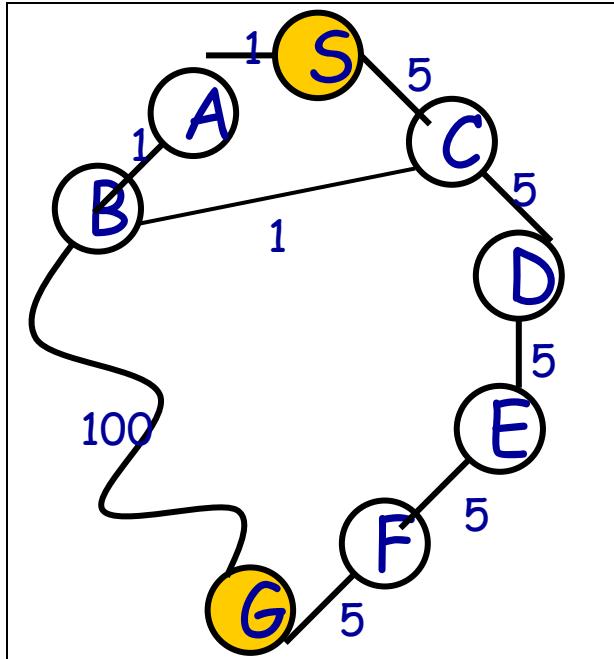
#	State	Depth	Cost	Parent
1	S	0	0	-
2	A	1	1	1
4	B	2	2	2
5	C	3	4	
7	D	4	5	
8	E	5	7	
6	G	3	102	4

Example



#	State	Depth	Cost	Parent
1	S	0	0	-
2	A	1	1	1
4	B	2	2	2
5	C	3	4	
7	D	4	5	
8	E	5	13	7
9	F	6	18	8
6	G	3	102	4

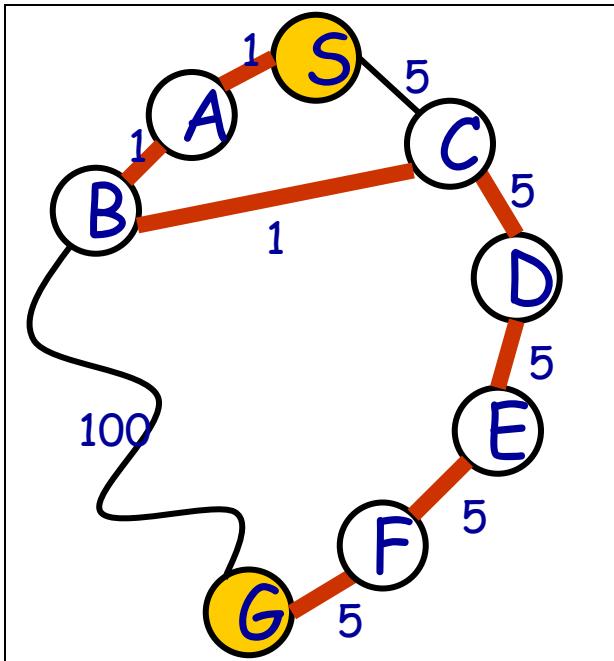
Example



#	State	Depth	Cost	Parent
1	S	0	0	-
2	A	1	1	1
4	B	2	2	2
5	C	3	4	
7	D	4	5	
8	E	5	7	
9	F	6	8	
10	G	7	9	

The node with state G and cost 102 has been removed from the open queue and replaced by cheaper node with state G and code 23 which was pushed into the open queue.

Example



#	State	Depth	Cost	Parent
1	S	0	0	-
2	A	1	1	1
4	B	2	2	2
5	C	3	4	
7	D	4	5	
8	E	5	7	
9	F	6	8	
10	G	7	9	

Goal reached

DEPTH-FIRST SEARCH

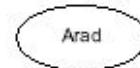
Depth-first search

Expand deepest unexpanded node

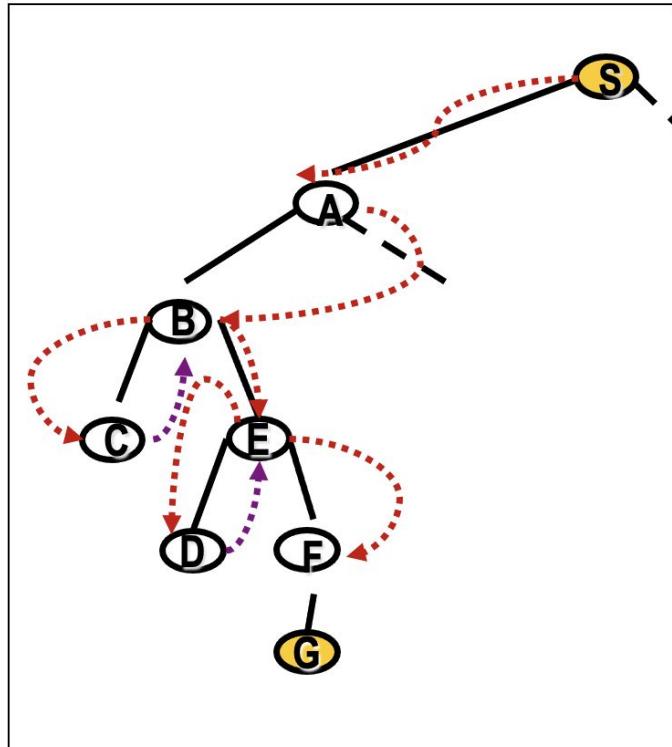
“last-in, first-out”

Implementation:

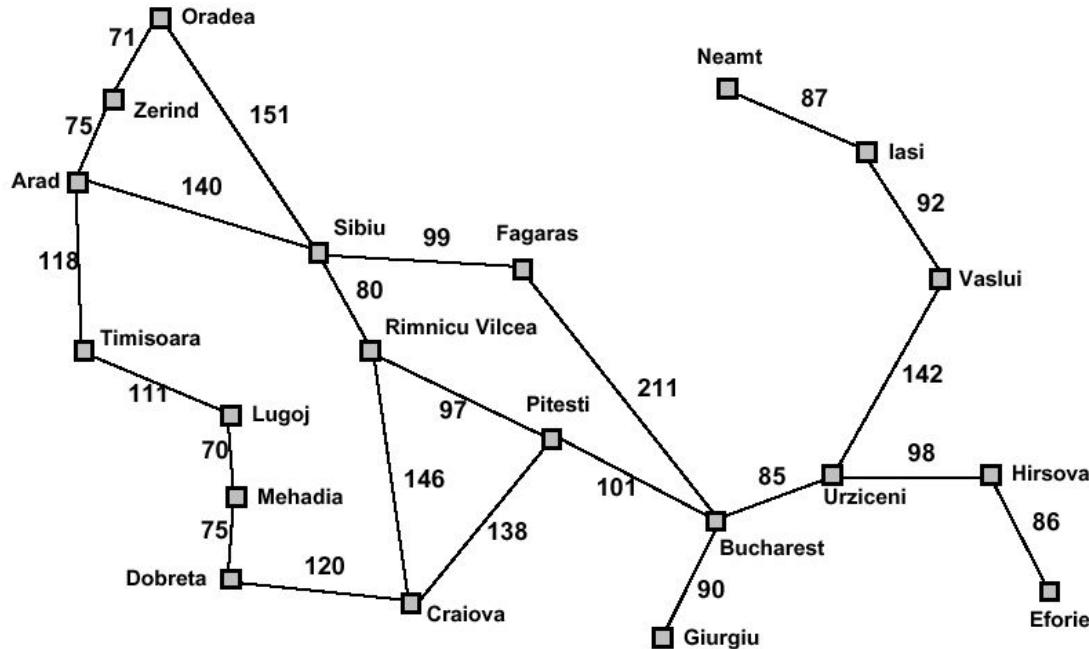
QUEUEINGFN = insert successors at front of queue



Depth First Search



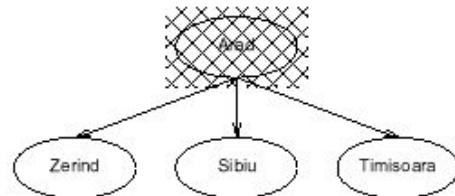
Romania with step costs in km



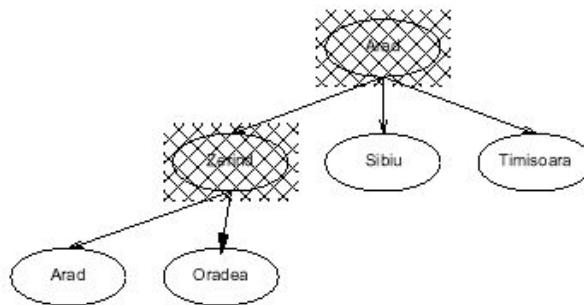
Straight-line distance
to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobrete	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

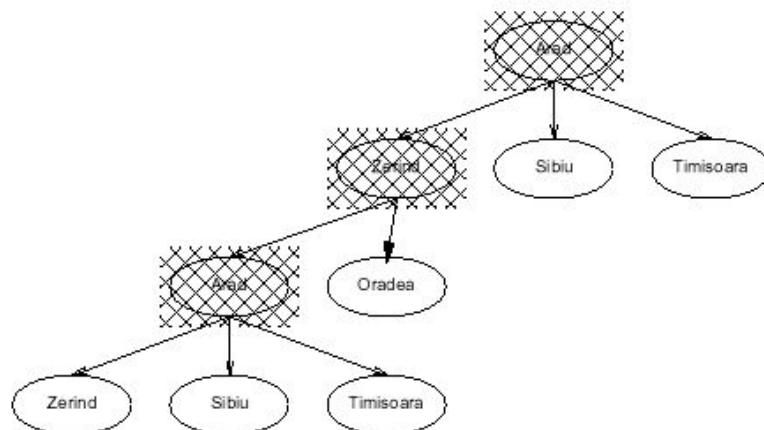
Depth-first search



Depth-first search



Depth-first search



I.e., depth-first search can perform infinite cyclic excursions
Need a finite, non-cyclic search space (or repeated-state checking)

Properties of depth-first search

- Completeness: No, fails in infinite state-space (yes if finite state space)
- Time complexity: $O(b^m)$
- Space complexity: $O(bm)$
- Optimality: No

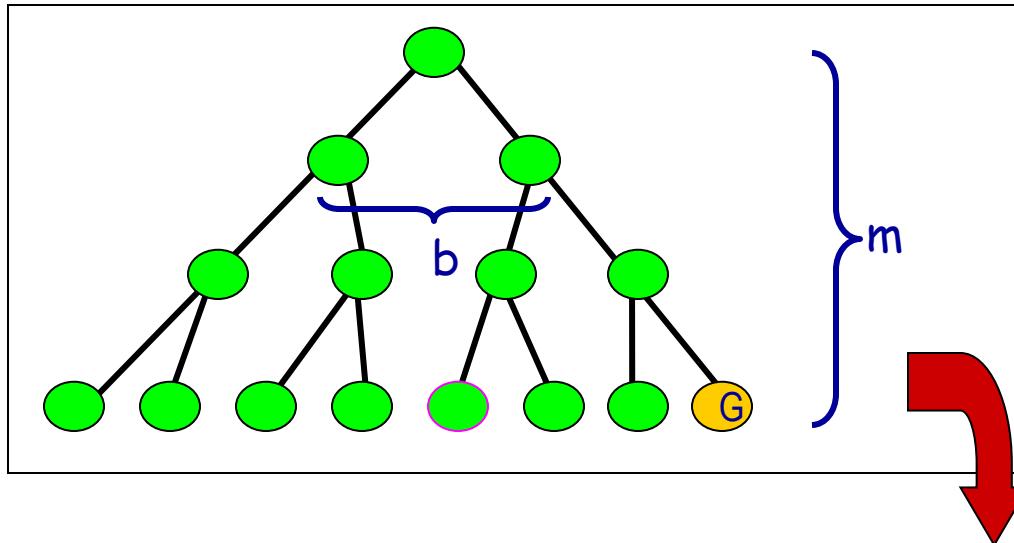
Remember:

b = branching factor

m = max depth of search tree

Time complexity of depth-first: details

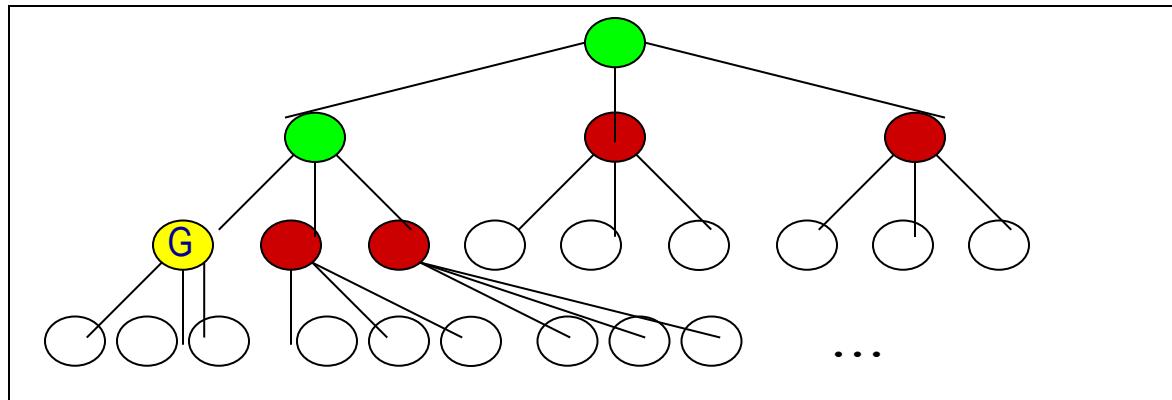
- In the worst case:
 - the (only) goal node may be on the right-most branch,



- Time complexity $= b^m + b^{m-1} + \dots + 1 = \frac{b^{m+1}-1}{b-1}$
- Thus: $O(b^m)$

Space complexity of depth-first

- Largest number of nodes in QUEUE is reached in bottom left-most node.
- Example: $m = 2$, $b = 3$:



- QUEUE contains all nodes. Thus: 4.
- In General: $((b-1) * m)$
- Order: $O(m*b)$

Avoiding repeated states

In increasing order of effectiveness and computational overhead:

- do not return to state we come from, i.e., expand function will skip possible successors that are in same state as node's parent.
- do not create paths with cycles, i.e., expand function will skip possible successors that are in same state as any of node's ancestors.
- do not generate any state that was ever generated before, by keeping track (in memory) of every state generated, unless the cost of reaching that state is lower than last time we reached it.

Depth-limited search

Is a depth-first search with a limited depth l

Implementation: Nodes at depth l have no successors

Complete: if cutoff chosen appropriately then it is guaranteed to find a solution

Optimal: it does not guarantee to find the least-cost solution

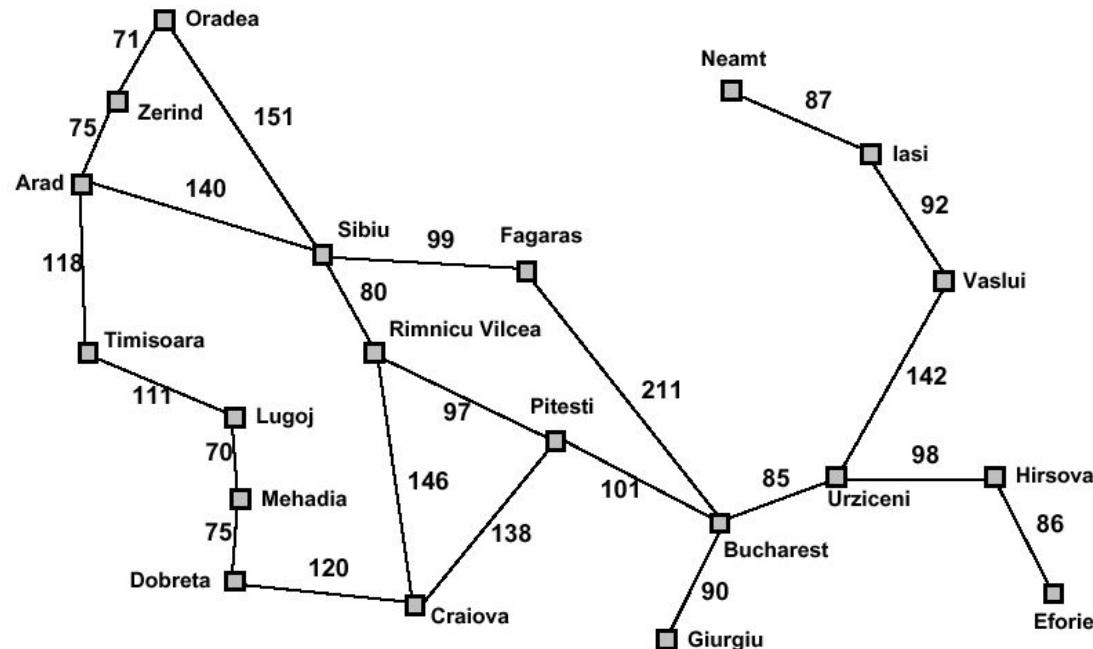
Iterative deepening search

```
Function Iterative-deepening-Search(problem) returns a solution or failure
  for depth = 0 to  $\infty$  do
    result  $\leftarrow$  Depth-Limited-Search(problem, depth)
    if result succeeds then return result
  end
  return failure
```

Combines the best of breadth-first and depth-first search strategies.

- Completeness: Yes,
- Time complexity: $O(b^d)$
- Space complexity: $O(bd)$
- Optimality: Yes, if step cost = 1

Romania with step costs in km

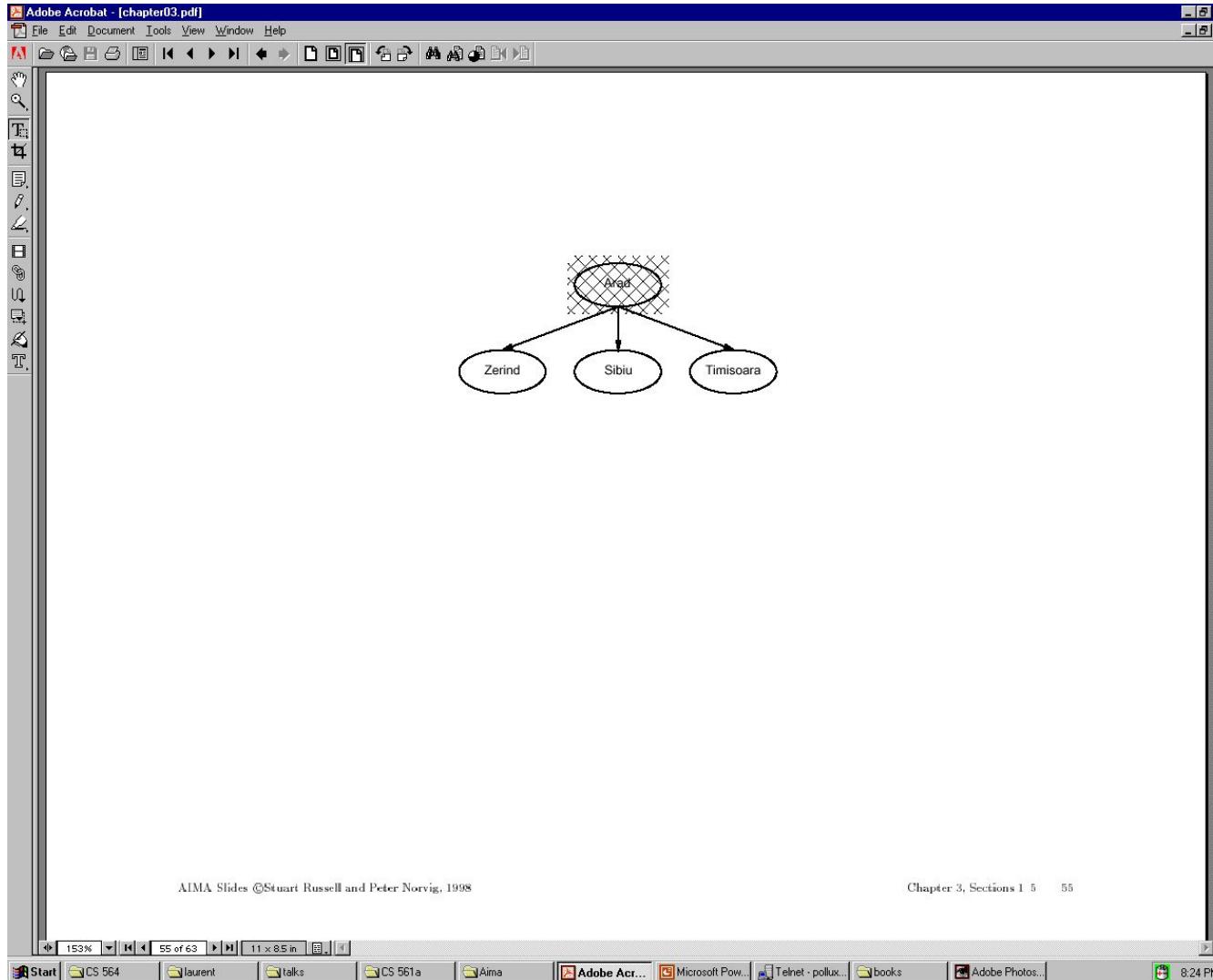


Straight-line distance
to Bucharest

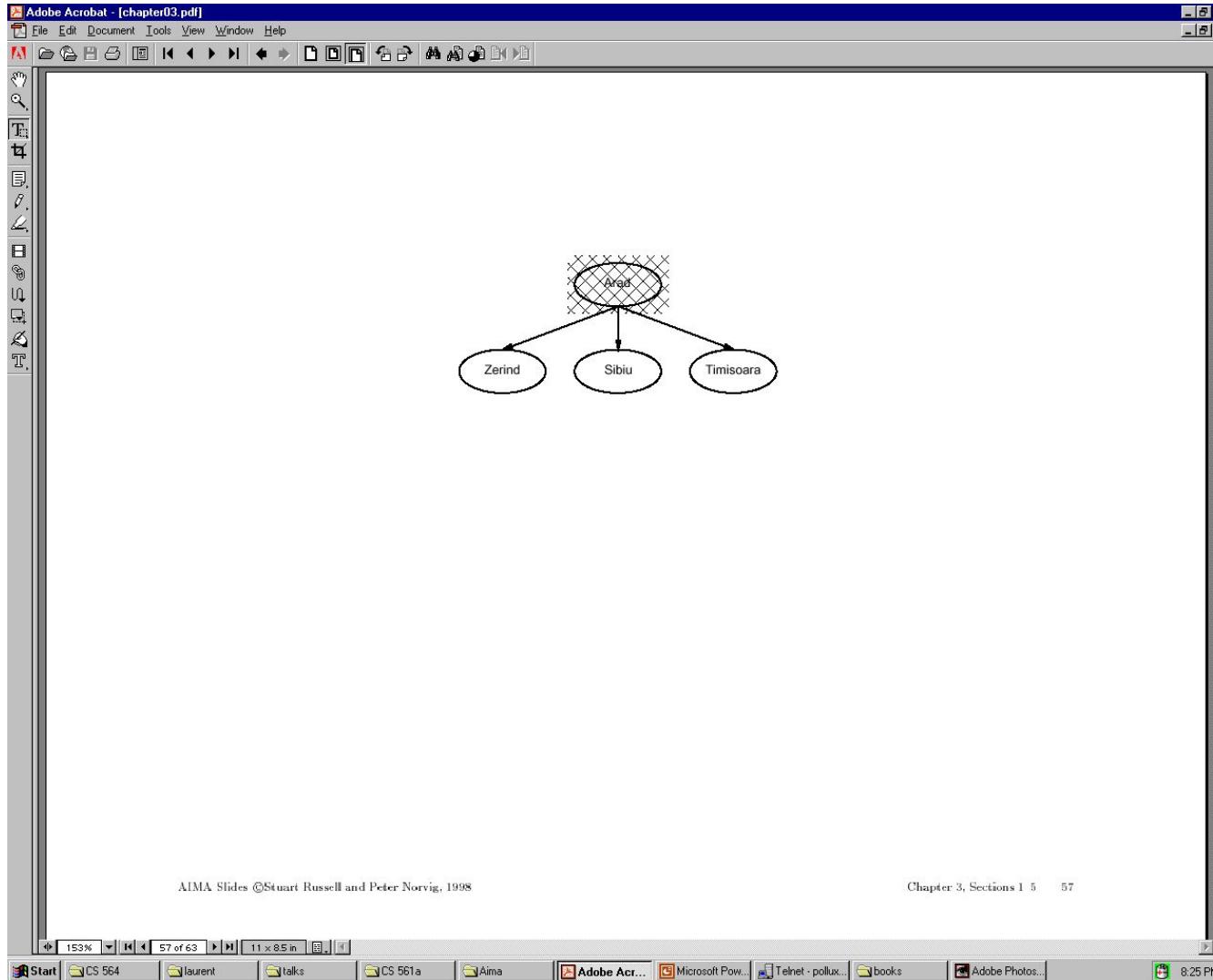
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

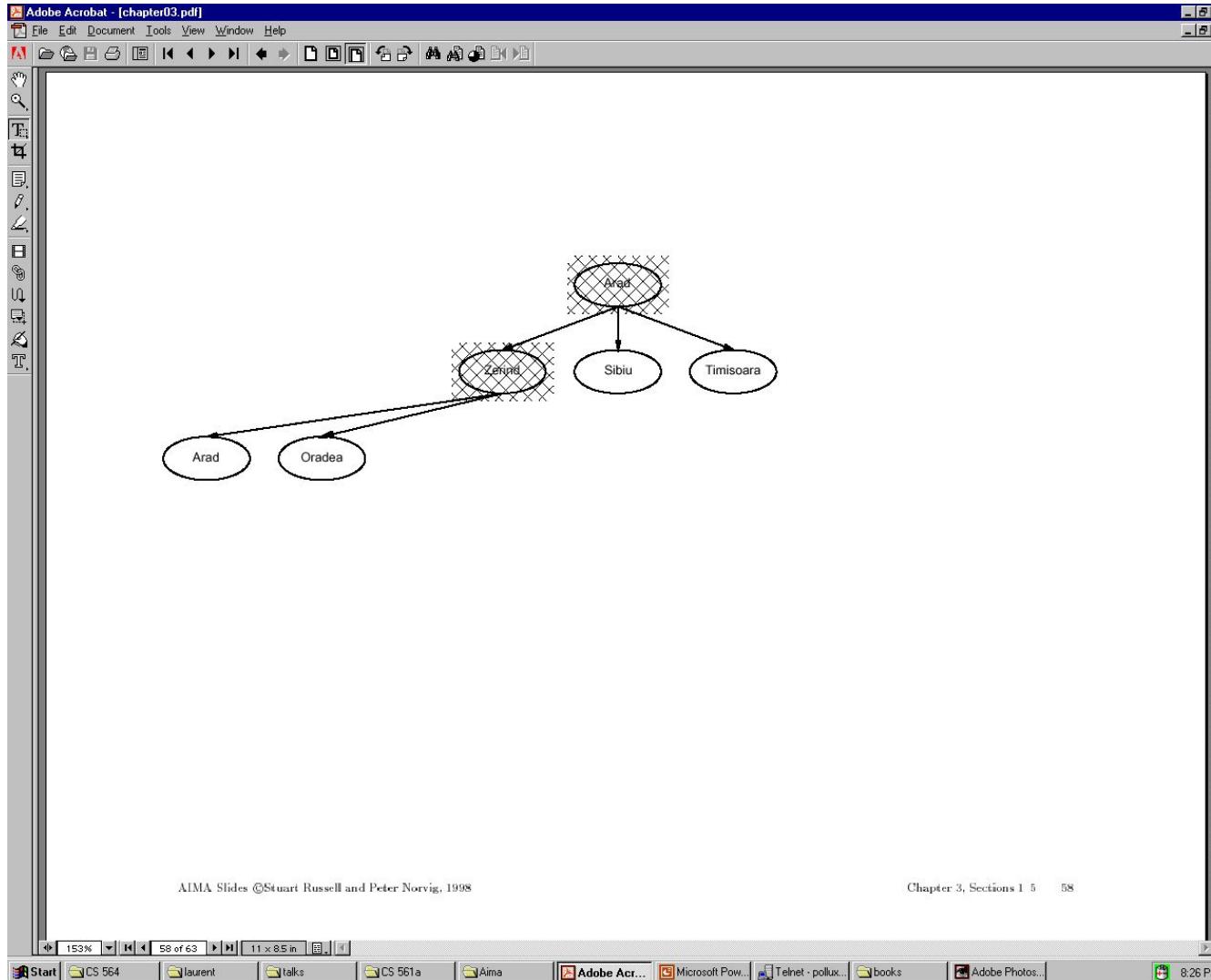


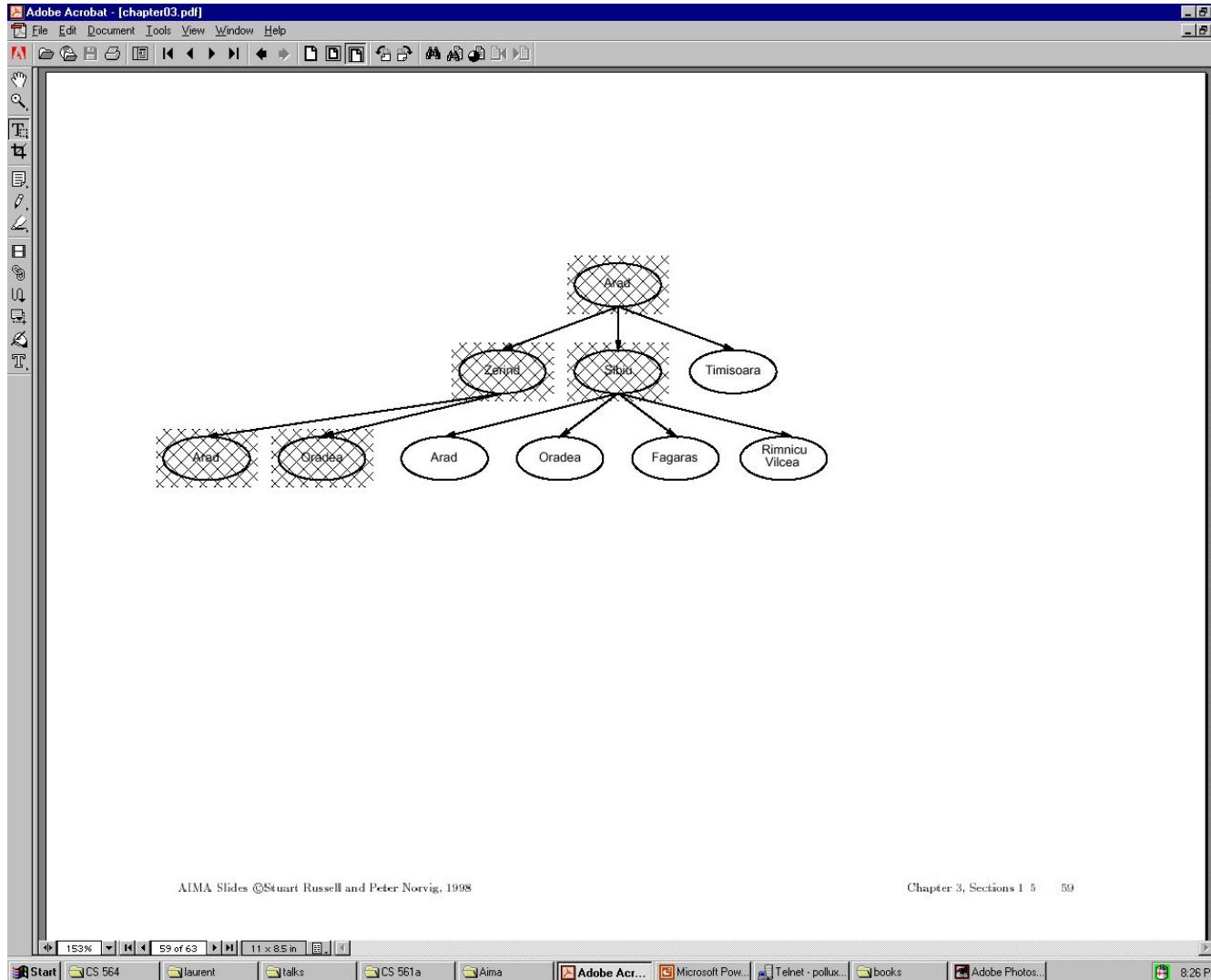


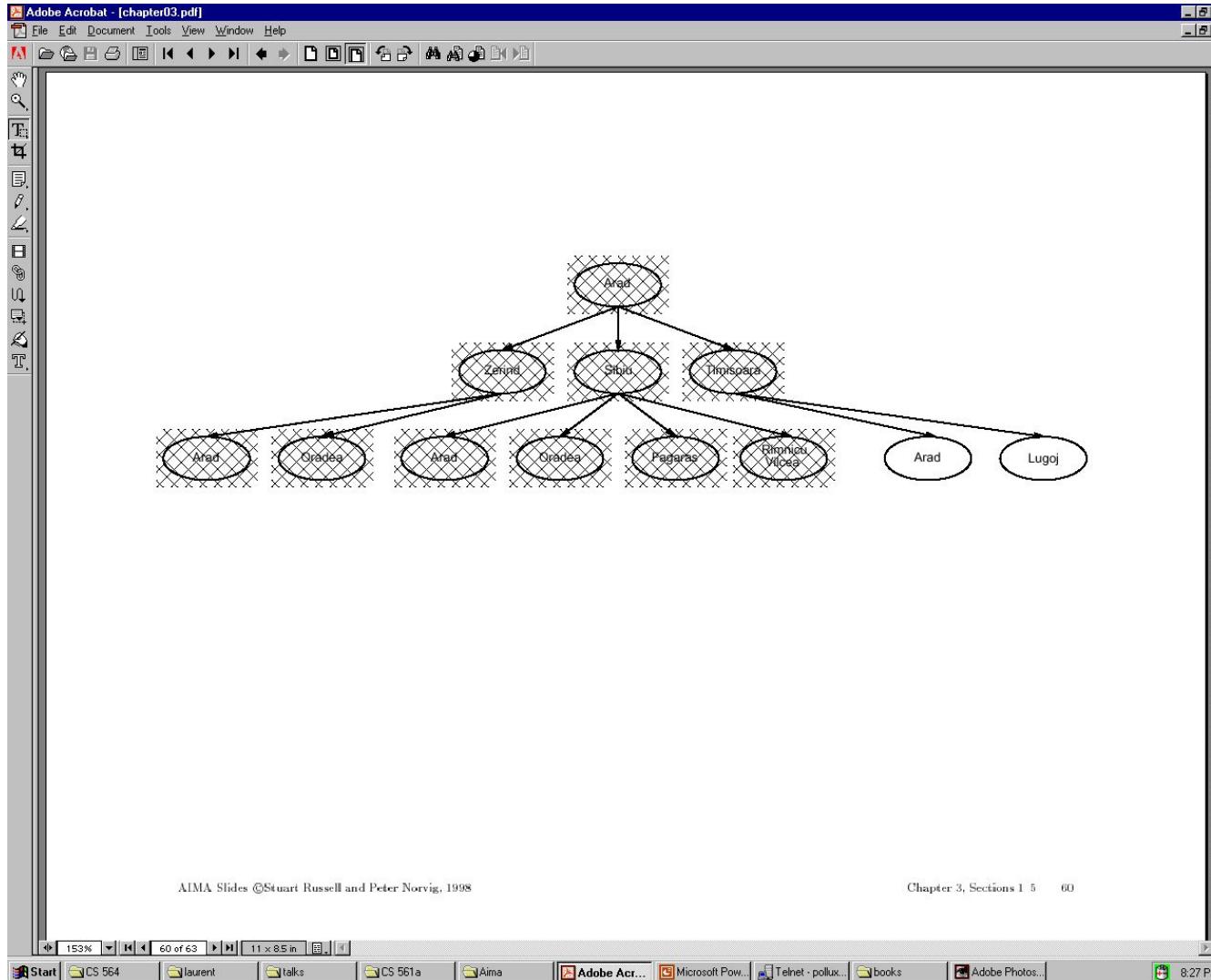












Iterative deepening complexity

- Iterative deepening search may seem wasteful because so many states are expanded multiple times.
- In practice, however, the overhead of these multiple expansions is small, because most of the nodes are towards leaves (bottom) of the search tree:
thus, the nodes that are evaluated several times (towards top of tree) are in relatively small number.

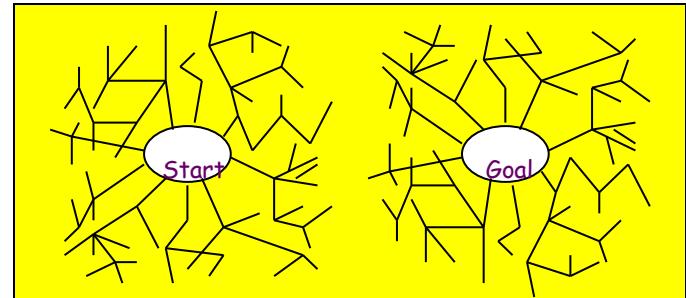
Iterative deepening complexity

- In iterative deepening, nodes at bottom level are expanded once, level above twice, etc. up to root (expanded $d+1$ times) so total number of expansions is:
$$(d+1)1 + (d)b + (d-1)b^2 + \dots + 3b^{(d-2)} + 2b^{(d-1)} + 1b^d = O(b^d)$$
- In general, iterative deepening is preferred to depth-first or breadth-first when search space large and depth of solution not known.

BI-DIRECTIONAL SEARCH

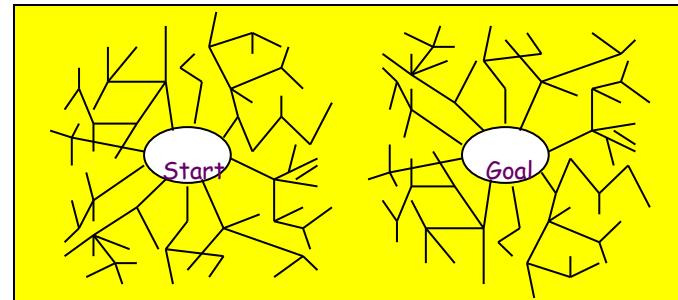
Bidirectional search

- Both search forward from initial state, and **backwards from goal**.
- Stop when the two searches meet in the middle.
- **Problem:** how do we search backwards from goal??
 - predecessor of node n = all nodes that have n as successor
 - this may not always be easy to compute!
 - if several goal states, apply predecessor function to them just as we applied successor (only works well if goals are explicitly known; may be difficult if goals only characterized implicitly).



Bidirectional search

- Problem: how do we search backwards from goal?? (cont.)
 - ...
 - for bidirectional search to work well, there must be an efficient way to check whether a given node belongs to the other search tree.
 - select a given search algorithm for each half.



Bidirectional search

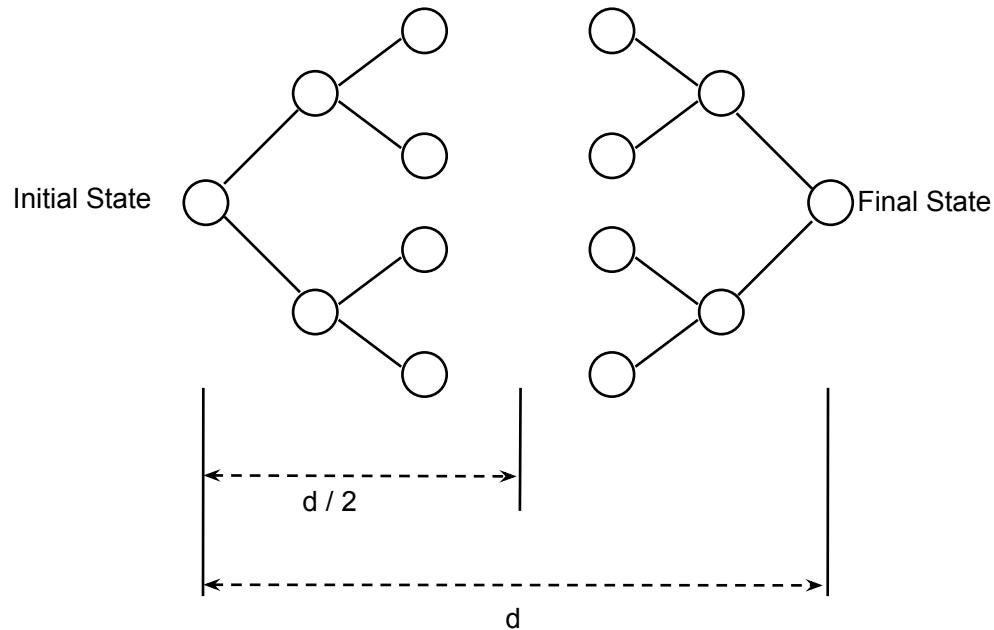
1. QUEUE1 <-- path only containing the root;
QUEUE2 <-- path only containing the goal;
2. WHILE both QUEUES are not empty
AND QUEUE1 and QUEUE2 do NOT share a state

DO remove their first paths;
create their new paths (to all children);
reject their new paths with loops;
add their new paths to back;
3. IF QUEUE1 and QUEUE2 share a state
THEN success;
ELSE failure;

Bidirectional search

- Completeness: Yes,
 - Time complexity: $2*O(b^{d/2}) = O(b^{d/2})$
 - Space complexity: $O(b^{m/2})$
 - Optimality: Yes
-
- To avoid one by one comparison, we need a hash table of size $O(b^{m/2})$
 - *If hash table is used, the cost of comparison is $O(1)$*

Bidirectional Search



Bidirectional search

- Bidirectional search merits:
 - Big difference for problems with branching factor b in both directions
 - A solution of length d will be found in $O(2b^{d/2}) = O(b^{d/2})$
 - For $b = 10$ and $d = 6$, only 2,222 nodes are needed instead of 1,111,111 for breadth-first search

Bidirectional search

- Bidirectional search issues
 - *Predecessors* of a node need to be generated
 - Difficult when operators are not reversible
 - What to do if there is no *explicit list of goal states*?
 - For each node: *check if it appeared in the other search*
 - Needs a hash table of $O(b^{d/2})$
 - What is the *best search strategy* for the two searches?

Comparing uninformed search strategies

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

- b – max branching factor of the search tree
- d – depth of the least-cost solution
- m – max depth of the state-space (may be infinity)
- l – depth cutoff

Summary

- Problem formulation usually requires **abstracting away real-world details** to define a **state space** that can be explored using computer algorithms.
- Once problem is formulated in abstract form, **complexity analysis** helps us picking out best algorithm to solve problem.
- Variety of uninformed search strategies; difference lies in method used to **pick node that will be further expanded**.
- **Iterative deepening** search only uses linear space and not much more time than other uninformed search strategies.

CSCI 561 - Foundation for Artificial Intelligence

04 Informed Search and Function Optimization

Professor Wei-Min Shen
University of Southern California

Outline

Informed Search

Use heuristics to guide the search

- Best first
- A*
- Heuristics

Function Optimization (Informed Search)

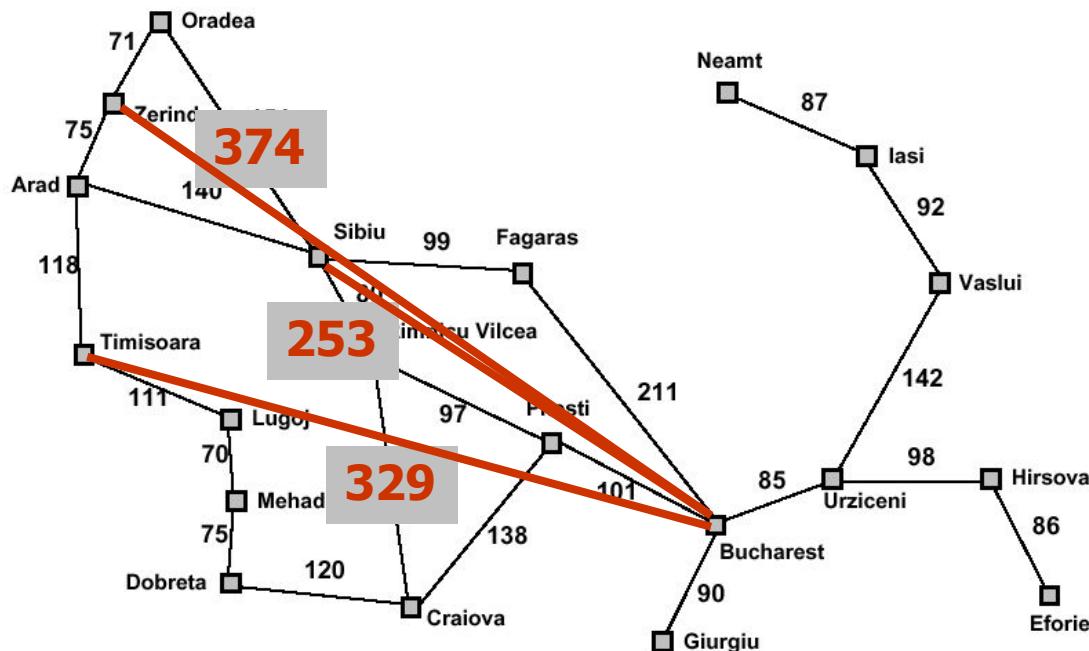
- Hill-climbing
- Simulated Annealing
- Genetic Algorithms

Best-First Search

- Idea:
 - use an evaluation function for each node; estimate of "*desirability*"
 - ⇒ expand most desirable unexpanded node.
- Implementation:

QueueingFn = insert successors in decreasing order of desirability
- Special cases:
 - uniformed-search (past cost only)
 - greedy search (future cost only)
 - A* search (sum of past and future cost)

Romania with step costs in km

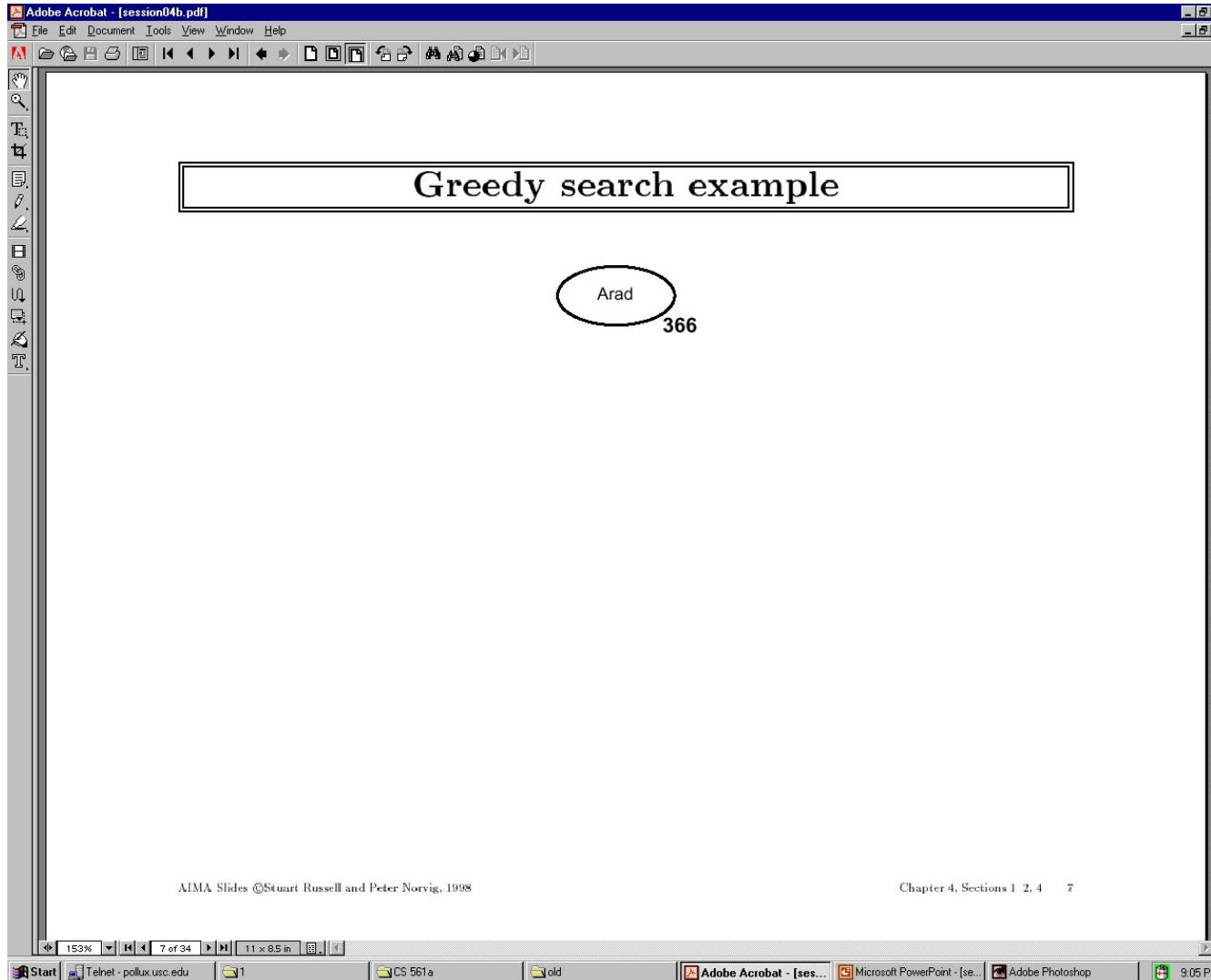


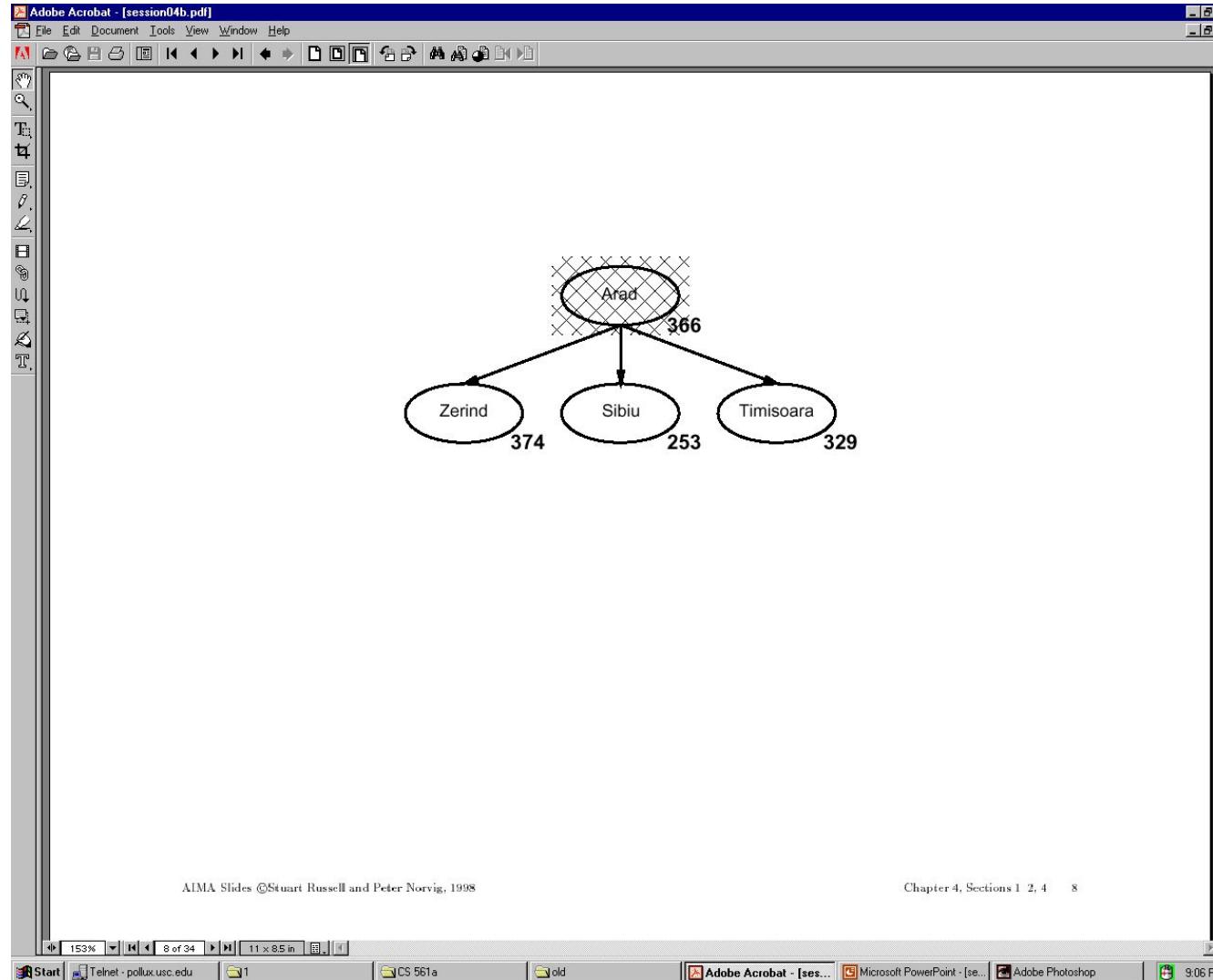
Straight-line distance
to Bucharest

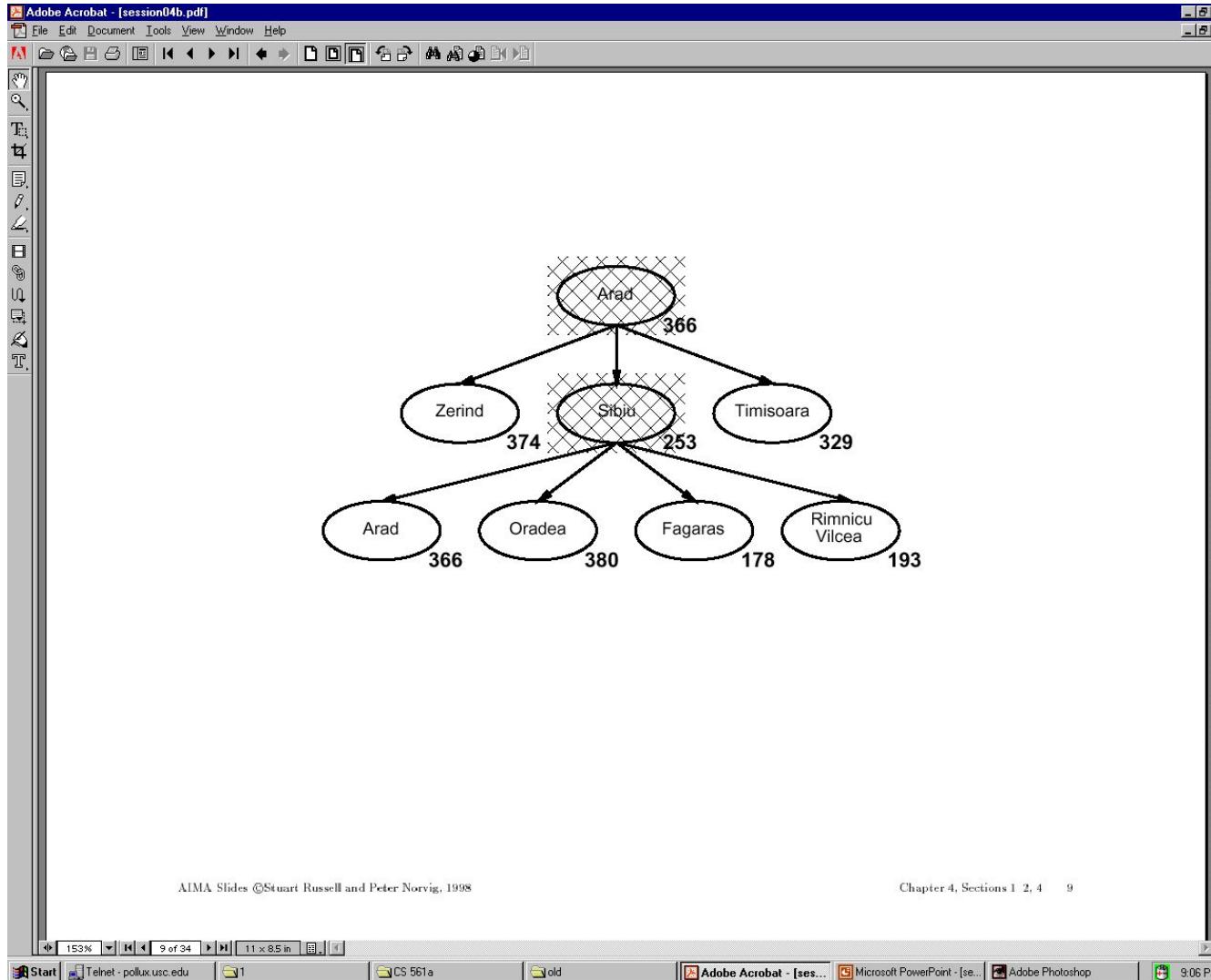
Arad	366
Bucharest	0
Craiova	160
Dobrete	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

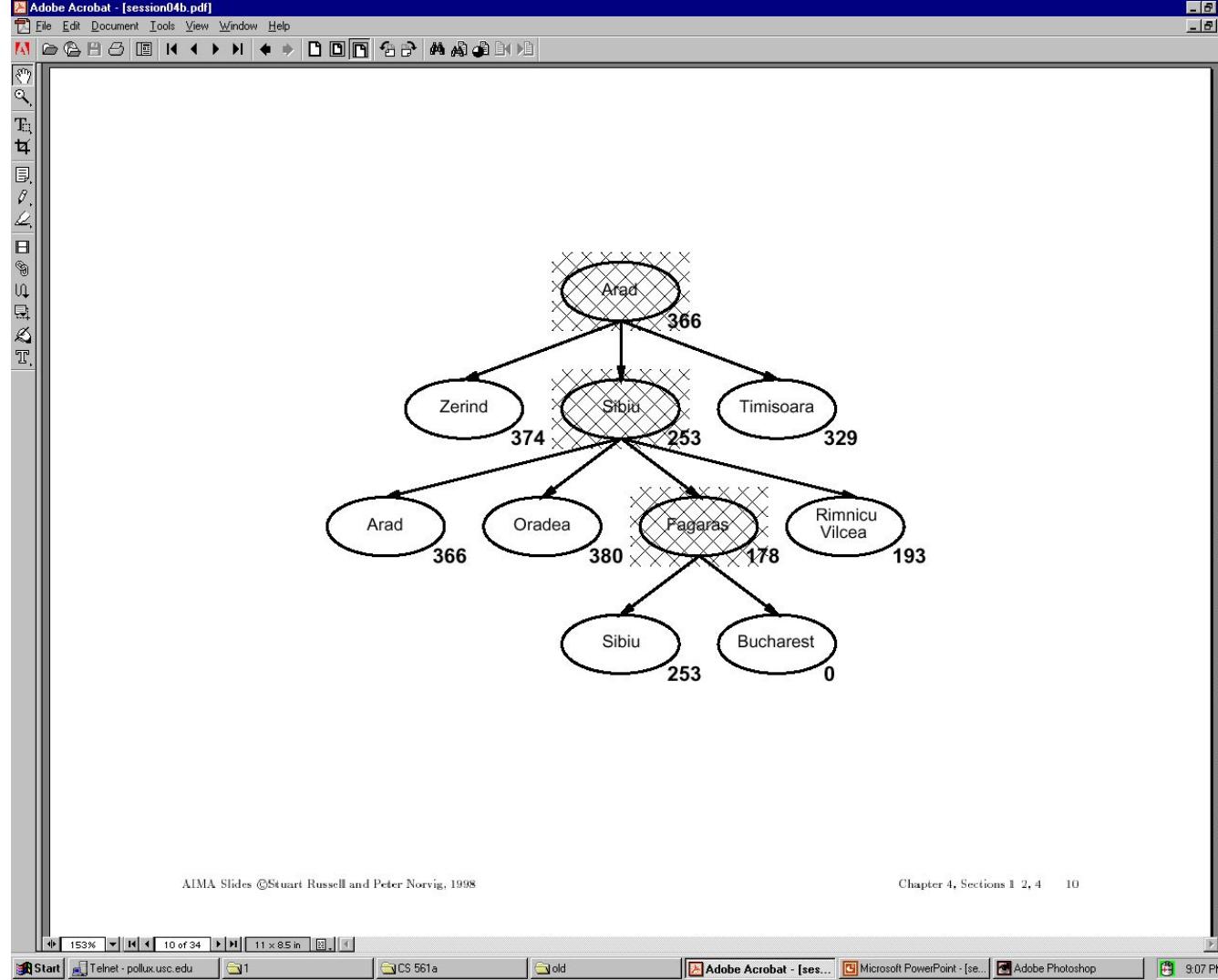
Greedy search

- Estimation function:
 $h(n)$ = estimate the “future” cost from now n to the goal (heuristic)
- For example:
 $h_{SLD}(n)$ = straight-line distance from n to Bucharest
- Greedy search expands first the node that appears to be closest to the goal, (or the least future cost), according to $h(n)$.









Properties of Greedy Search

- Complete?
- Time?
- Space?
- Optimal?

Properties of Greedy Search

- Complete? No – can get stuck in loops
e.g., Iasi > Neamt > Iasi > Neamt > ...
Complete in finite space with repeated-state checking.
- Time? $O(b^m)$ but a good heuristic can give dramatic improvement
- Space? $O(b^m)$ – keeps all nodes in memory
- Optimal? No.

A* search

- Idea: avoid expanding paths that are already expensive

evaluation function: $f(n) = g(n) + h(n)$ with:

$g(n)$ – “**past cost**” so far to reach now n

$h(n)$ – “**estimated future cost**” from n to the goal

$f(n)$ – estimated total cost of path through n to goal

- A* search uses an **admissible** heuristic, that is, “never over-estimate the future”!
 $h(n) \leq h^*(n)$ where $h^*(n)$ is the **true** cost from n
For example: the GPS distance $h_{SLD}(n)$ never overestimates the actual road distance
- Theorem: A* search is optimal

A* search

Admissible:
Never over-estimate the future cost

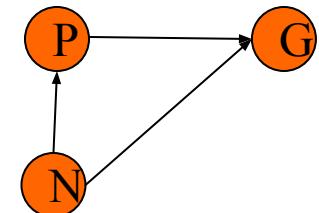
- A* search uses an **admissible** heuristic, that is,
 $h(n) \leq h^*(n)$ where $h^*(n)$ is the **true** “future” cost from n to a goal
- **Theorem:** A* search is optimal

- Note: A* is also optimal if the heuristic is **consistent**, i.e.,

$$h(N) \leq c(N, P) + h(P) \text{ and}$$

$$h(G) = 0.$$

Actual cost
From N to P



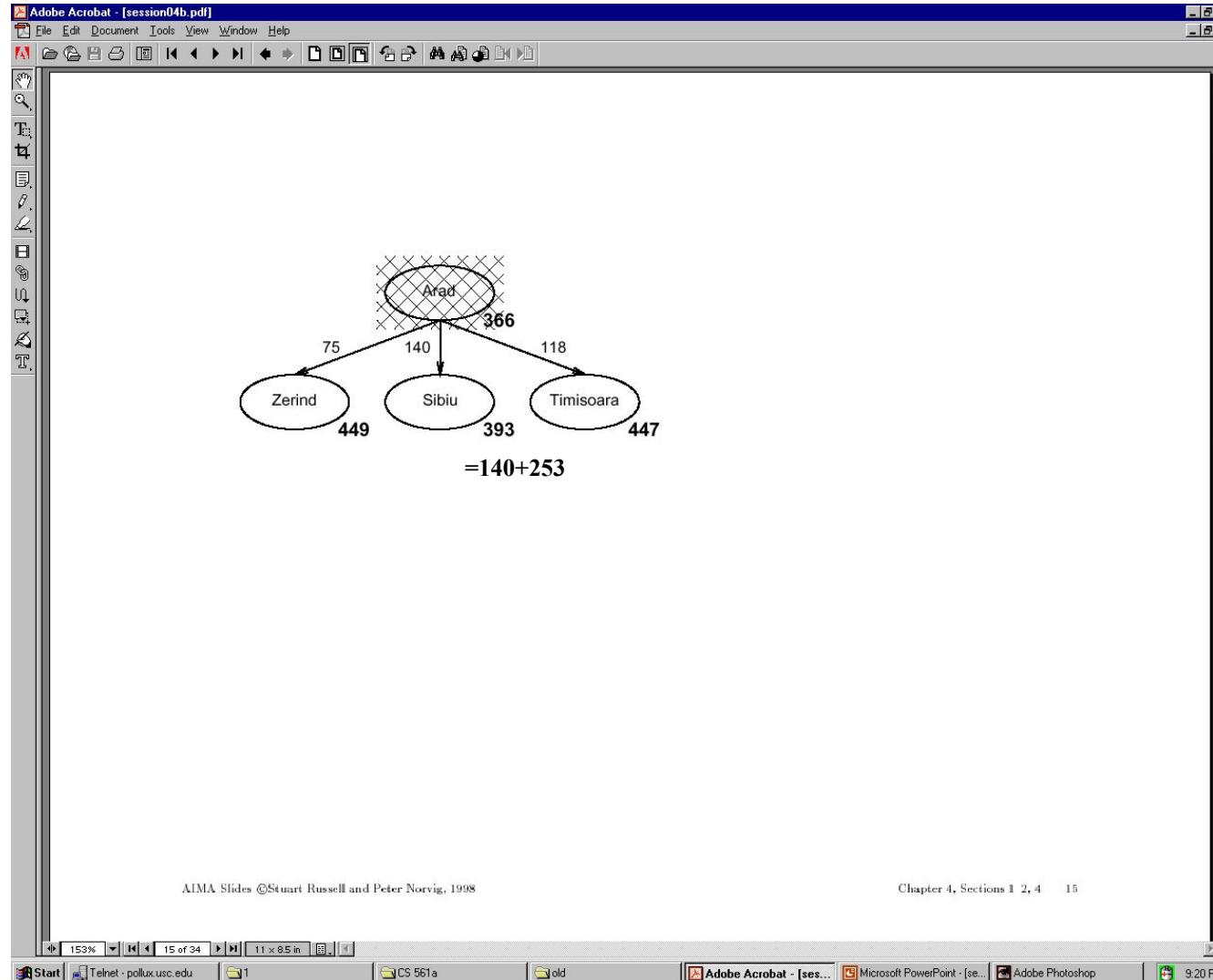
Consistent:
One step is never worse than two steps!

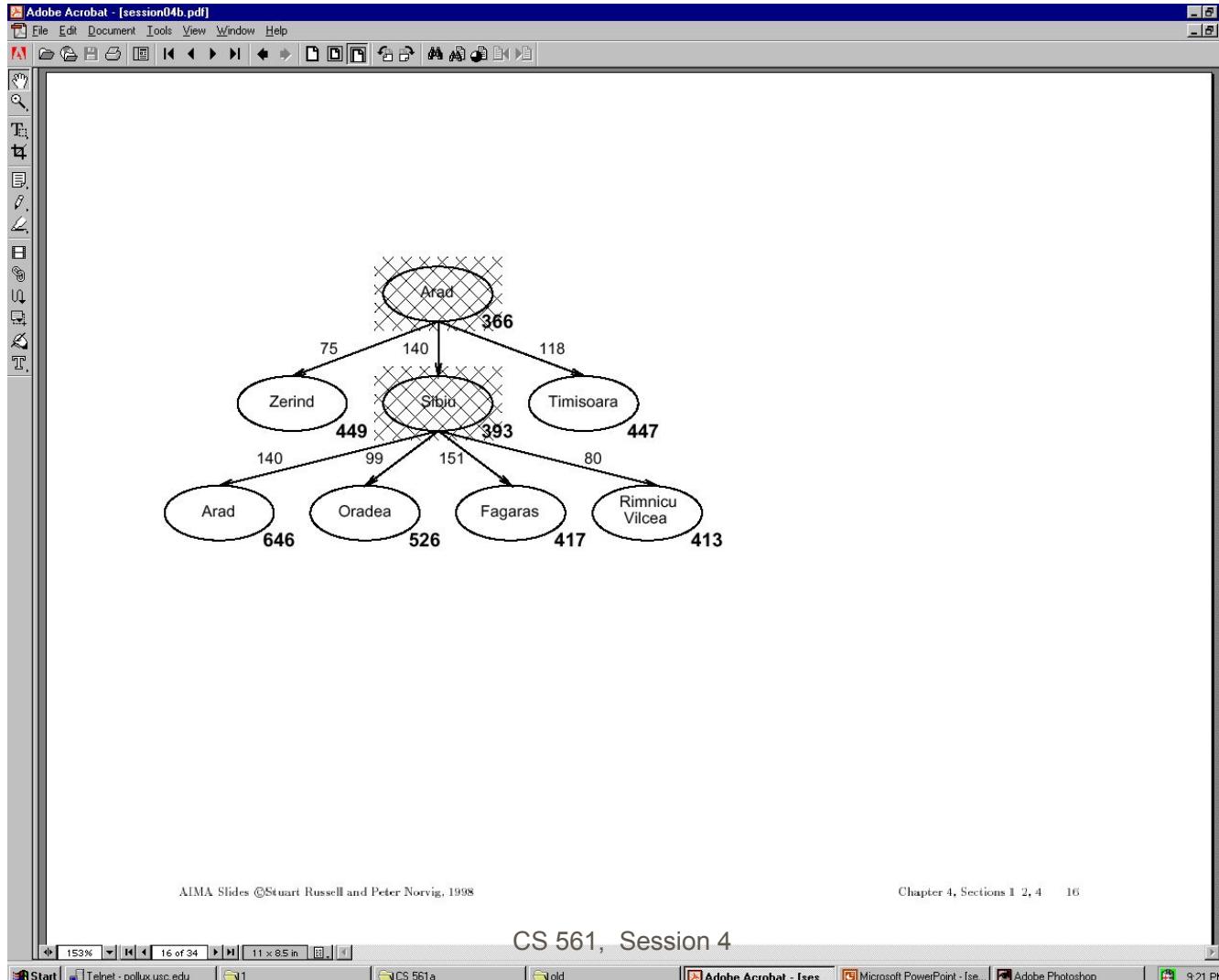
- (a consistent heuristic is admissible (by induction), but the converse is not always true)

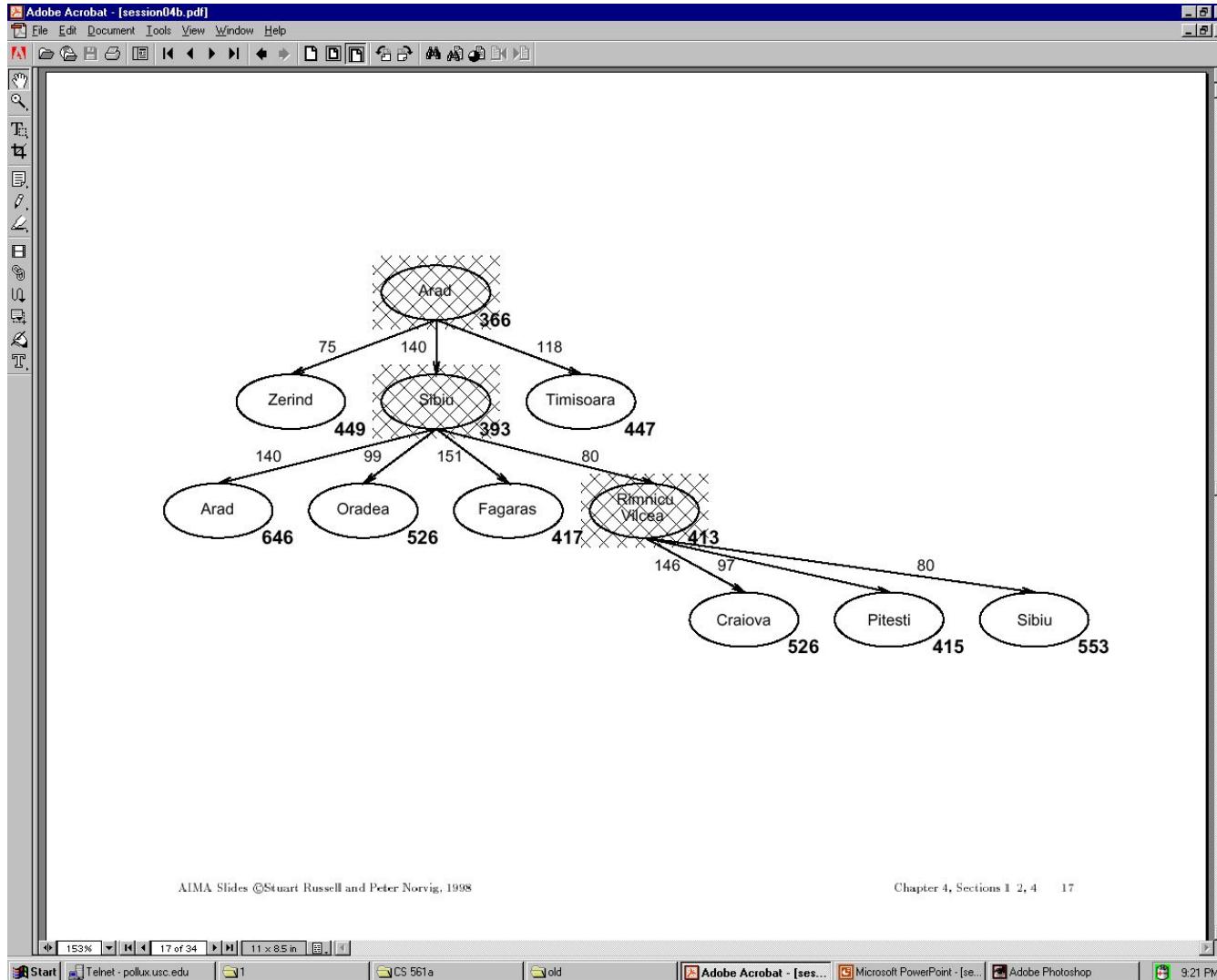
A* search example

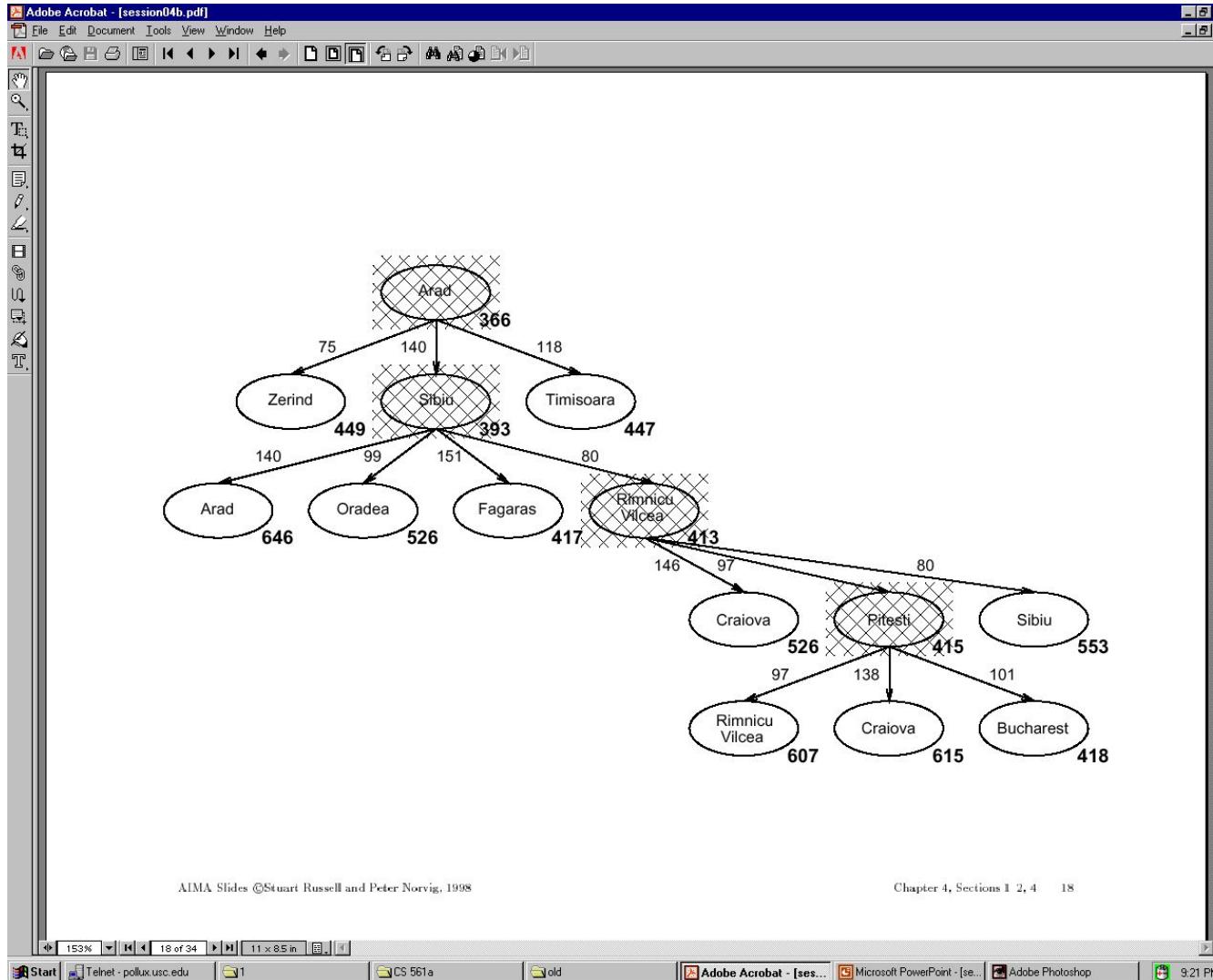
Arau

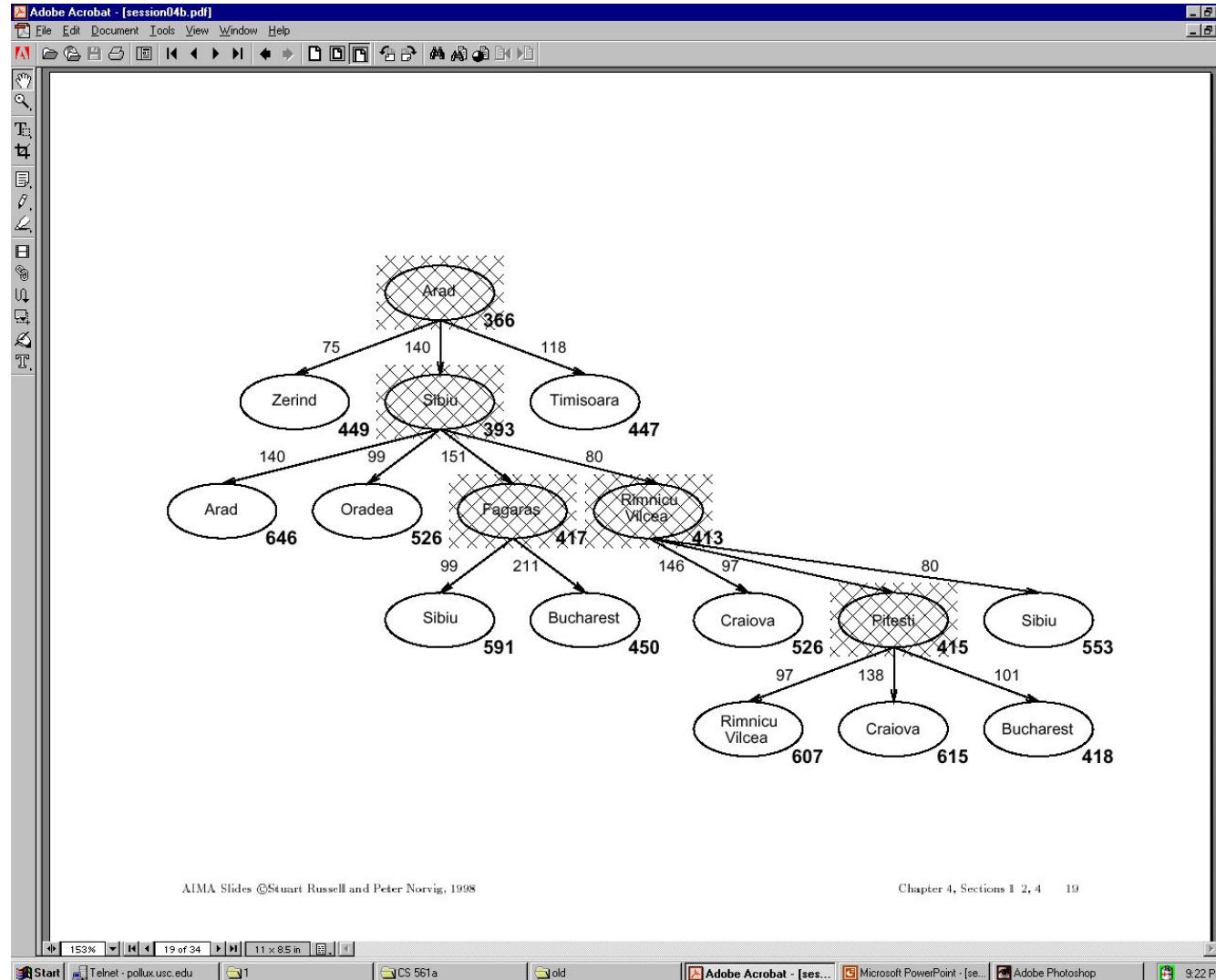
366





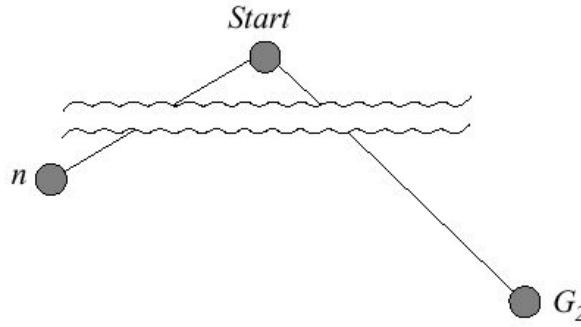






Optimality of A* (standard proof)

Suppose some suboptimal goal G_2 has been generated and is in the queue. Let n be an unexpanded node on a shortest path to an optimal goal G_1 .



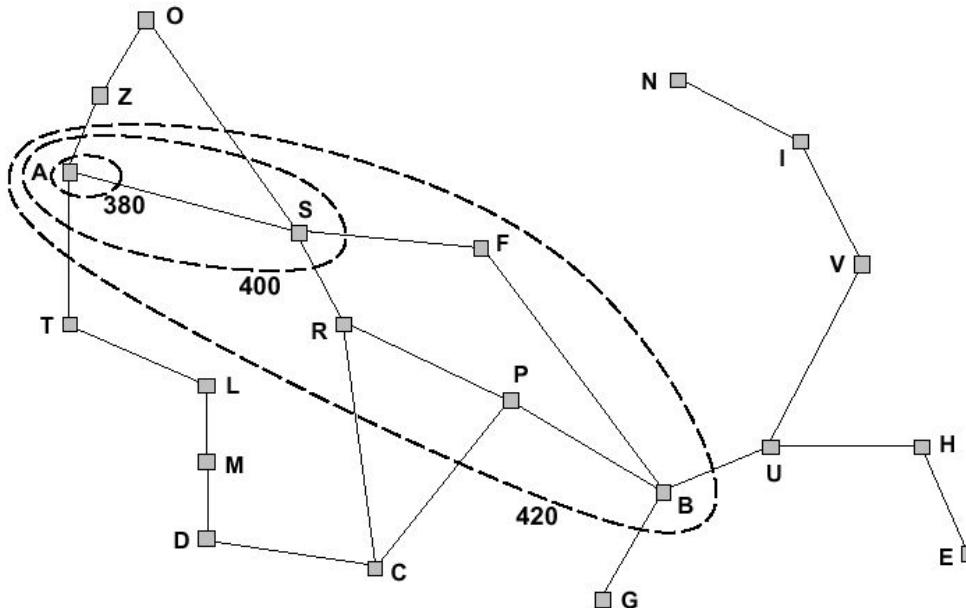
$$\begin{aligned} f(G_2) &= g(G_2) && \text{since } h(G_2) = 0 \\ &> g(G_1) && \text{since } G_2 \text{ is suboptimal} \\ &\geq f(n) && \text{since } h \text{ is admissible} \end{aligned}$$

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

Optimality of A* (more useful proof)

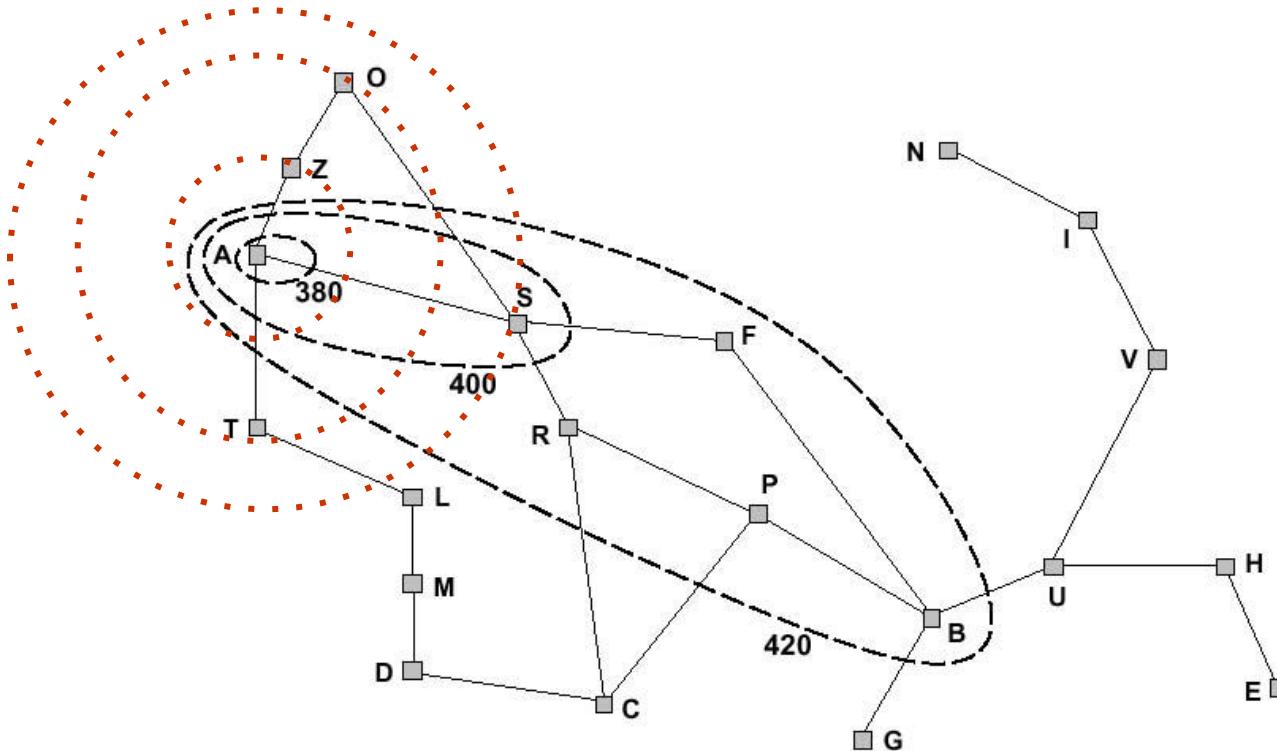
Lemma: A* expands nodes in order of increasing f value

Gradually adds “ f -contours” of nodes (cf. breadth-first adds layers)
Contour i has all nodes with $f = f_i$, where $f_i < f_{i+1}$



f-contours

How do the contours look like when $h(n) = 0$?



Properties of A*

- Complete?
- Time?
- Space?
- Optimal?

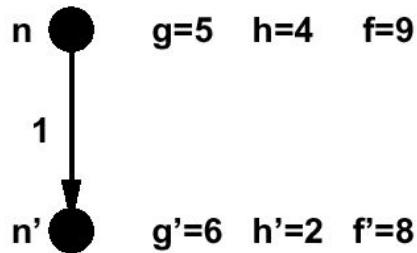
Properties of A*

- Complete? Yes, unless infinitely many nodes with $f \leq f(G)$
- Time? Exponential in [(relative error in h) \times (length of solution)]
- Space? Keeps all nodes in memory
- Optimal? Yes – cannot expand f_{i+1} until f_i is finished

Proof of lemma: pathmax

For some admissible heuristics, f may *decrease* along a path

E.g., suppose n' is a successor of n



But this throws away information!

$f(n) = 9 \Rightarrow$ true cost of a path through n is ≥ 9

Hence true cost of a path through n' is ≥ 9 also

Pathmax modification to A*:

Instead of $f(n') = g(n') + h(n')$, use $f(n') = \max(g(n') + h(n'), f(n))$

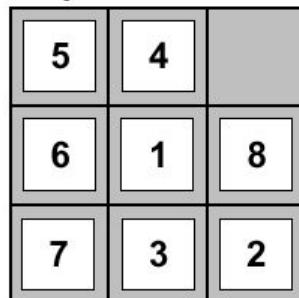
With pathmax, f is always nondecreasing along any path

Admissible heuristics

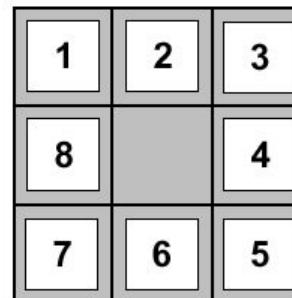
E.g., for the 8-puzzle:

$h_1(n)$ = number of misplaced tiles

$h_2(n)$ = total Manhattan distance
(i.e., no. of squares from desired location of each tile)



Start State



Goal State

$$h_1(S) = ??$$

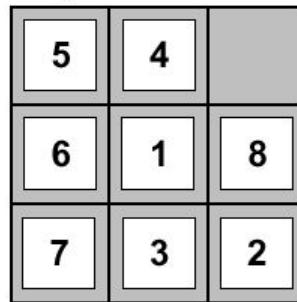
$$\underline{h_2(S) = ??}$$

Admissible heuristics

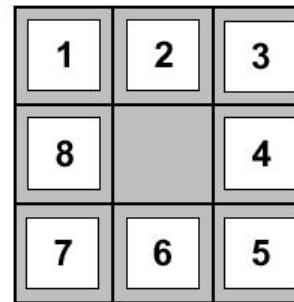
E.g., for the 8-puzzle:

$h_1(n)$ = number of misplaced tiles

$h_2(n)$ = total Manhattan distance
(i.e., no. of squares from desired location of each tile)



Start State



Goal State

$$h_1(S) = ?? \ 7$$

$$\underline{h_2(S)} = ?? \ 2+3+3+2+4+2+0+2 = 18$$

Define Heuristics by Relaxing Problem

- Admissible heuristics can be derived from the exact solution cost of a relaxed version of the 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.

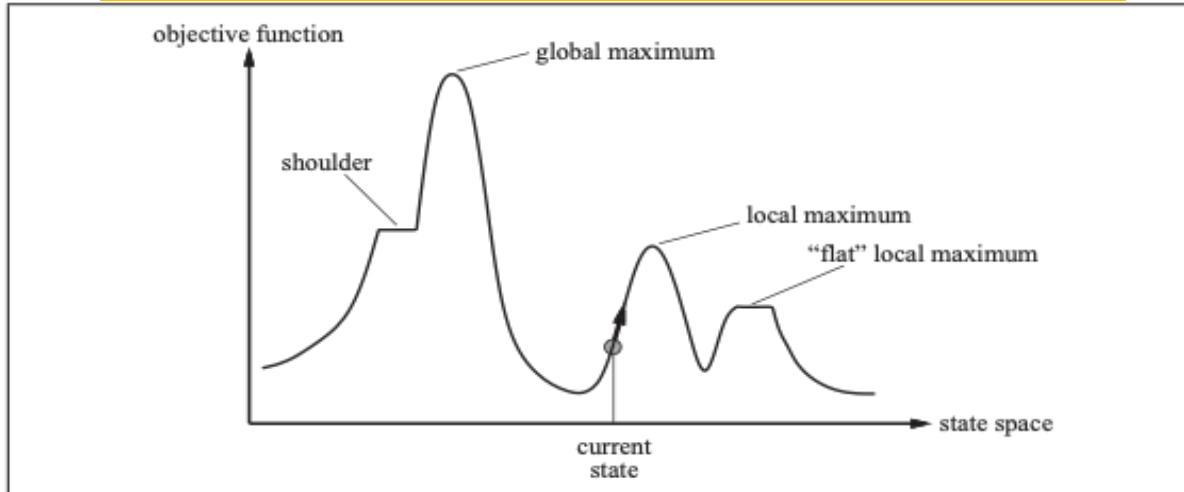
FUNCTION OPTIMIZATION

Function Optimization

One of the most fundamental Problems

- What is optimization?
- All (almost) engineering/AI problems are optimizations! Why?

The landscape of the function to be optimized



Function Optimization

- Iterative improvement
- Hill climbing
- Simulated annealing
- Genetic Algorithms

Iterative improvement

- In many optimization problems, **path** is irrelevant; the goal state itself is the solution.
- Then, state space = space of “**complete**” configurations.
Algorithm goal:
 - find optimal configuration (e.g., TSP), or,
 - find configuration satisfying constraints
(e.g., n-queens)
- In such cases, can use **iterative improvement algorithms**: keep a single “**current**” state, and try to improve it.

Iterative improvement example: vacuum world

Simplified world: 2 locations, each may or not contain dirt,
each may or not contain vacuuming agent.

Goal of agent: clean up the dirt.

If path does not matter, do not need to keep track of it.

Single-state, start in #5. Solution??

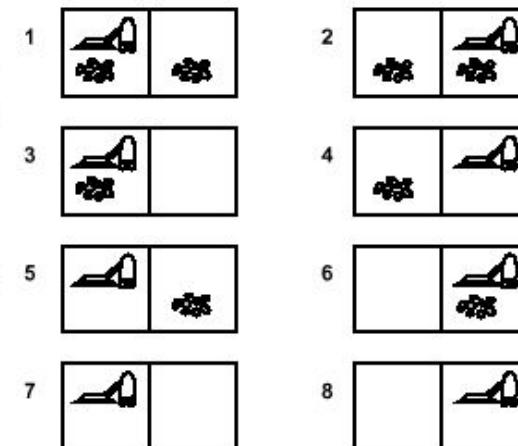
Multiple-state, start in $\{1, 2, 3, 4, 5, 6, 7, 8\}$
e.g., Right goes to $\{2, 4, 6, 8\}$. Solution??

Contingency, start in #5

Murphy's Law: *Suck* can dirty a clean carpet

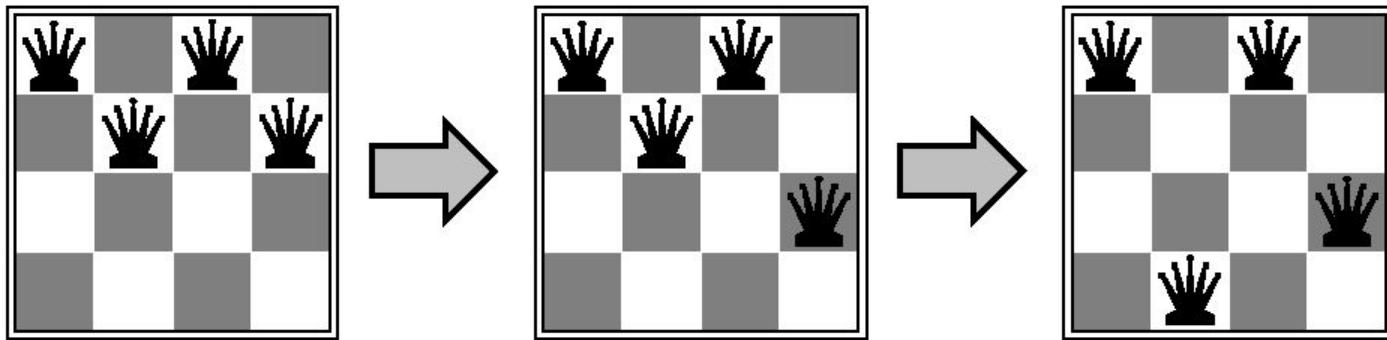
Local sensing: dirt, location only.

Solution??



Iterative improvement example: n-queens

- **Goal:** Put n chess-game queens on an $n \times n$ board, with no two queens on the same row, column, or diagonal.



- Here, goal state is initially unknown but is specified by constraints that it must satisfy.

Hill climbing (or gradient ascent/descent)

- Iteratively maximize “**value**” of current state, by replacing it by successor state that has highest value, as long as possible.

“Like climbing Everest in thick fog with amnesia”

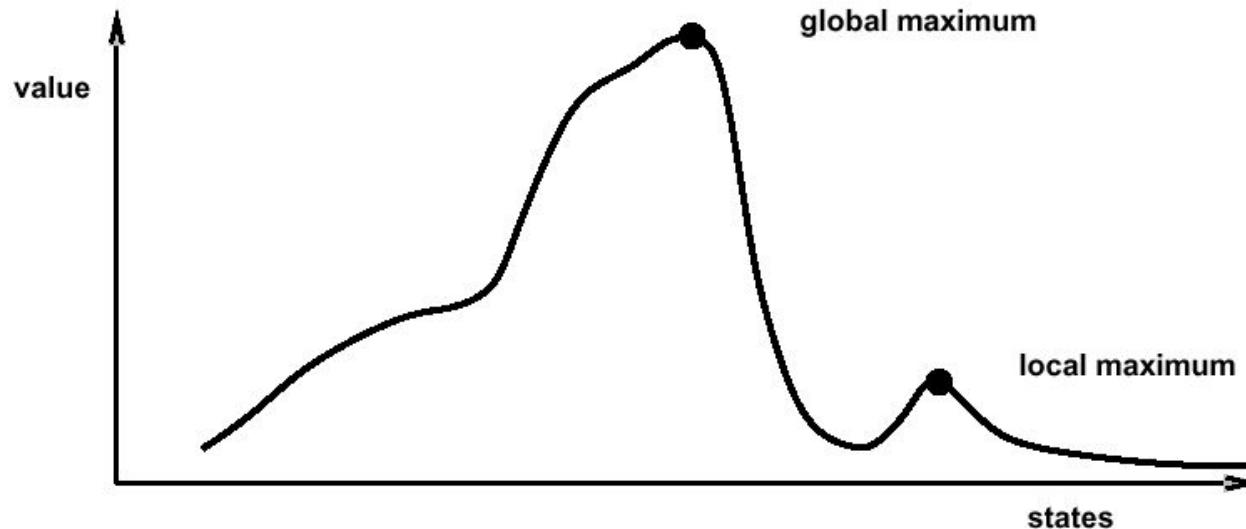
```
function HILL-CLIMBING(problem) returns a solution state
    inputs: problem, a problem
    local variables: current, a node
                  next, a node
    current  $\leftarrow$  MAKE-NODE(INITIAL-STATE[problem])
    loop do
        next  $\leftarrow$  a highest-valued successor of current
        if VALUE[next] < VALUE[current] then return current
        current  $\leftarrow$  next
    end
```

Hill climbing

- Note: minimizing a “value” function $v(n)$ is equivalent to maximizing $-v(n)$, thus both notions are used interchangeably.
- Notion of “**extremization**”: find extrema (minima or maxima) of a value function.

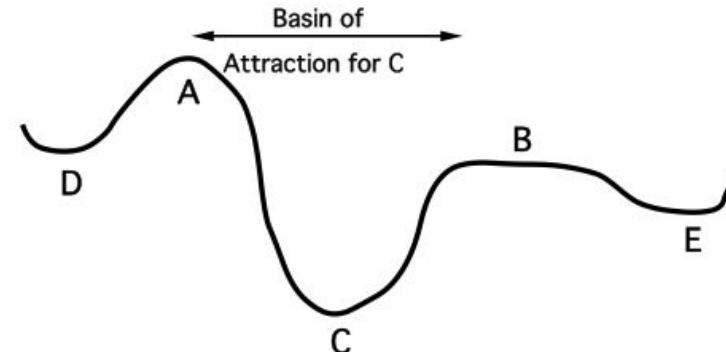
Hill climbing

- **Problem:** depending on initial state, may get stuck in local extremum.



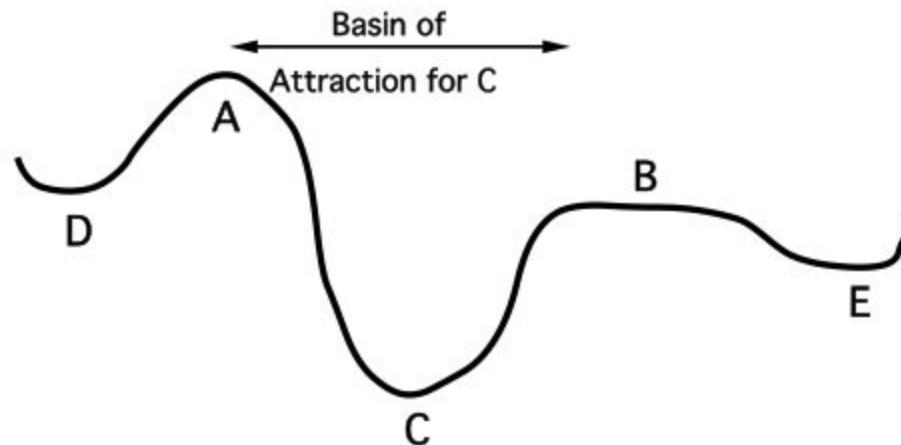
Minimizing energy

- Let's now change the formulation of the problem a bit, so that we can employ new formalism:
 - let's compare our state space to that of a physical system that is subject to natural interactions,
 - and let's compare our value function to the overall potential energy E of the system.
- On every updating,
we have $\Delta E \leq 0$



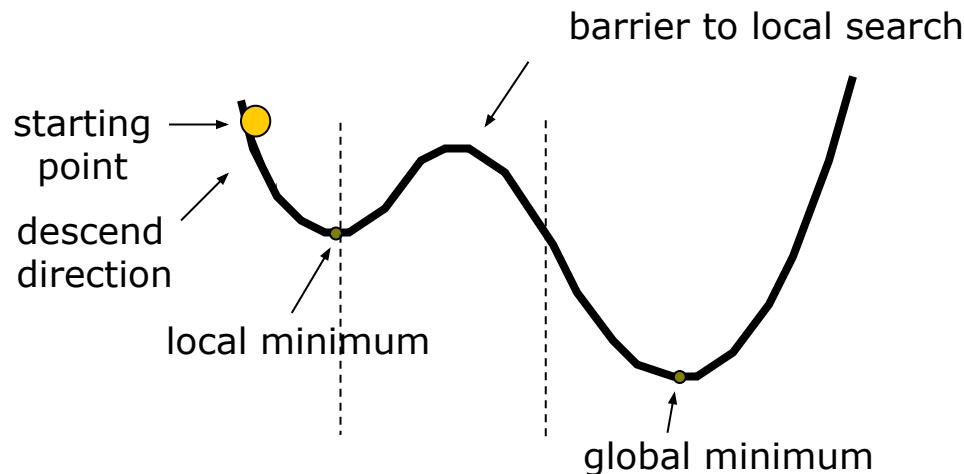
Minimizing energy

- Hence the dynamics of the system tend to move E toward a minimum.
- We stress that there may be different such states — they are *local* minima. Global minimization is not guaranteed.



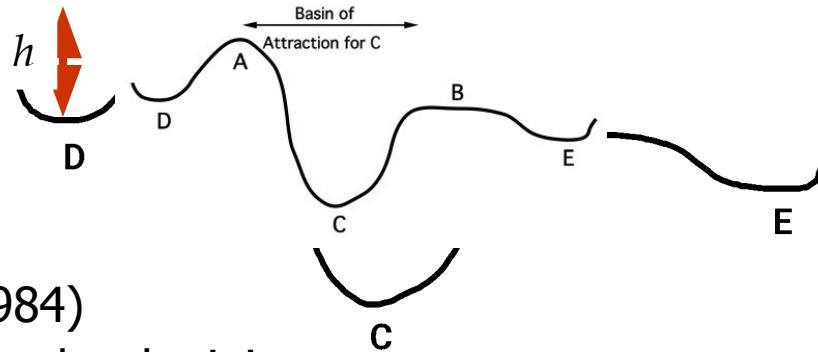
Local Minima Problem

- Question: How do you avoid this local minimum?



Boltzmann machines

The Boltzmann Machine of Hinton, Sejnowski, and Ackley (1984) uses simulated annealing to escape local minima.

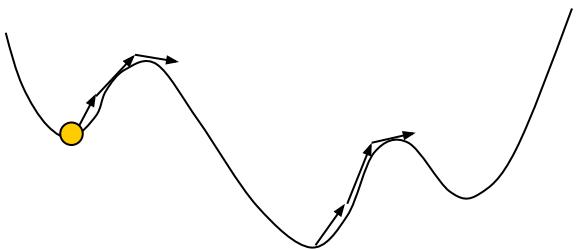


To motivate their solution, consider how one might get a ball-bearing traveling along the curve to "probably end up" in the deepest minimum. The idea is to shake the box "about h hard" — then the ball is more likely to go from D to C than from C to D. So, on average, the ball should end up in C's valley.

Consequences of the Occasional Ascents

desired effect

Help escaping the local optima.



adverse effect

Might pass global optima after reaching it

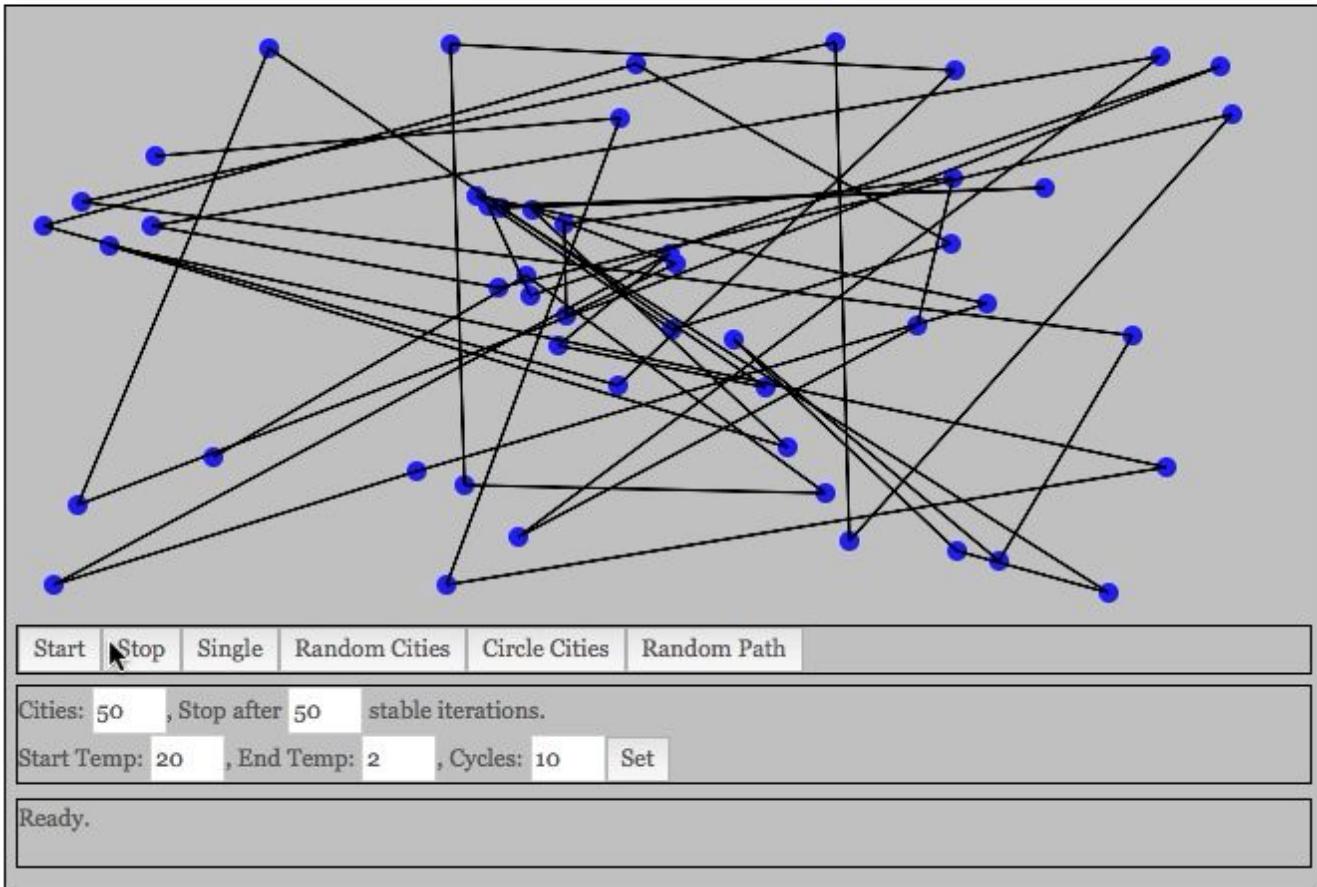
(easy to avoid by keeping track of best-ever state)

Simulated annealing: basic idea

- From current state, pick a **random** successor state;
- If it has better value than current state, then “accept the transition,” that is, use successor state as current state;
- Otherwise, do not give up, but instead flip a coin and accept the transition with a given probability (that is lower as the successor is worse).
- So we accept to sometimes “un-optimize” the value function a little with a non-zero probability.

Demo

AIFH Volume 1, Chapter 9: Traveling Salesman (TSP): Simulated Annealing



Boltzmann's statistical theory of gases

- In the statistical theory of gases, the gas is described not by a deterministic dynamics, but rather by the probability that it will be in different states.
- The 19th century physicist [Ludwig Boltzmann](#) developed a theory that included a probability distribution of temperature (i.e., every small region of the gas had the same kinetic energy).
- Hinton, Sejnowski and Ackley's idea was that this distribution might also be used to describe neural interactions, where low temperature T is replaced by a small noise term T (the neural analog of random thermal motion of molecules). While their results primarily concern optimization using neural networks, the idea is more general.

Boltzmann distribution

- At thermal equilibrium at temperature T, the Boltzmann distribution gives the relative probability that the system will occupy state A vs. state B as:

$$\frac{P(A)}{P(B)} = \exp\left(-\frac{E(A) - E(B)}{T}\right) = \frac{\exp(E(B)/T)}{\exp(E(A)/T)}$$

- where E(A) and E(B) are the energies associated with states A and B.

Simulated annealing

Kirkpatrick et al. 1983:

- Simulated annealing is a general method for making likely the escape from local minima by allowing jumps to higher energy states.
- The analogy here is with the process of annealing used by a craftsman in forging a sword from an alloy.
- He heats the metal, then slowly cools it as he hammers the blade into shape.
 - If he cools the blade too quickly the metal will form patches of different composition;
 - If the metal is cooled slowly while it is shaped, the constituent metals will form a uniform alloy.



Simulated annealing in practice

- Set a temperature T
 - optimize for the given T
 - lower T (see Geman & Geman, 1984)
 - repeat
-
- Geman & Geman (1984): if T is lowered sufficiently slowly (with respect to the number of iterations used to optimize at a given T), simulated annealing is guaranteed to find the global minimum.
 - **Caveat:** this algorithm has no end (Geman & Geman's T decrease schedule is in the $1/\log$ of the number of iterations, so, T will never reach zero), so it may take an infinite amount of time for it to find the global minimum.

Simulated annealing algorithm

- Idea: Escape local extrema by allowing “bad moves,” but gradually decrease their size and frequency.

```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
    inputs: 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
         $\Delta E \leftarrow \text{VALUE}[\textit{next}] - \text{VALUE}[\textit{current}]$ 
        if  $\Delta E > 0$  then current  $\leftarrow$  next
        else current  $\leftarrow$  next only with probability  $e^{\Delta E / T}$ 
```

Note: goal here is to maximize E.

Simulated annealing algorithm

- Idea: Escape local extrema by allowing “bad moves,” but gradually decrease their size and frequency.

```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
    inputs: 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
         $\Delta E \leftarrow \text{VALUE}[\textit{next}] - \text{VALUE}[\textit{current}]$ 
        if  $\Delta E < 0$  then current  $\leftarrow$  next
        else current  $\leftarrow$  next only with probability  $e^{\Delta E / T}$ 
```

Algorithm when goal
is to minimize E.

Note on simulated annealing: limit cases

- Boltzmann distribution: accept “bad move” with $\Delta E < 0$ (goal is to maximize E) with probability $P(\Delta E) = \exp(\Delta E/T)$
- If T is large: $\Delta E < 0$
 - $\Delta E/T < 0$ and very small
 - $\exp(\Delta E/T)$ close to 1
 - accept bad move with high probability
- If T is near 0: $\Delta E < 0$
 - $\Delta E/T < 0$ and very large
 - $\exp(\Delta E/T)$ close to 0
 - accept bad move with low probability

Note on simulated annealing: limit cases

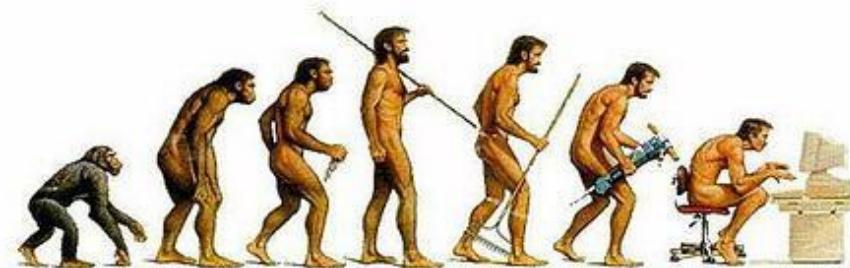
- Boltzmann distribution: accept “bad move” with $\Delta E < 0$ (goal is to maximize E) with probability $P(\Delta E) = \exp(\Delta E/T)$
- If T is large: $\Delta E < 0$
 $\Delta E/T < 0$ and very small
 $\exp(\Delta E/T)$ close to 1
accept bad move with high probability
- If T is near 0: $\Delta E < 0$
 $\Delta E/T < 0$ and very large
 $\exp(\Delta E/T)$ close to 0
accept bad move with low probability

Random walk

Deterministic
down-hill

GENETIC ALGORITHM

Genetic Algorithms



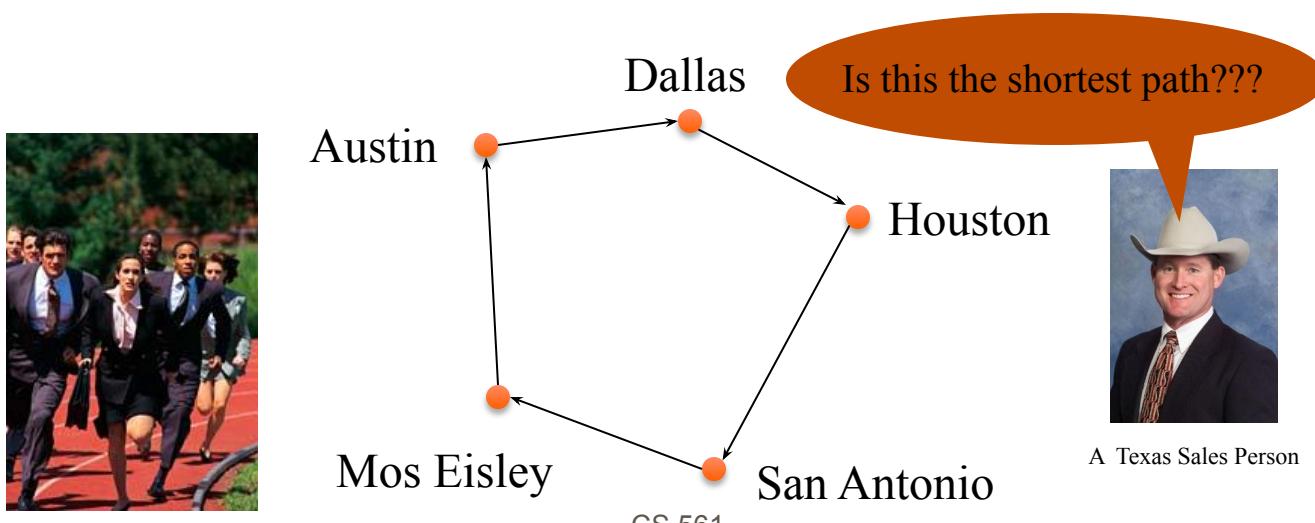
How do you find a solution in a large complex space?

- Ask an expert?
- Adapt existing designs?
- Trial and error?



Example: Traveling Sales-Person (TSP)

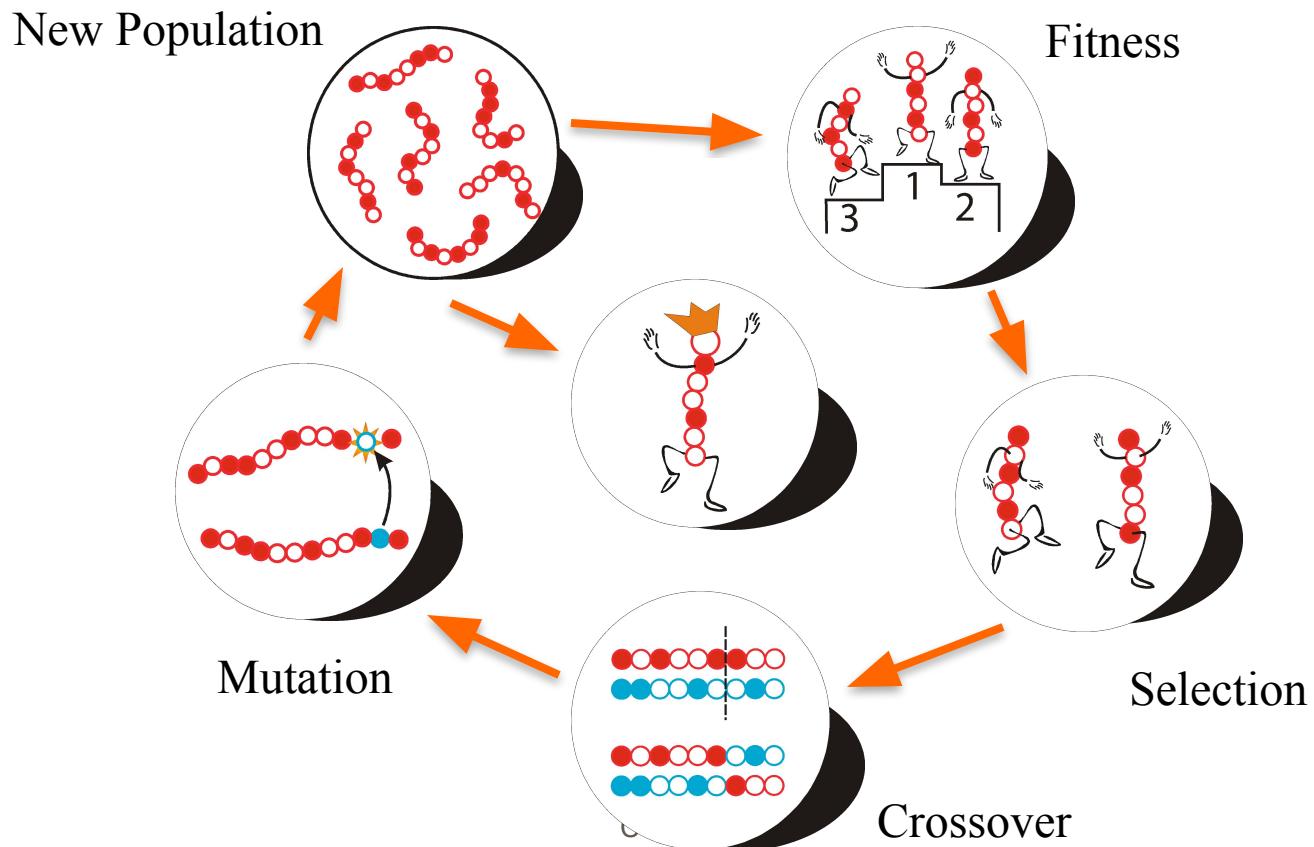
- Classic Example: You have N cities, find the shortest route such that your salesperson will visit each city once and return.
- This problem is known to be **NP-Hard**
 - As a new city is added to the problem, computation time in the classic solution increases exponentially $O(2^n)$... *(as far as we know)*



What if.....

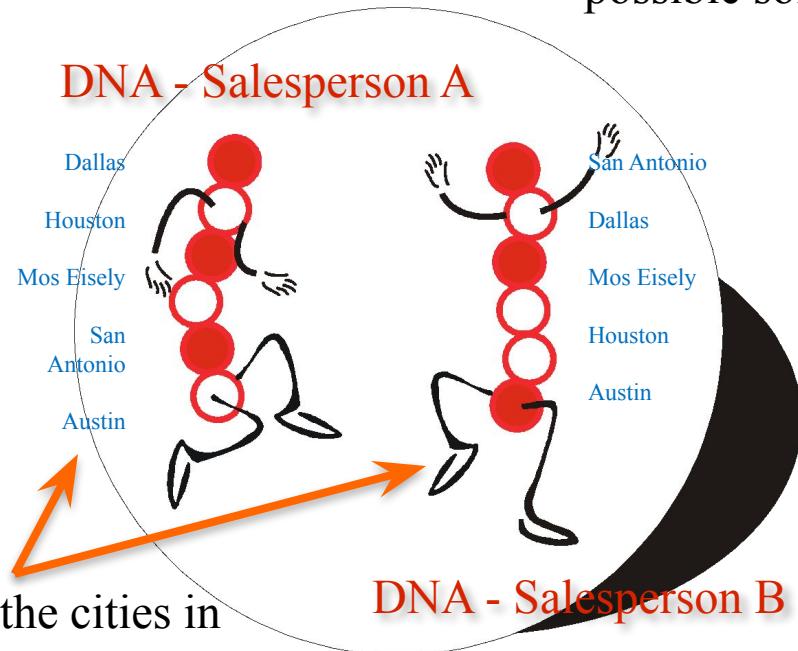
- Let us create a whole bunch of random sales people and see how well they do and pick the best one(s).
 - Salesperson A
 - Houston -> Dallas -> Austin -> San Antonio -> Mos Eisely
 - Distance Traveled 780 Km
 - Salesperson B
 - Houston -> Mos Eisley -> Austin -> San Antonio -> Dallas
 - Distance Traveled 820 Km
 - Salesperson A is better (more fit) than salesperson B
 - Perhaps we would like sales people to be more like **A** and less like **B**
- Question:
 - do we want to just keep picking random sales people like this and keep testing them?

Overview of the GA Cycle



Represent problem like a DNA sequence

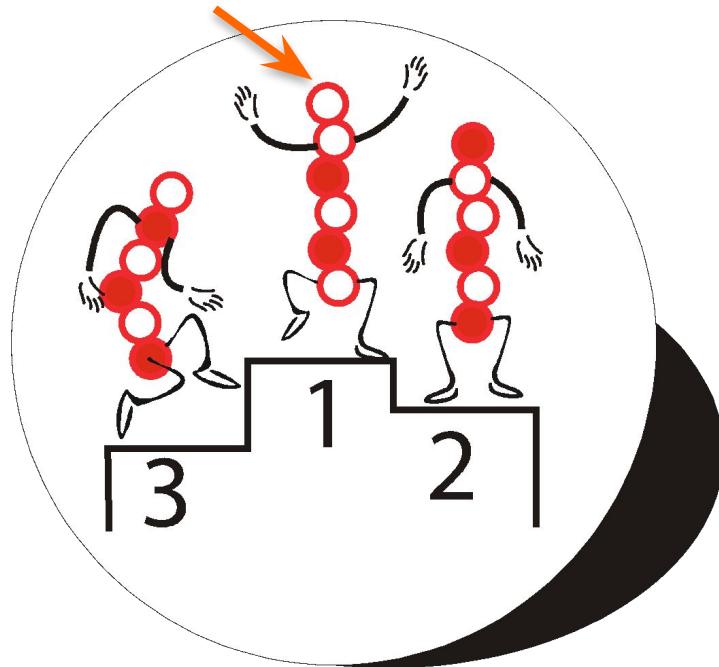
Each DNA sequence is an encoding of a possible solution to the problem.



The order of the cities in the genes is the order of the cities the TSP will take.

Ranking by Fitness:

Travels Shortest Distance



Here we've created three different salespeople. We then checked to see how far each one has to travel. This gives us a measure of “Fitness”

Note: we need to be able to measure fitness in polynomial time, otherwise we are in trouble.

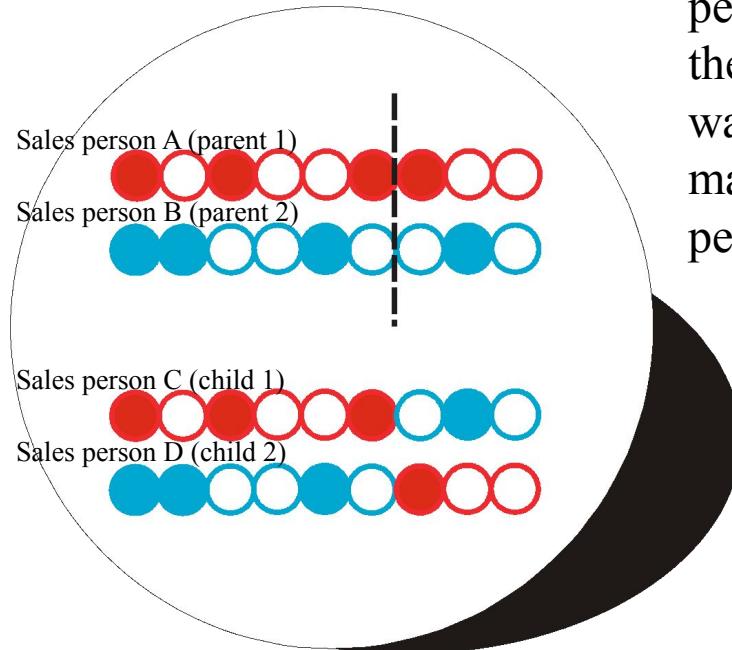
Let's breed them!

- We have a population of traveling sales people. We also know their fitness based on how long their trip is. We want to create more, but we don't want to create **too many**.
- We take the notion that the salespeople who perform better are closer to the optimal salesperson than the ones which performed more poorly. Could the optimal sales person be a "combination" of the better sales people?
- We create a **population** of sales people as **solutions** to the problem.
- *How do we actually mate a population of data???*



Crossover:

Exchanging information through some part of information (representation)

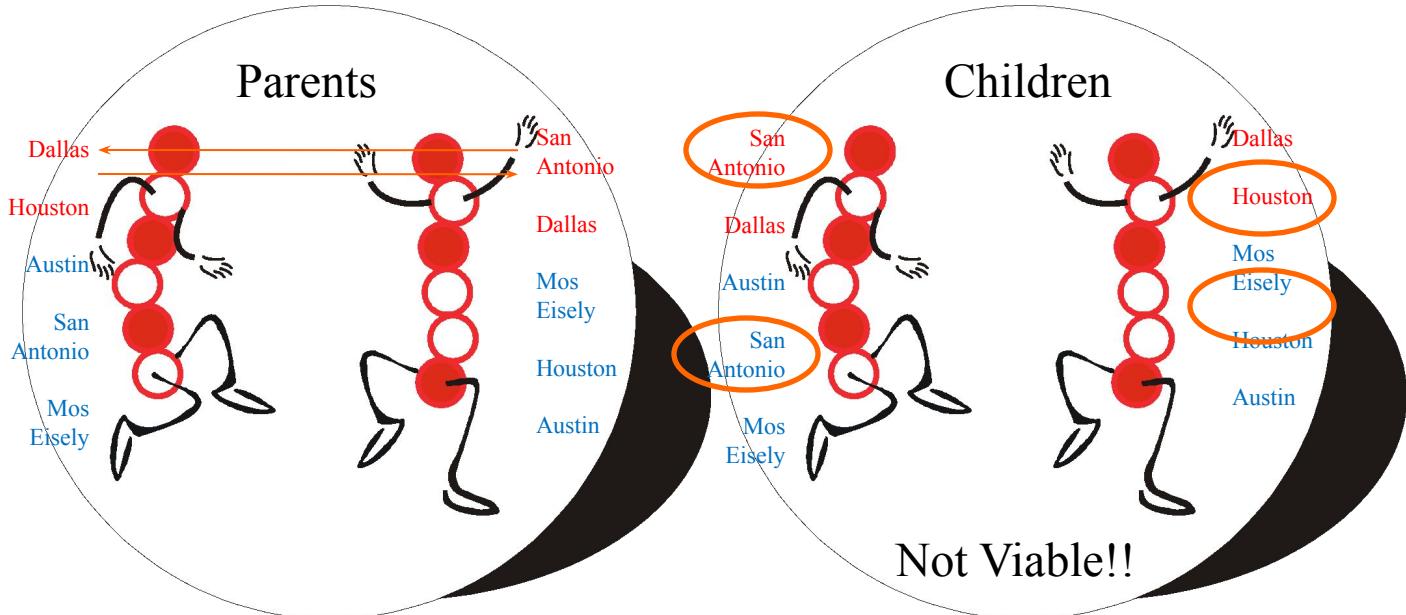


Once we have found the best sales people we will in a sense mate them. We can do this in several ways. Better sales people should mate more often and poor sales people should mate less often.

Sales People	City DNA
Parent 1	F A B E C G D
Parent 2	D E A C G B F
Child 1	F A B C G B F
Child 2	D E A E C G D

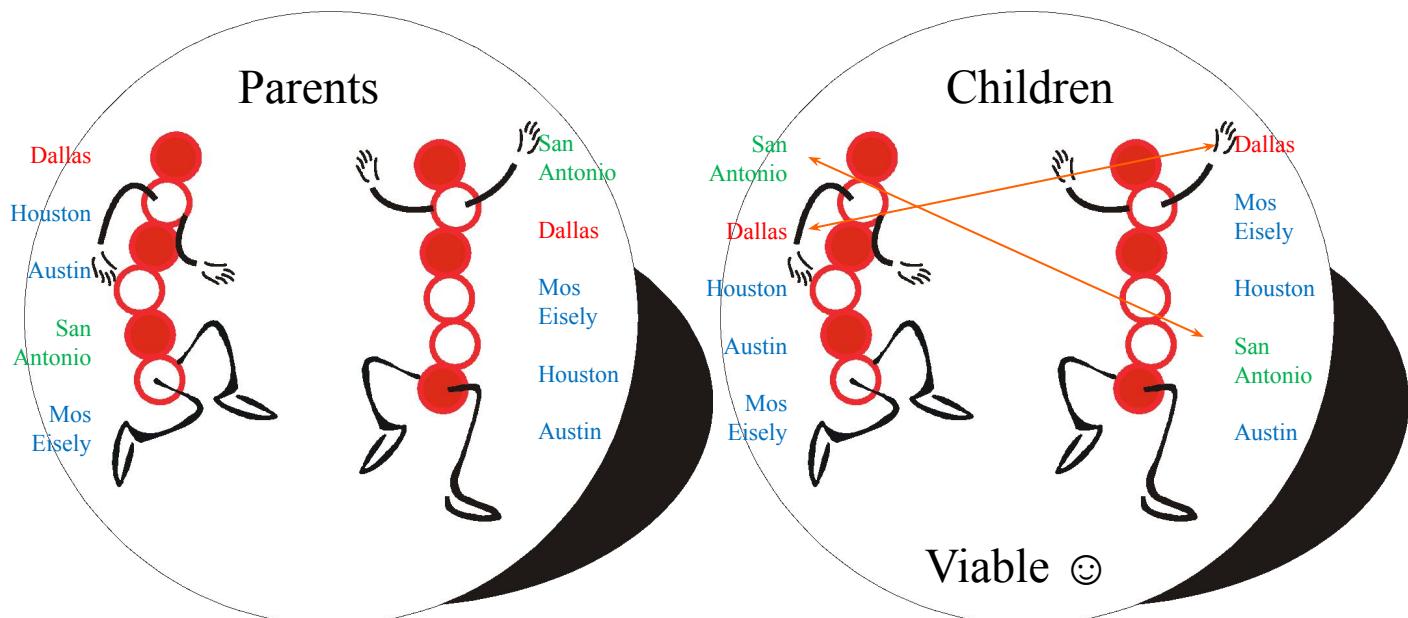
Crossover Bounds (Houston we have a problem)

- Not all crossed pairs are viable. **We can only visit a city once.**
- Different GA problems may have different bounds.



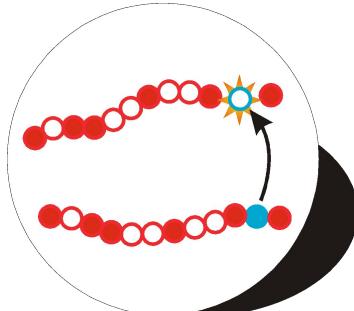
TSP needs some special rules for crossover

- Many GA problems also need special crossover rules.
- Since each genetic sequence contains all the cities in the travel, crossover is a swapping of travel order.
- Remember that crossover also needs to be **efficient**.

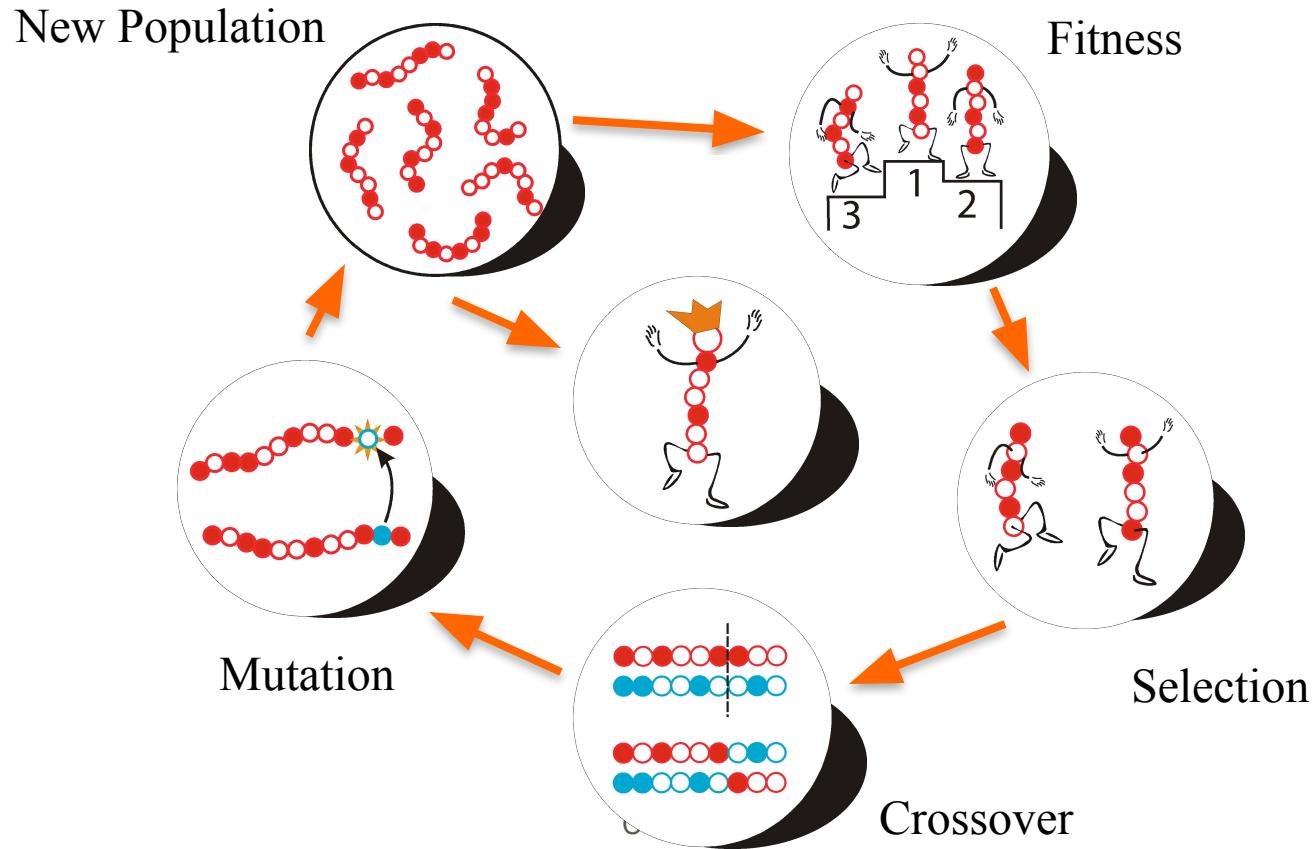


What about local extrema?

- With just crossover breeding, we are constrained to gene sequences which are a cross product of our current population.
- Introduce random effects into our population.
 - Mutation** – Randomly twiddle the genes with some probability.
 - Cataclysm** – Kill off n% of your population and create fresh new salespeople if it looks like you are reaching a local minimum.
 - Annealing of Mating Pairs** – Accept the mating of suboptimal pairs with some probability.
 - Etc...

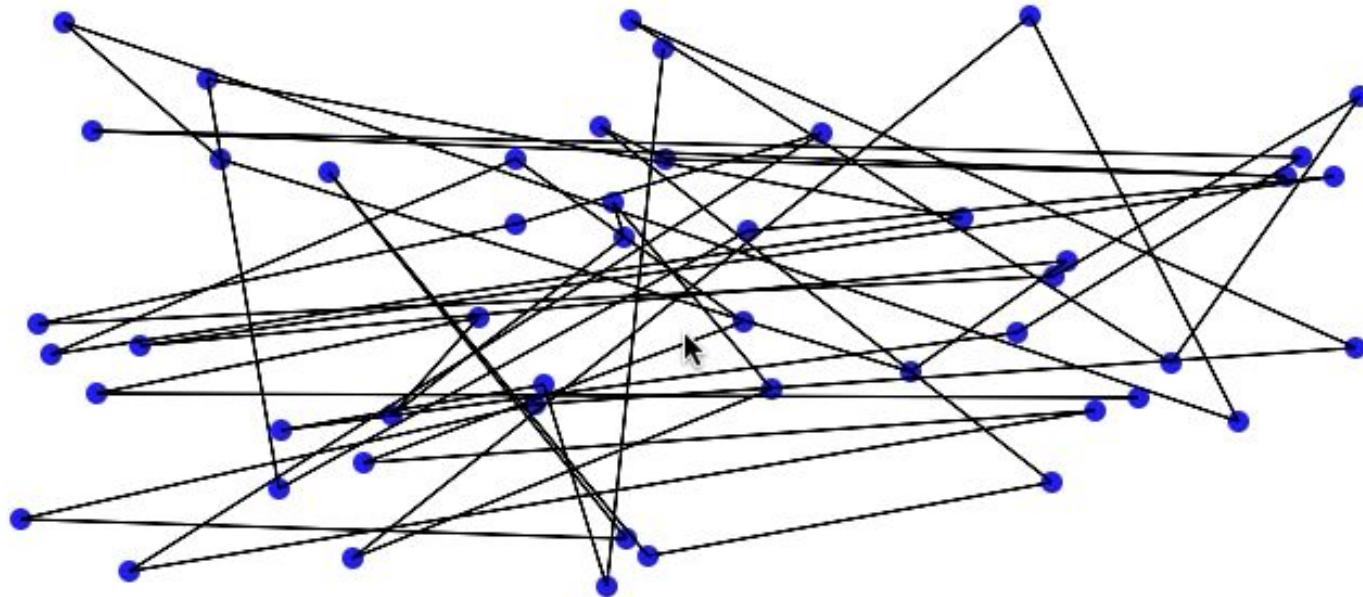


In summation: The GA Cycle



Demo

AIFH Volume 2, Chapter 9: Traveling Salesman (TSP): Genetic Algorithm



Cities: , Stop after stable iterations.

Population: , Mutation %: , % to Mate: , Eligible Pop %:

Ready.

GA and TSP: the claims

- Can solve for over 3500 cities (still took over 1 CPU years).
 - Maybe holds the record.
- Will get within 2% of the optimal solution.
 - This means that the result is **not a best solution per se** but it is an **approximation**.



GA Discussion

- We can apply the GA solution to any problem where we can represent the problems solution (even very abstractly) as a string.
- We can create strings of:
 - Digits
 - Labels
 - Pointers
 - **Code Blocks** – This creates new programs from strung together blocks of code. The key is to make sure the code can run.
 - **Whole Programs** – Modules or complete programs can be strung together in a series. We can also re-arrange the linkages between programs.
- The last two are examples of **Genetic Programming**

Things to consider

- How large is your population?
 - A large population will take more time to run (you have to test each member for fitness!).
 - A large population will cover more bases at once.
- How do you select your initial population?
 - You might create a population of approximate solutions. However, some approximations might start you in the wrong position with too much bias.
- How will you cross bread your population?
 - You want to cross bread and select for your best specimens.
 - Too strict: You will tend towards local minima
 - Too lax: Your problem will converge slower
- How will you mutate your population?
 - Too little: your problem will tend to get stuck in local minima
 - Too much: your population will fill with noise and not settle.

GA is a good *no clue* approach to problem solving

- GA is superb if:
 - Your space is loaded with lots of weird bumps and local minima.
 - GA tends to spread out and test a larger subset of your space than many other types of learning/optimization algorithms.
 - You don't quite understand the underlying *process* of your problem space.
 - NO I DONT: What makes the stock market work??? Don't know? Me neither! Stock market prediction might thus be good for a GA.
 - YES I DO: Want to make a program to predict people's height from personality factors? This might be a Gaussian process and a good candidate for statistical methods which are more efficient.
 - You have lots of processors
 - GA's parallelize very easily!

Why not use GA?

- Creating generations of samples and cross breeding them can be resource intensive.
 - Some problems may be better solved by a general gradient descent method which uses less resource.
 - However, resource-wise, GA is still quite efficient (no computation of derivatives, etc).
- In general if you know the mathematics, shape or underlying process of your problem space, there may be a better solution designed for your specific need.
 - Consider Kernel Based Learning and Support Vector Machines?
 - Consider Neural Networks?
 - Consider Traditional Polynomial Time Algorithms?
 - Etc.

Summary

- Best-first search = general search, where the minimum-cost nodes (according to some measure) are expanded first.
- Greedy search = best-first with the estimated cost to reach the goal as a heuristic measure.
 - Generally faster than uninformed search
 - not optimal
 - not complete.
- A* search = best-first with measure = path cost so far + estimated path cost to goal.
 - combines advantages of uniform-cost and greedy searches
 - complete, optimal and optimally efficient
 - space complexity still exponential
- Hill climbing and simulated annealing: iteratively improve on current state
 - lowest space complexity, just $O(1)$
 - risk of getting stuck in local extrema (unless following proper simulated annealing schedule)
- Genetic algorithms: parallelize the search problem

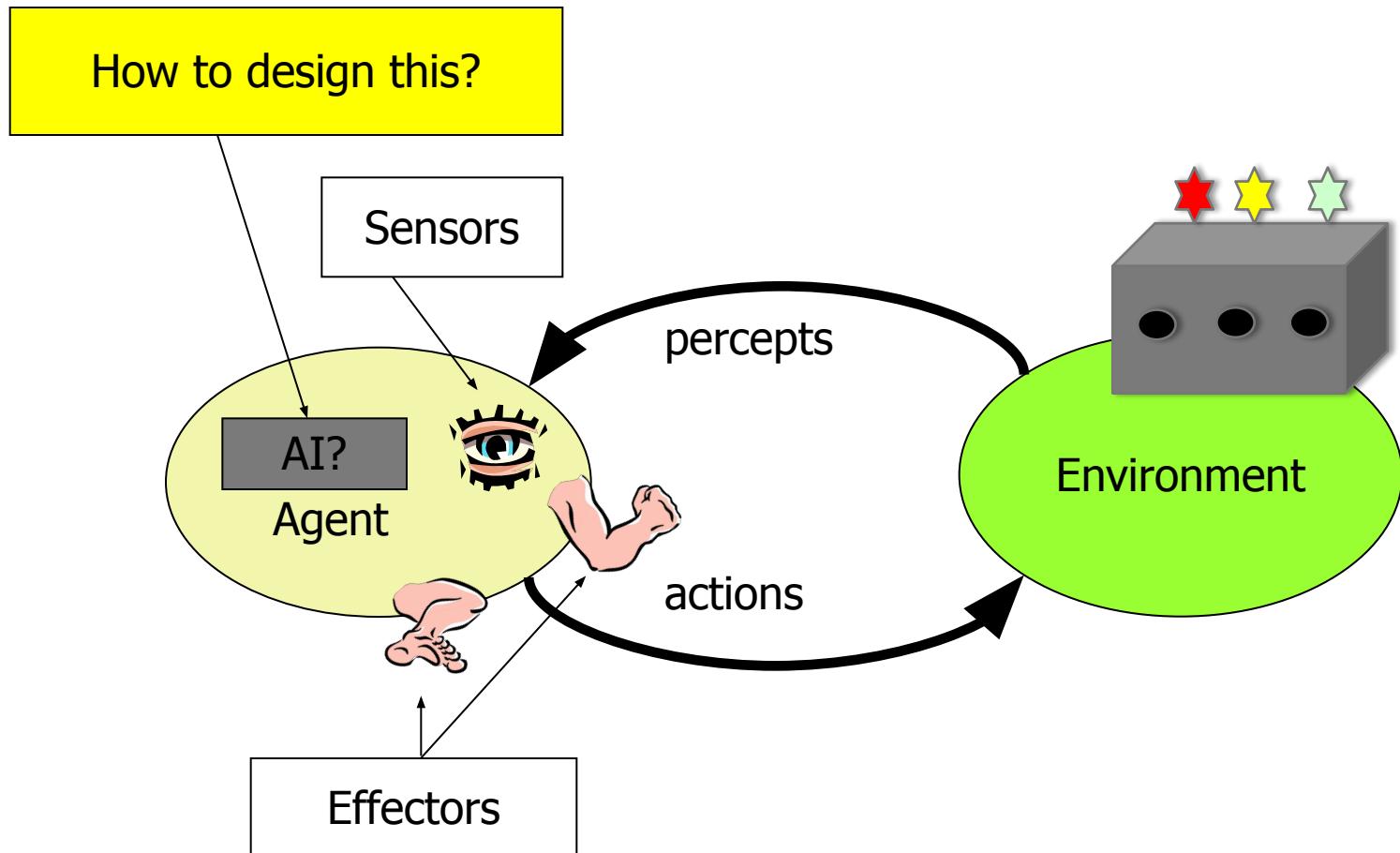
CSCI 561 - Foundation for Artificial Intelligence

DISCUSSION SECTION (WEEK 2)

PROF WEI-MIN SHEN

WHAT IS “PROBLEM SOLVING”?

WHAT IS “SEARCH”?

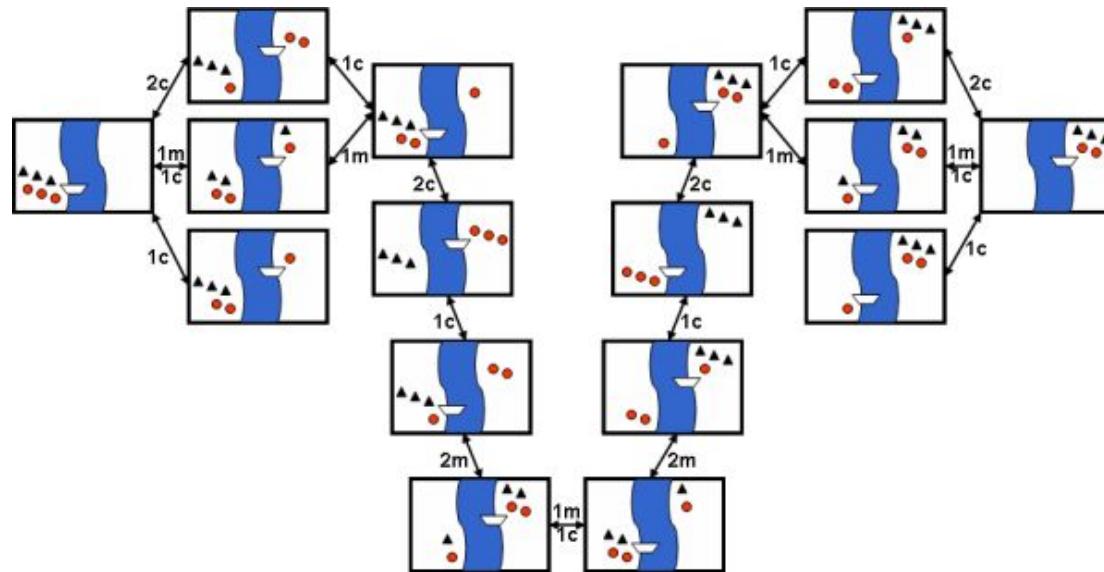


ESSENTIALS OF SEARCH

- **How to represent a “problem”?**
 - How to construct a Search Tree/Graph?
 - Nodes, Goals, Initials, Links
- **How to find a solution “systematically” or “optimally” in your representation?**
 - Use the uninformed algorithms you learned
 - Use the informed algorithms you learned

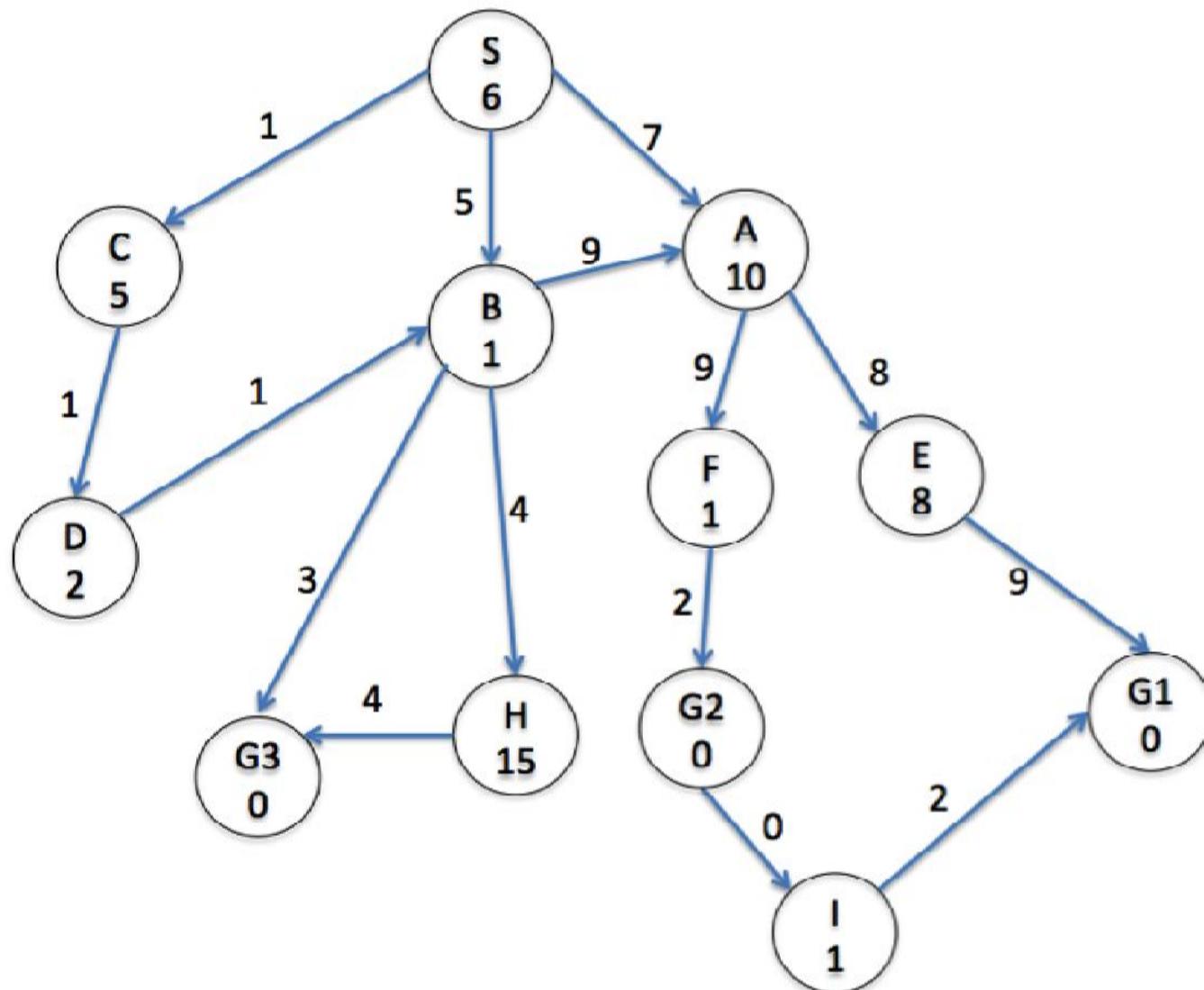
MISSIONARIES AND CANNIBALS

Did you find that there was much search involved in finding a solution?



Why do people have a hard time solving this problem?

SEARCH GRAPH



GRAPH SEARCH

```
function GRAPH-SEARCH(problem) return a solution or failure
  frontier  $\leftarrow$  MAKE-QUEUE(MAKE-NODE(problem.INITIAL-STATE))
  explored_set  $\leftarrow$  empty
  loop do
    if EMPTY?(frontier) then return failure
    node  $\leftarrow$  REMOVE-FIRST(frontier)
    if problem.GOAL-TEST applied to node.STATE succeeds
      then return SOLUTION(node)
    explored_set  $\leftarrow$  INSERT(node, explored_set)
    for each new_node in EXPAND(node, problem) do
      if NOT(MEMBER?(new_node, frontier)) and
         NOT(MEMBER?(new_node, explored_set))
      then frontier  $\leftarrow$  INSERT(new_node, frontier)
```

GRAPH SEARCH

```
function GRAPH-SEARCH(problem) return a solution or failure
  frontier  $\leftarrow$  MAKE-QUEUE(MAKE-NODE(problem.INITIAL-STATE))
  explored_set  $\leftarrow$  empty
  loop do
    if EMPTY?(frontier) then return failure
    node  $\leftarrow$  REMOVE-FIRST(frontier)
    if problem.GOAL-TEST applied to node.STATE succeeds
      then return SOLUTION(node)
    explored_set  $\leftarrow$  INSERT(node, explored_set)
    for each new_node in EXPAND(node, problem) do
      if NOT(MEMBER?(new_node, frontier)) and
         NOT(MEMBER?(new_node, explored_set))
      then frontier  $\leftarrow$  INSERT(new_node, frontier)
```

How to modify this algorithm to become
the following algorithms? (important!)

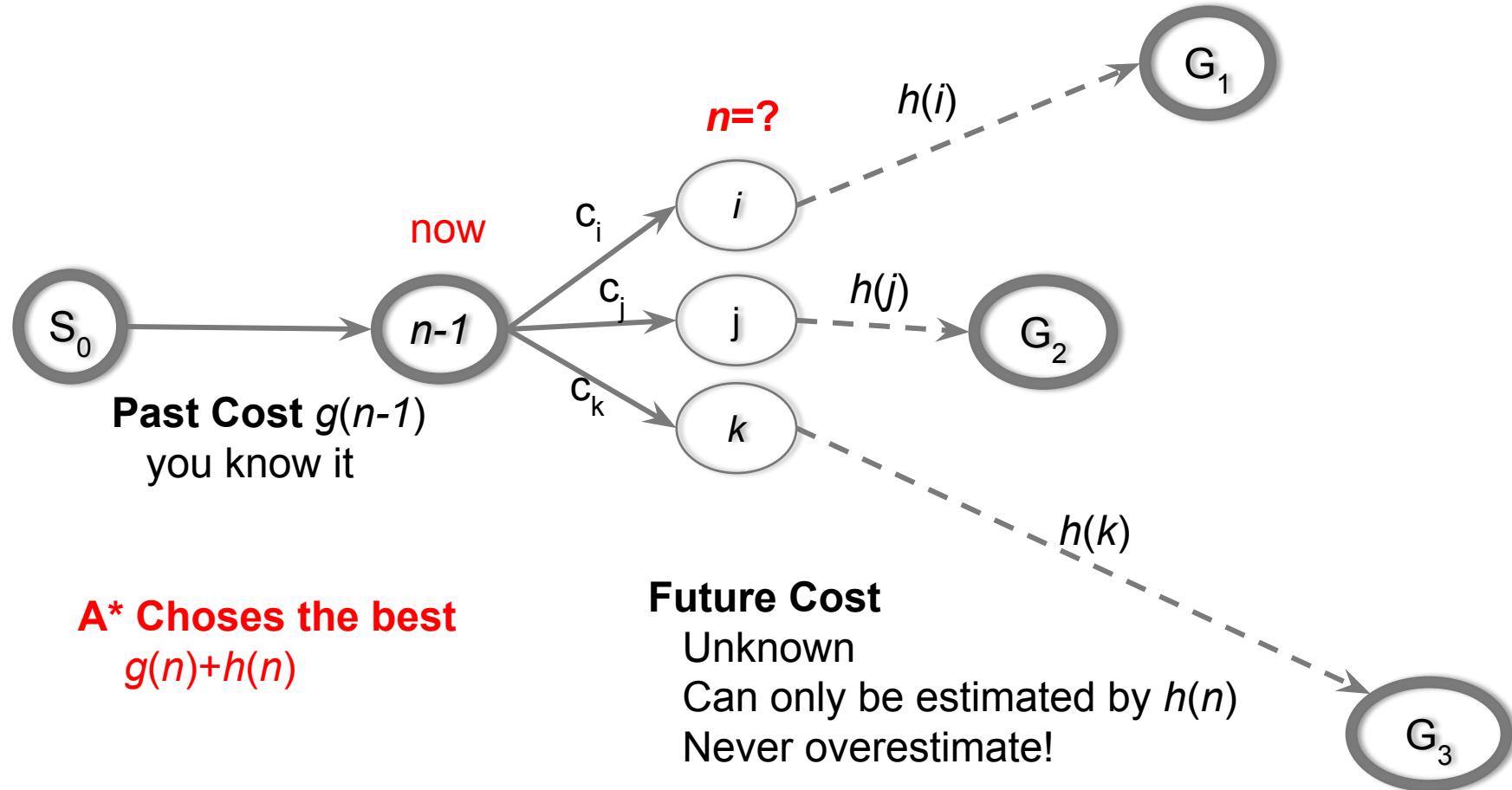
BFS

DFS

UCS

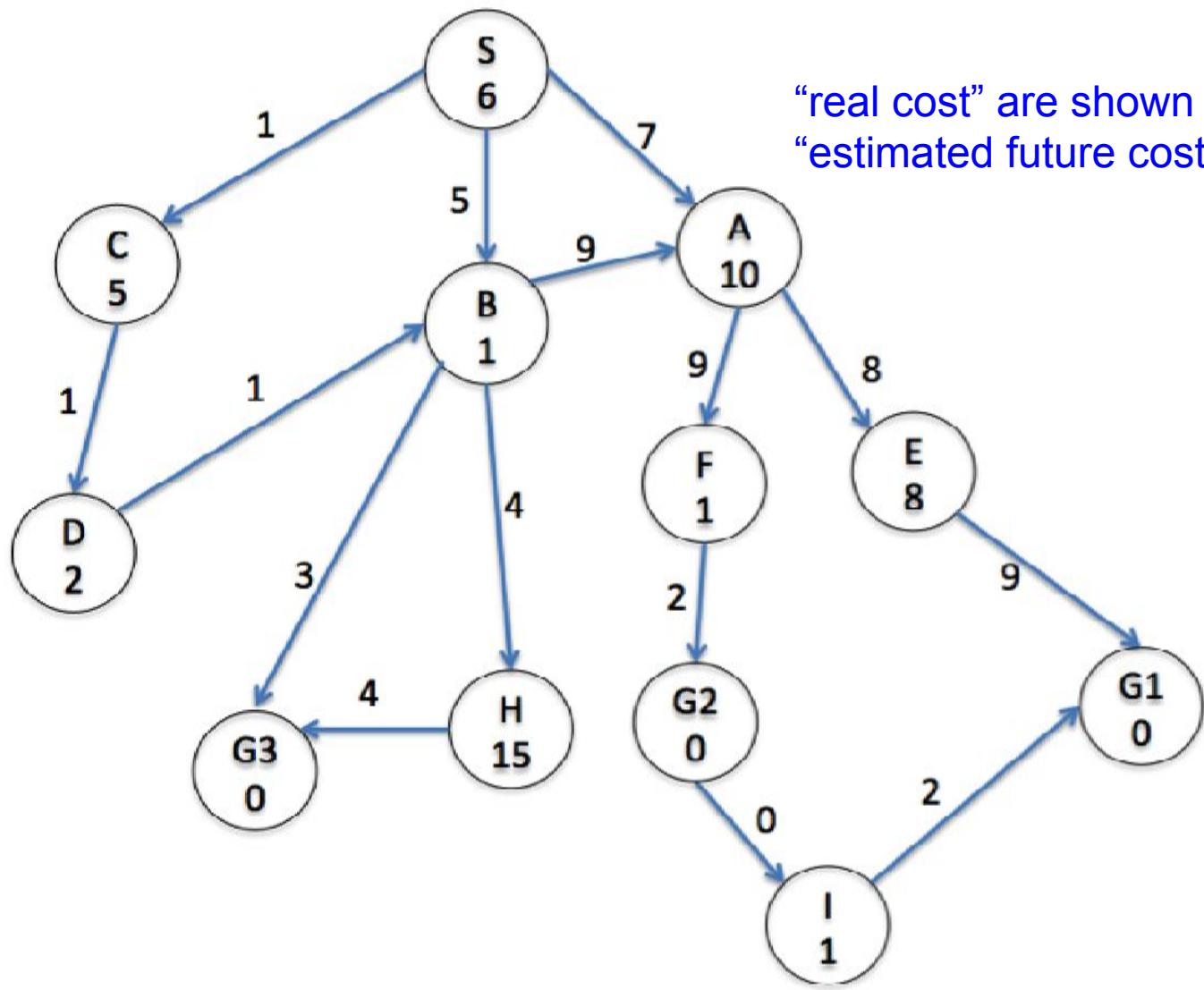
A*

A* = BEST-FIRST (PAST + ESTIMATED FUTURE)



Note: Uniform-cost search uses only $g(n)$, no $h(n)$.

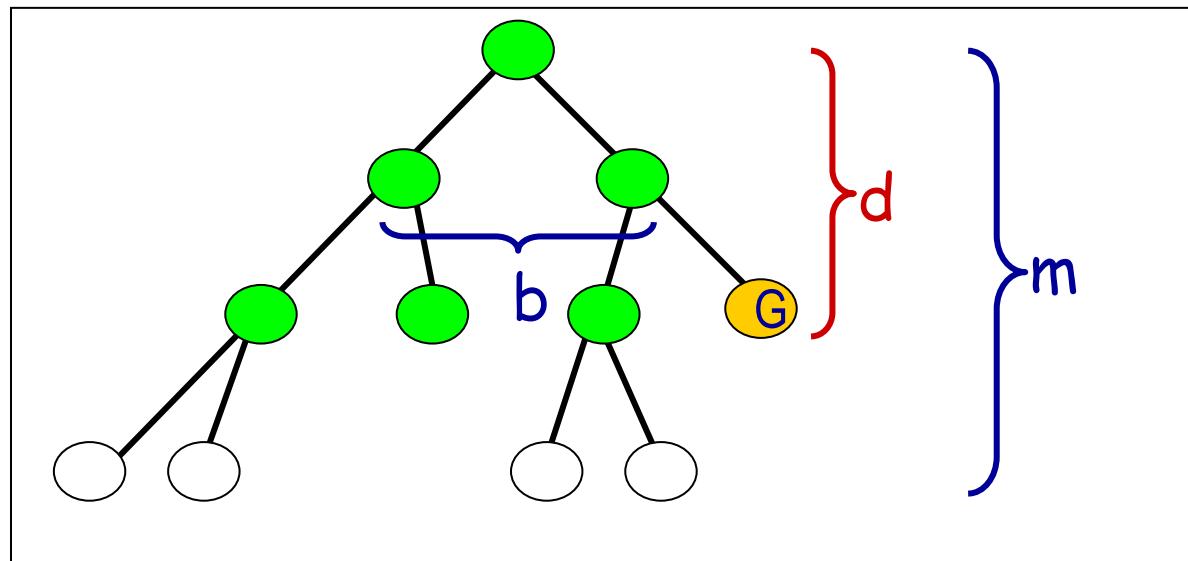
Is it good to have $h(x)=0$ for all x ?



“real cost” are shown on edge
 “estimated future cost” are inside circle

TIME COMPLEXITY OF BREADTH-FIRST SEARCH

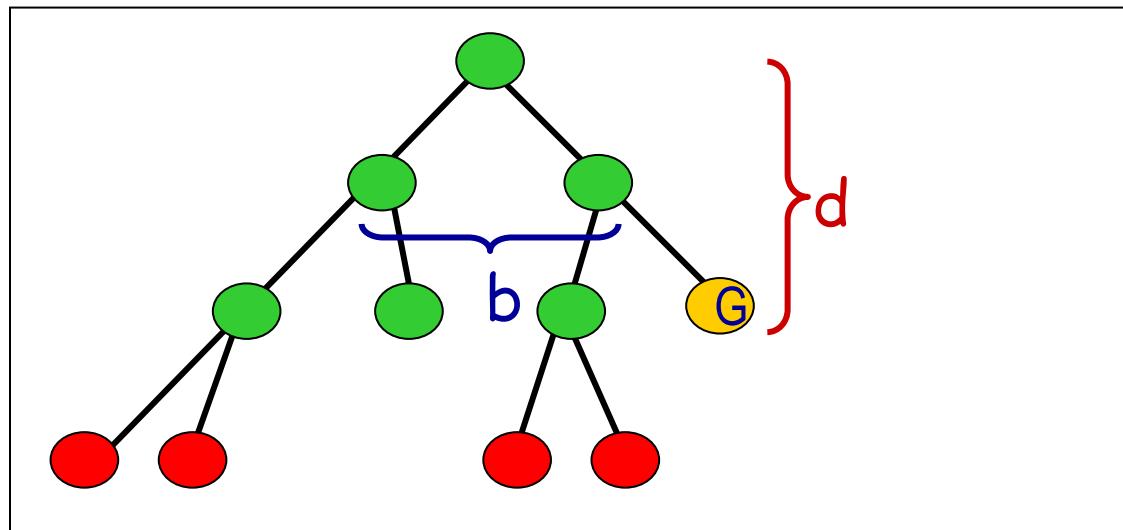
- Illustrates when goal check is done when node **is selected for expansion**
- If a goal node is found on depth **d** of the tree, all nodes up till that depth are created and examined (note: and the children of nodes at depth d are created and queued, but not yet examined).



- Thus: $O(b^{d+1})$

SPACE COMPLEXITY OF BREADTH-FIRST SEARCH

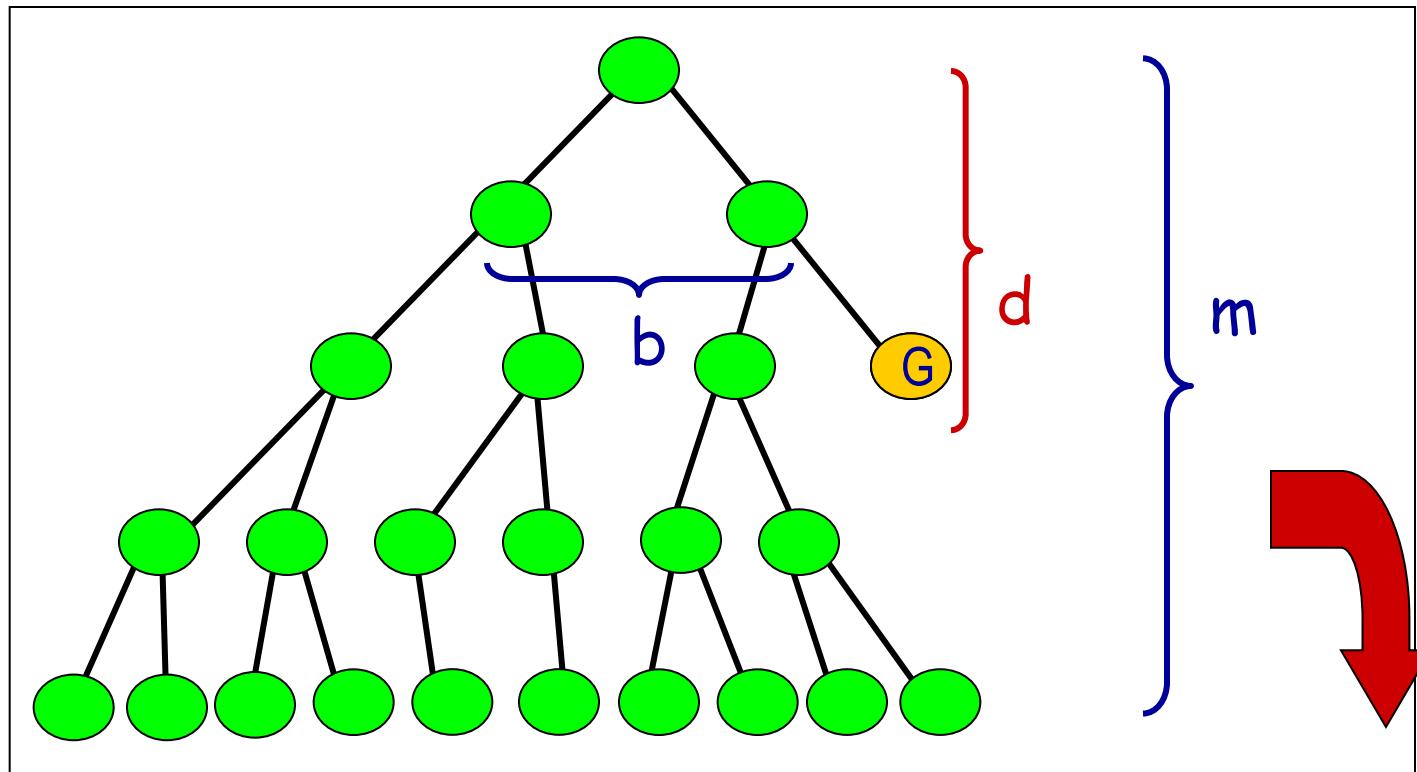
- Illustrates when goal check is done when node **is selected for expansion**
- Largest number of nodes in FRONTIER is reached on the level $d+1$ just beyond the goal node.



- QUEUE contains all nodes. (Thus: 4) .
- In General: $b^{d+1} - b \sim b^{d+1}$

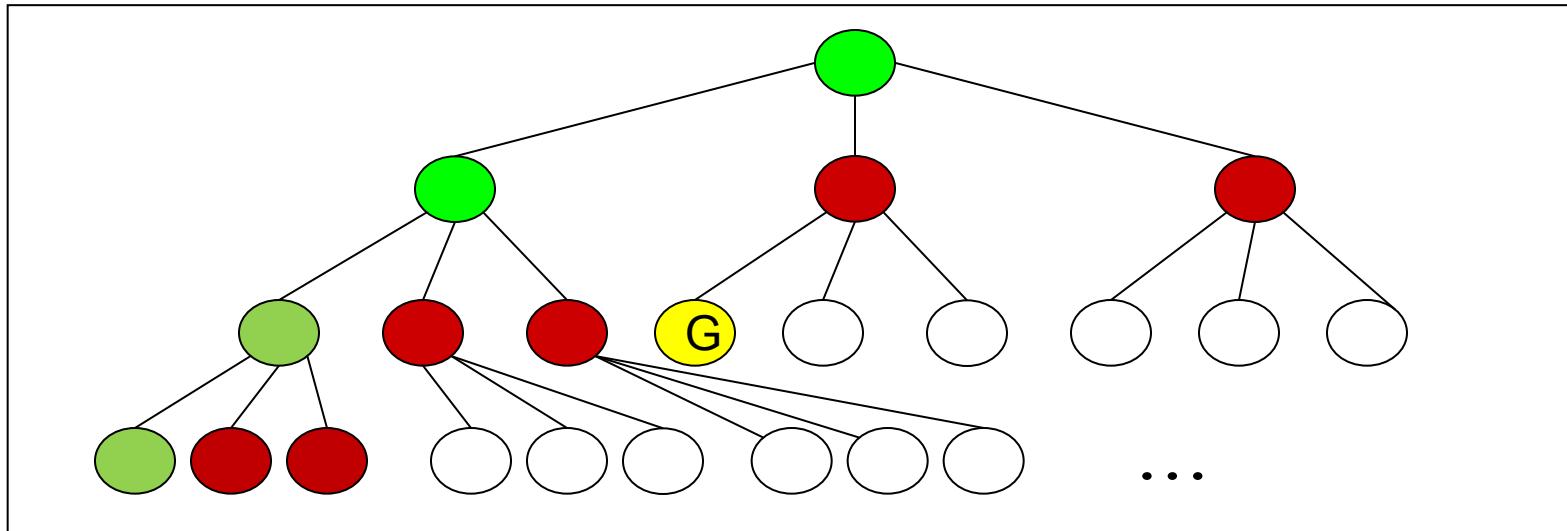
TIME COMPLEXITY OF DEPTH-FIRST SEARCH

- In the worst case:
 - the (only) goal node may be on the right-most branch,



SPACE COMPLEXITY OF DEPTH-FIRST

- Largest number of nodes in FRONTIER is reached in bottom left-most node.
- Example: $m = 3, b = 3$:



- FRONTIER contains all nodes. Thus: 6.
- In General FRONTIER contains $((b-1) * m)$
- Order: $O(m*b)$

SEARCH IN AI APPLICATIONS

Is search involved in these AI applications? If so, in what part or parts of the application?

- Building a driverless car that will drive down a roadway. (Leave aside the search involved in route planning.)
- Building a system like Siri.
- Text-to-speech synthesis

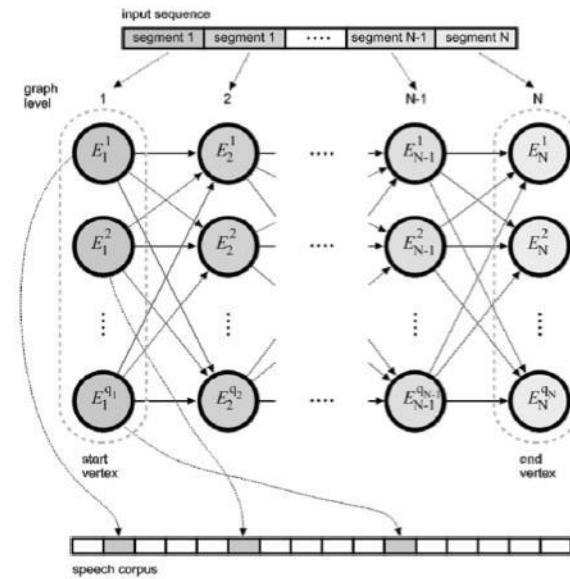
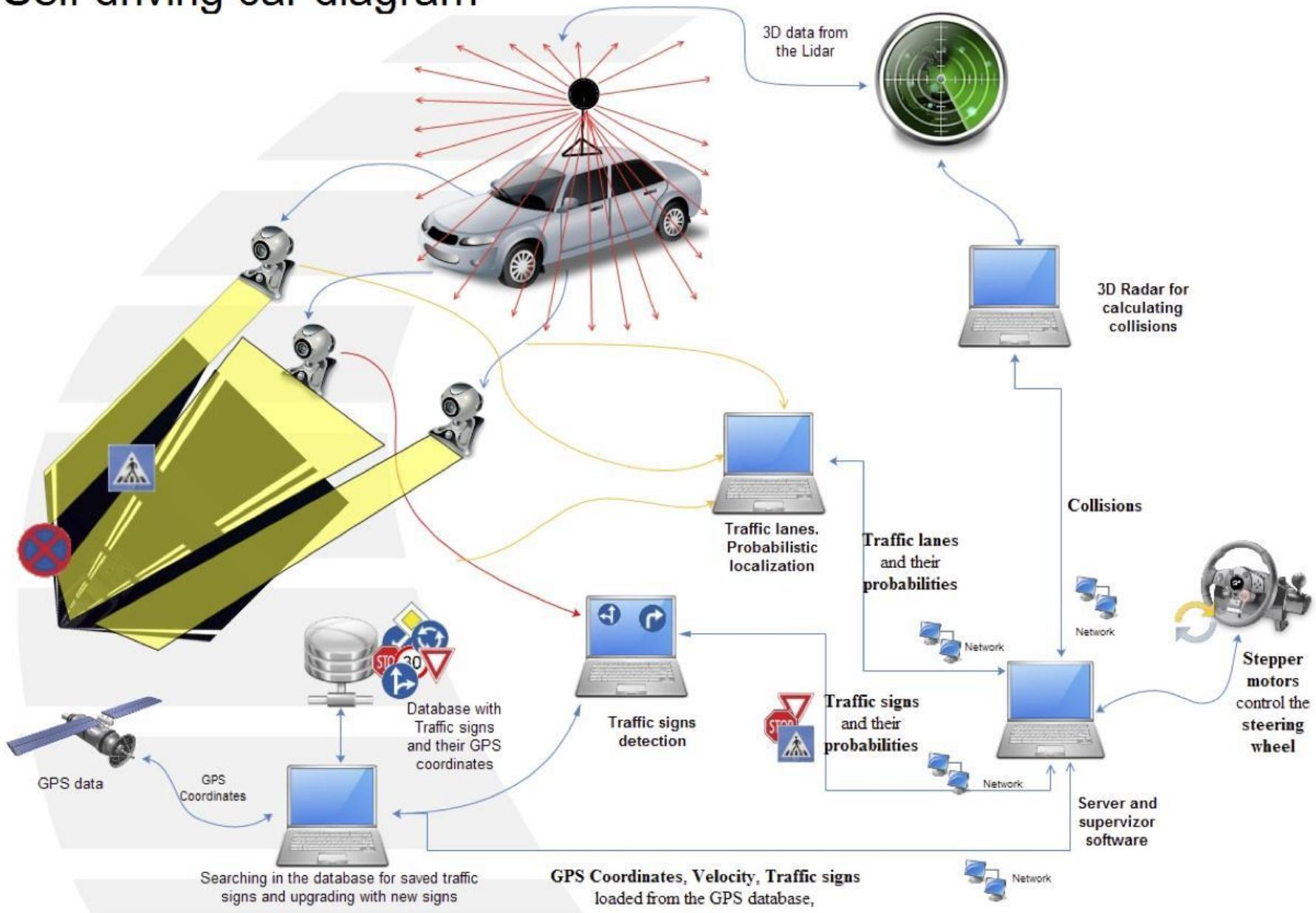


Figure 1. Structure of the graph for finding the optimal speech unit sequence; E_1^i are the graph initial-level vertices, E_N^i are the graph final-level vertices.

Self driving car diagram



WHAT YOU SHOULD KNOW

- What is the difference between uninformed and informed search? Which ones are optimal?
- What are the advantages and disadvantages of depth-first search?
- Be familiar with the differences between search strategies shown in Figure 3.21

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes ^a	Yes ^{a,b}	No	No	Yes ^a	Yes ^{a,d}
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes ^c	Yes	No	No	Yes ^c	Yes ^{c,d}

WANT MORE?

BigO and complexity:

<https://apelbaum.wordpress.com/2011/05/05/big-o/>

CSCI 561 - Foundation for Artificial Intelligence

05 -- Constraint Satisfaction

06 -- Game Playing

wk3 -- Discussions

Professor Wei-Min Shen
University of Southern California

Constraint Satisfaction

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

Constraint satisfaction problems

Standard search problem:

- state is a “black box” – any data structure that supports successor function (actions), heuristic evaluation function, and goal tests

CSP:

- State: is defined by variables X_i with values from domains D_i
- Goal Test: 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 problem



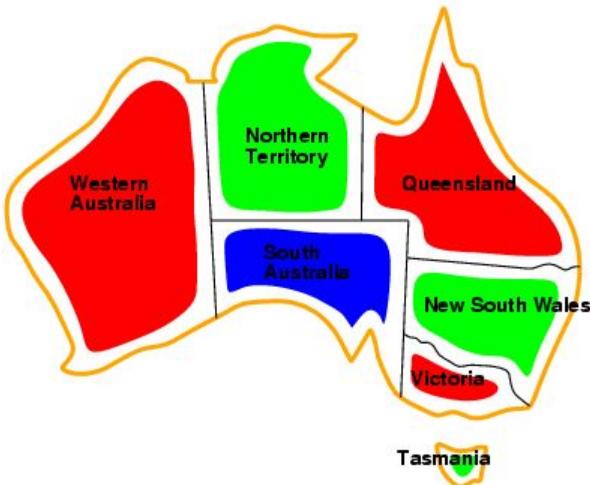
- **Variables:** WA, NT, Q, NSW, V, SA, T
- **Domains:** $D_i = \{\text{red, green, blue}\}$ (one for each variable)
- **Constraints:** $C_i = <\text{scope, rel}>$ where *scope* is a tuple of variables and *rel* is the relation over the values of these variables
 - E.g., here, adjacent regions must have different colors
e.g., $WA \neq NT$, or $(WA,NT) \in \{(red,green), (red,blue), (green,red), (green,blue), (blue,red), (blue,green)\}$

Example: A Map Coloring Problem



- **Assignment:** values are given to some or all variables
- **Consistent (legal) assignment:** assigned values do not violate any constraint
- **Complete assignment:** every variable is assigned a value
- **Solution to a CSP:** a consistent and complete assignment

Example: A Map Coloring Problem



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

Demo

USA Mainland Coloring using CCM (Chemical Casting Model) -- ver 1.13, 1996-11-23 --

Stop

Restart

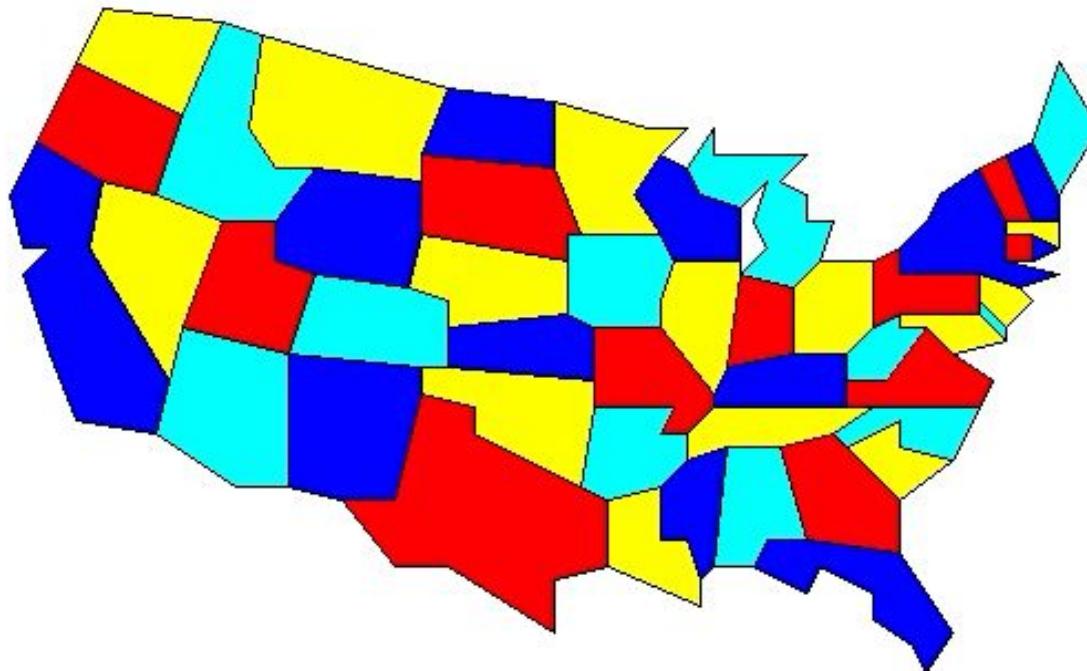
Frustration ON

or options:

Medium speed (20 reactions/sec)

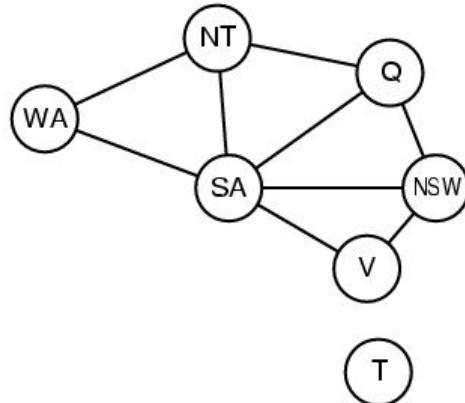
Variable catalyst rule

Stopped. #tests = 2002 #reactions = 100 MOD = 1.0 Time = 5.374



Constraint Graph

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



Varieties of CSPs

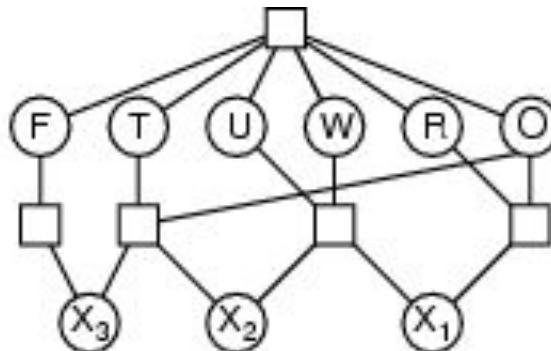
- Discrete variables
 - Finite domains:
 - n variables, domain size $d \square O(d^n)$ complete assignments
 - e.g., Boolean CSPs, including Boolean Satisfiability (NP-complete)
 - Infinite domains:
 - integers, strings, etc.
 - e.g., job scheduling, variables are start/end days for each job
 - need 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

Constraint Types and Varieties

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

Example: Cryptarithmetic

$$\begin{array}{r} & & 1 \\ & T & W & O \\ + & T & W & O \\ \hline F & O & U & R \end{array}$$



- 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:
 - $\text{Alldiff}(F, T, U, W, R, O)$
 - $O + O = R + 10 \cdot X_1$
 - $X_1 + W + W = U + 10 \cdot X_2$
 - $X_2 + T + T = O + 10 \cdot X_3$
 - $X_3 = F, T \neq 0, F \neq 0$

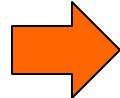
Constraint Hypergraph
Circles: nodes for variable
Squares: hypernodes for n-ary constraints

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

Example: Sudoku

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



?

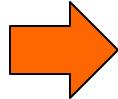
Variables: each square (81 variables)

Domains: [1 .. 9]

Constraints: each column, each row, and each of the nine 3×3 sub-grids that compose the grid contain all of the digits from 1 to 9

Example: Sudoku

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



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

Variables: each square (81 variables)

Domains: [1 .. 9]

Constraints: each column, each row, and each of the nine 3×3 sub-grids that compose the grid contain all of the digits from 1 to 9

Solving CSP Using the “Standard” Search Approach

Let's start with the straightforward approach, then improve 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 be discarded
 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
branch b = d, depth of the search tree is n, 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

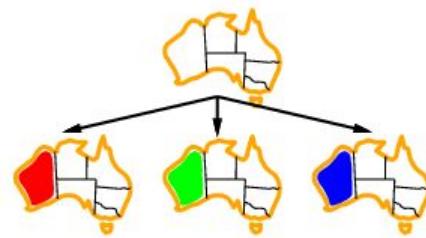
(note: textbook has a slightly more complex version)

```
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
```

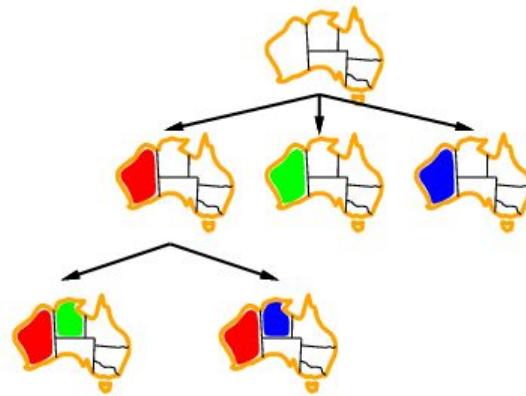
Backtracking example



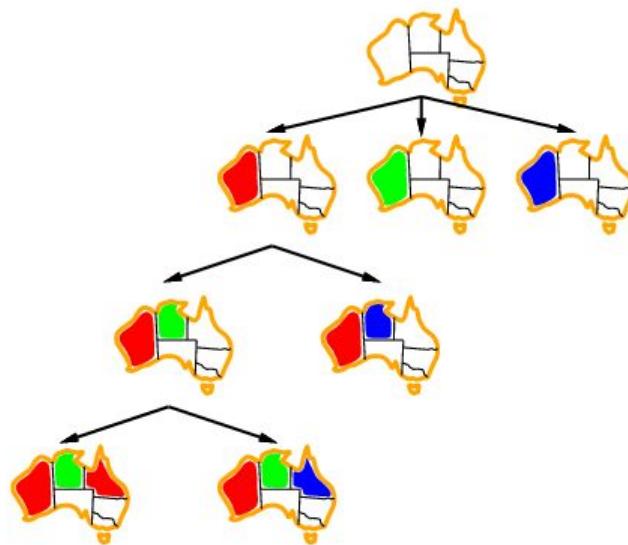
Backtracking example



Backtracking example



Backtracking example

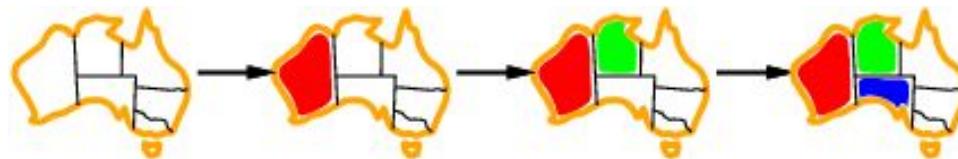


Improving backtracking efficiency

- General-purpose methods can give huge gains in speed (like using heuristics in informed search):
 - Which variable should be assigned next?
 - In what order should its values be tried?
 - Can we detect inevitable failure early?

Selecting the Most Constrained Variable

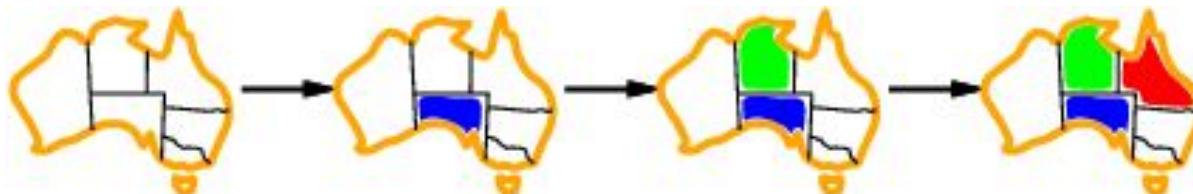
- Most constrained variable:
choose the variable with the fewest legal values



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

Most constraining variable

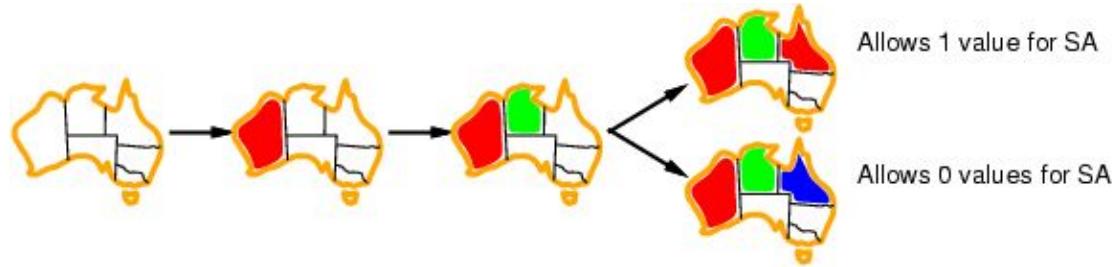
- Tie-breaker among most constrained variables
- Most constraining variable:
 - choose the variable with the most constraints on remaining variables



- also known as the **degree heuristic**

Least constraining value

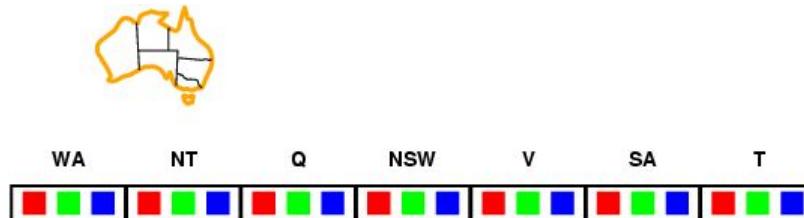
- Given a variable, choose the least constraining value:
 - the one that rules out the fewest values in the remaining variables



- Combining these heuristics makes 1000 queens feasible

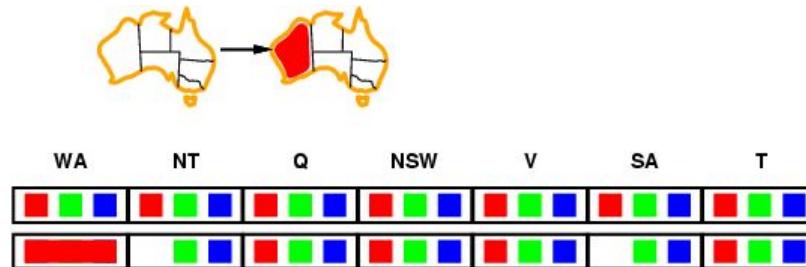
Forward Checking (to detect failures as early as possible)

- Idea:
 - Keep track of remaining legal values for unassigned variables (inference step)
 - Terminate search when any variable has no legal values



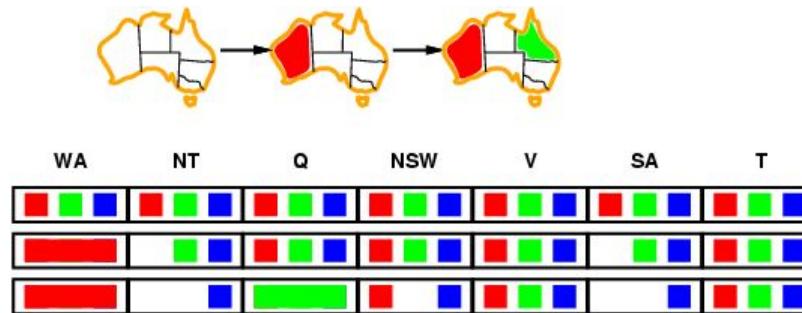
Forward checking

- Idea:
 - Keep track of remaining legal values for unassigned variables (inference step)
 - Terminate search when any variable has no legal values



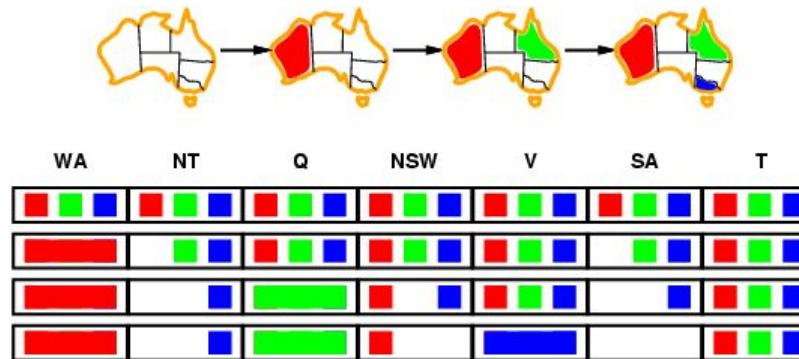
Forward checking

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



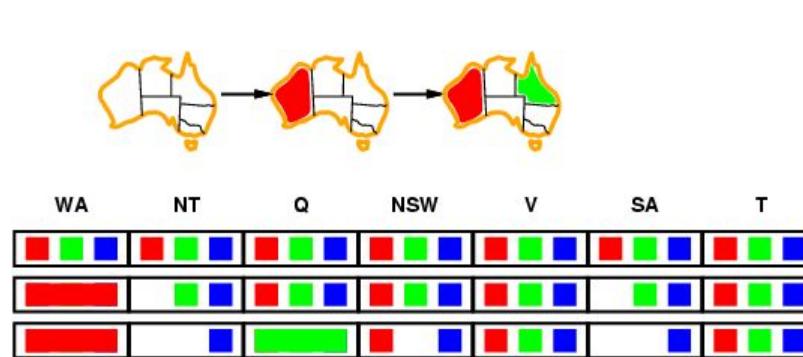
Forward checking

- 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



Node and Arc Consistency

A single variable is **node-consistent** if all the values in its domain satisfy the variable's unary constraints

A variable is **arc-consistent** if every value in its domain satisfies the binary constraints

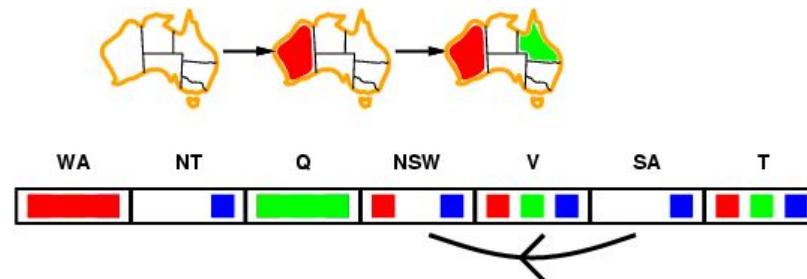
- i.e., X_i arc-consistent with X_j if for every value in D_i there exists a value in D_j that satisfies the binary constraints on arc (X_i, X_j)

A **network is arc-consistent** if every variable is arc-consistent with every other variable.

Arc-consistency algorithms: reduce domains of some variables to achieve network arc-consistency.

Arc Consistency

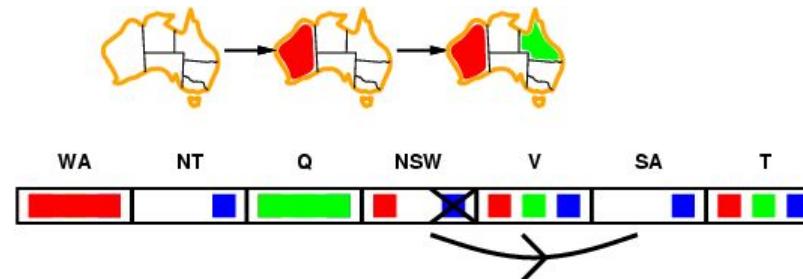
- Simplest form of propagation makes each arc **consistent**
- $X \rightarrow Y$ is consistent iff
 - for **every** value x of X there is **some** allowed value y for Y



In this case, SA is indeed Arc-consistent with NSW

Arc consistency

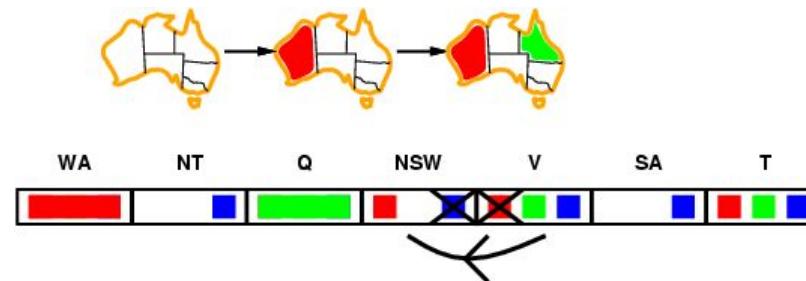
- Simplest form of propagation makes each arc **consistent**
- $X \rightarrow Y$ is consistent iff
 - for **every** value x of X there is **some** allowed value y for Y



In this case, NSW is Not Arc-consistent with SA

Arc consistency

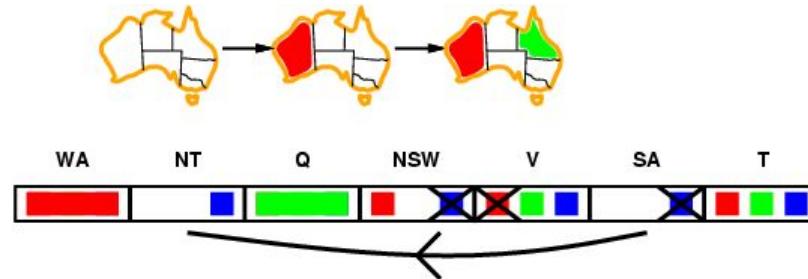
- Simplest form of propagation makes each arc **consistent**
- $X \rightarrow Y$ is consistent iff
 - for **every** value x of X there is **some** allowed value y for Y



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

Arc consistency

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



- If X loses a value, neighbors of X need to be rechecked
- Arc consistency detects failure earlier than forward checking
- After running AC-3, either every arc is arc-consistent or some variable has empty domain, indicating the CSP cannot be solved.
- Can be run as a preprocessor or after each assignment

Arc Consistency Algorithm: AC-3

- Start with a queue that contains all arcs
- Pop one arc (X_i, X_j) and make X_i arc-consistent with respect to X_j
 - If D_i was not changed, continue to next arc,
 - Otherwise, D_i was **revised** (domain was reduced), so need to check all arcs connected to X_i again: add all connected arcs (X_k, X_i) to the queue. (this is because the reduction in D_i may yield further reductions in D_k)
 - If D_i is revised to empty, then the CSP problem has no solution.

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
     $(X_i, X_j) \leftarrow \text{REMOVE-FIRST}(\textit{queue})$ 
    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: ? (n variables, d values)

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
     $(X_i, X_j) \leftarrow \text{REMOVE-FIRST}(\textit{queue})$ 
    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: $O(n^2d^3)$ (n variables, d values)
- (each arc can be queued only d times, n^2 arcs (at most), checking one arc is $O(d^2)$)

Demo

CSP Applet Version 4.6.1 --- simple1.xml

File Edit View CSP Options Help

Create Variable Create Constraint Add Variable to Constraint Select Delete Set Properties

Create Solve

Click the canvas to create a variable.

```
graph LR; A((A: {1, 2, 3, 4})) --- AB[A < B]; AB --- B((B: {1, 2, 3, 4})); B --- BC[B < C]; BC --- C((C: {1, 2, 3, 4}))
```

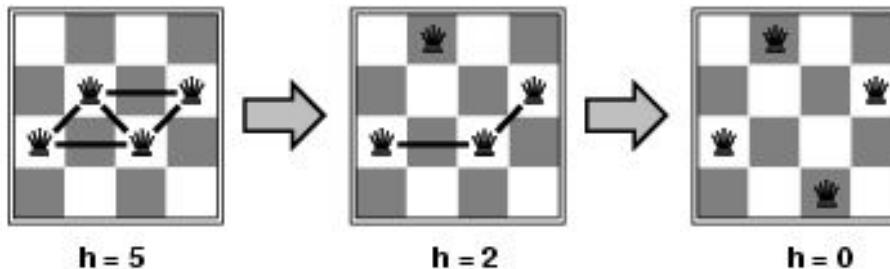
The diagram illustrates a Constraint Satisfaction Problem (CSP) with three variables: A, B, and C. Each variable is represented by a circle containing its domain: {1, 2, 3, 4}. The variables are connected by constraints: A is less than B, and B is less than C. These constraints are represented by rectangular boxes labeled "A < B" and "B < C" respectively, positioned between the nodes for A and B, and between B and C. A cursor is visible above the first constraint box.

Another Approach: 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:
 - choose value that violates the fewest constraints
 - i.e., hill-climb with $h(n) = \text{total number of violated constraints}$

Example: 4-Queens

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



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

Min-Conflicts Algorithm

```
function MIN-CONFLICTS(csp, max-steps) returns a solution or failure
  inputs: csp, a constraint satisfaction problem
           max-steps, the number of steps allowed before giving up

  current  $\leftarrow$  an initial complete assignment for csp
  for i = 1 to max-steps do
    if current is a solution for csp then return current
    var  $\leftarrow$  a randomly chosen conflicted variable from csp.VARIABLES
    value  $\leftarrow$  the value v for var that minimizes CONFLICTS(var, v, current, csp)
    set var = value in current
  return failure
```

Figure 6.8 The MIN-CONFLICTS algorithm for solving CSPs by local search. The initial state may be chosen randomly or by a greedy assignment process that chooses a minimal-conflict value for each variable in turn. The CONFLICTS function counts the number of constraints violated by a particular value, given the rest of the current assignment.

Example: N-Queens

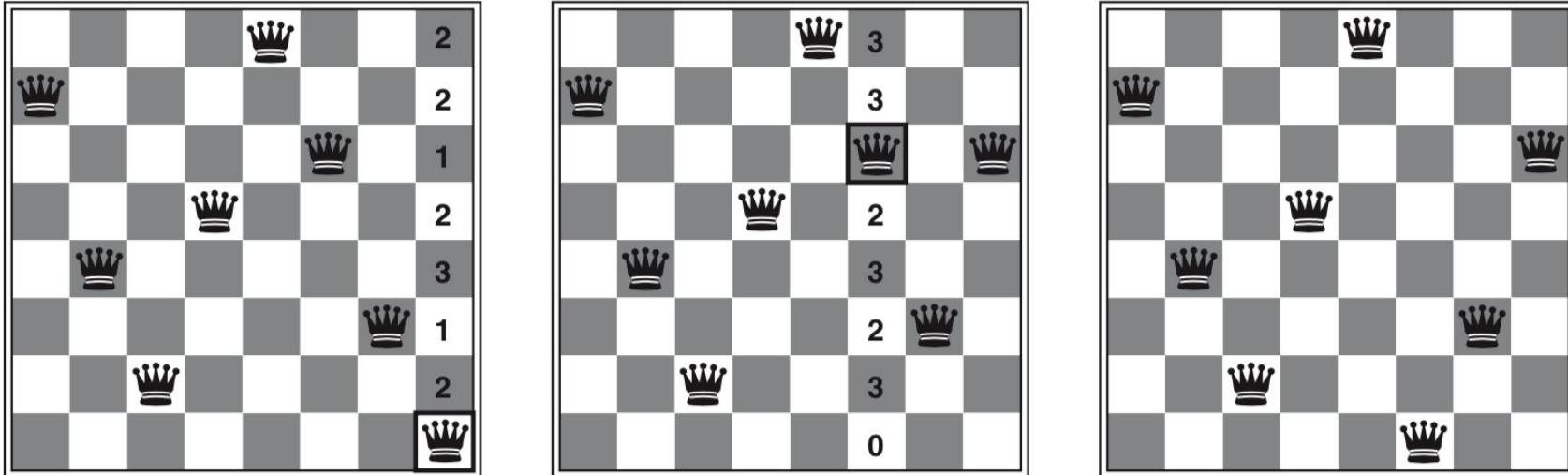


Figure 6.9 A two-step solution using min-conflicts for an 8-queens problem. At each stage, a queen is chosen for reassignment in its column. The number of conflicts (in this case, the number of attacking queens) is shown in each square. The algorithm moves the queen to the min-conflicts square, breaking ties randomly.

Summary

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

Game Playing

- **Game Playing**

- The minimax algorithm
- Resource limitations
- alpha-beta pruning
- Elements of chance



What kind of games?

- **Abstraction:** To describe a game we must capture every relevant aspect of the game. Such as:
 - Chess, Tic-Tac-Toe, others
- **Accessible environments:** Such games are characterized by perfect information and completely observable
- **Search:** game-playing then consists of a search through possible game positions
- **Unpredictable opponent:** introduces **uncertainty** thus game-playing must deal with **contingency problems**

Searching for the Next Move

- **Complexity:** many games have a huge search space
 - **Chess:** $b = 35, m=100 \Rightarrow \text{nodes} = 35^{100}$
if each node takes about 1 ns to explore
then each move will take about **10⁵⁰ millennia**
to calculate.
 - **Resource (e.g., time, memory) limit:** optimal solution not feasible/possible, thus must approximate
1. **Pruning:** makes the search more efficient by discarding portions of the search tree that cannot improve quality result
 2. **Evaluation functions:** heuristics to evaluate utility of a state without exhaustive search

Two-Player Games

- A game formulated as a search problem:
 - State space
 - Initial state: ?
 - Operators: ?
 - Terminal state: ?
 - Utility function: ?

Two-Player Games

- A game formulated as a search problem:

- Initial state: board position and turn
- Operators: definition of legal moves
- Terminal state: conditions for when game is over
- Utility function: a numeric value that describes the outcome of the game. E.g., -1, 0, 1 for loss, draw, win.
(AKA **payoff function**)

Game vs. Search Problem

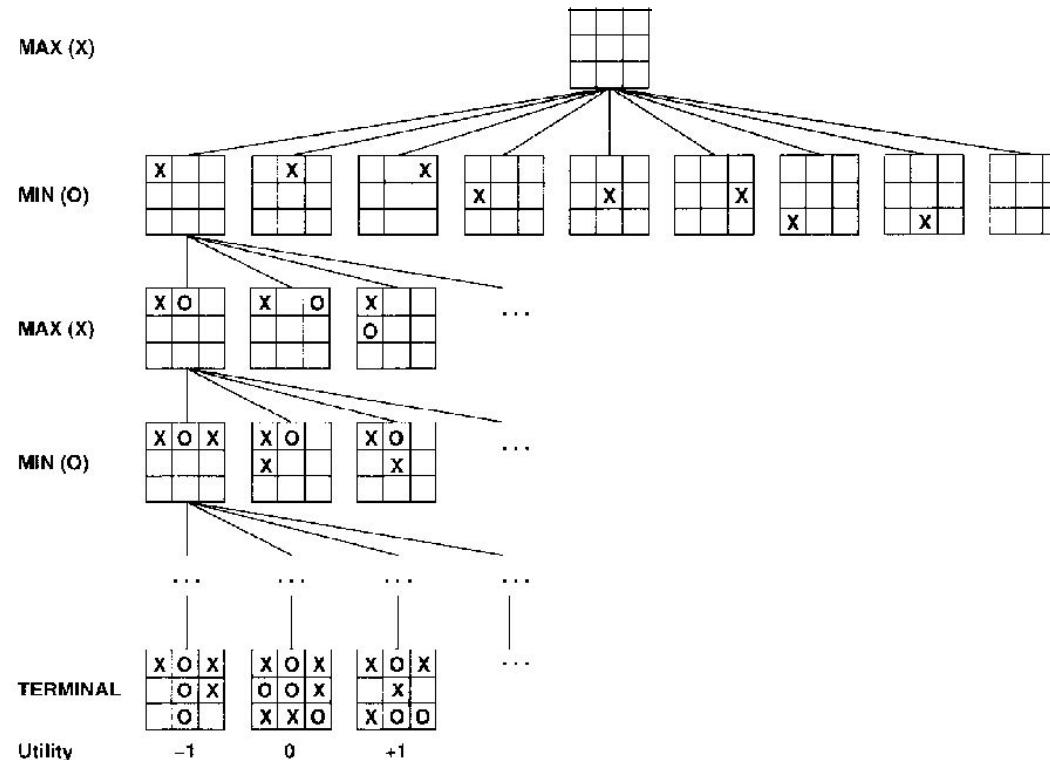
“Unpredictable” opponent \Rightarrow solution is a contingency plan

Time limits \Rightarrow unlikely to find goal, must approximate

Plan of attack:

- algorithm for perfect play (Von Neumann, 1944)
- finite horizon, approximate evaluation (Zuse, 1945; Shannon, 1950; Samuel, 1952–57)
- pruning to reduce costs (McCarthy, 1956)

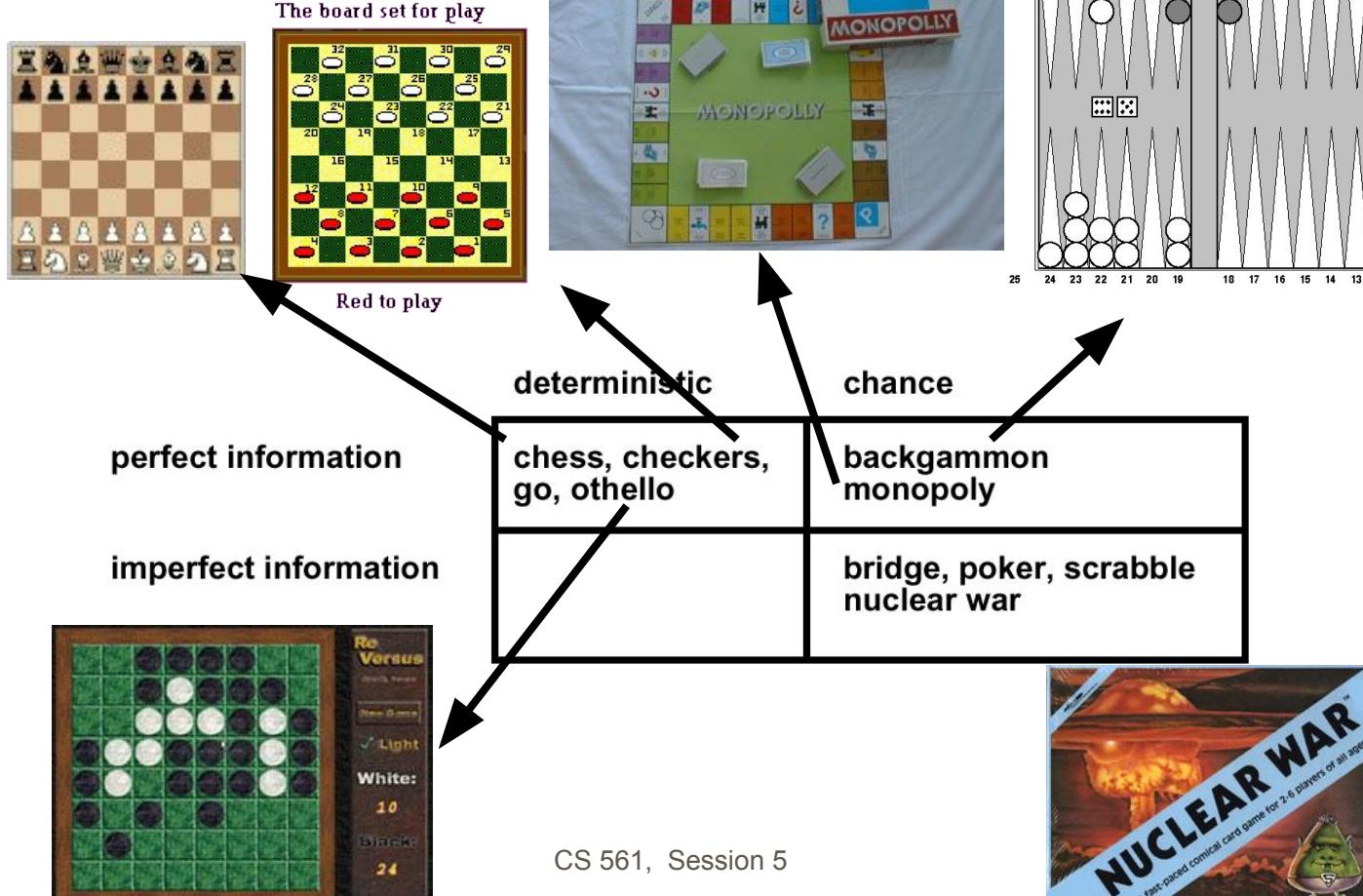
Example: Tic-Tac-Toe



Type of Games

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

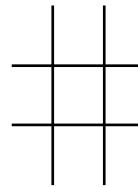
Type of Games



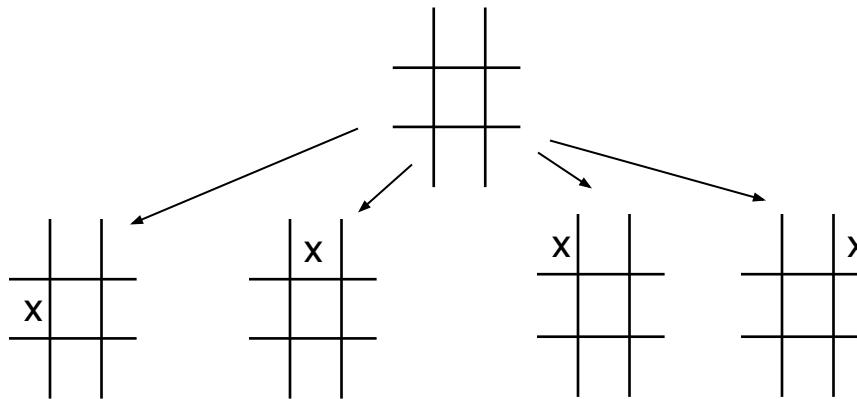
The Minimax Algorithm

- Perfect play for deterministic environments with perfect information
- **Basic idea:** choose move with highest minimax value
 - = best achievable payoff against best play
- **Algorithm:**
 1. Generate game tree completely
 2. Determine utility of each terminal state
 3. Propagate the utility values upward in the tree by applying MIN and MAX operators on the nodes in the current level
 4. At the root node use minimax decision to select the move with the max (of the min) utility value
- Steps 2 and 3 in the algorithm assume that the opponent will play perfectly.

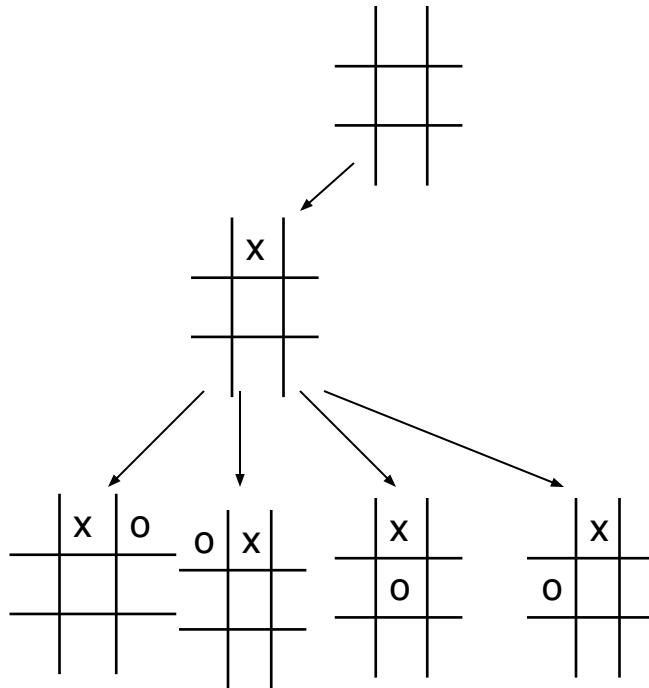
Generate Game Tree



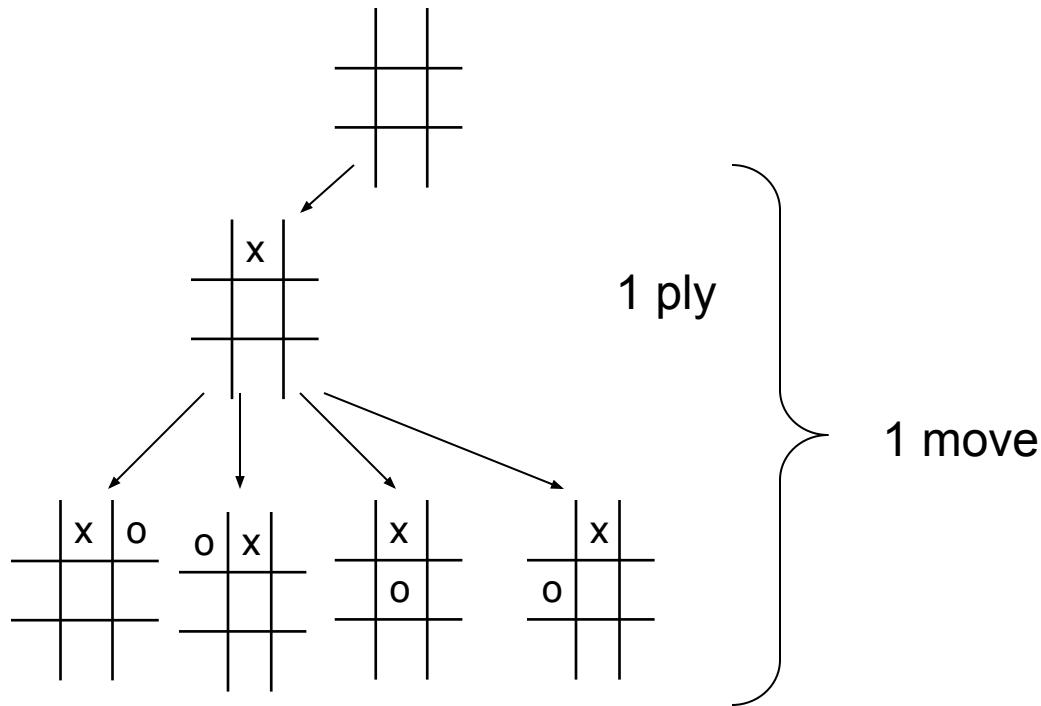
Generate Game Tree



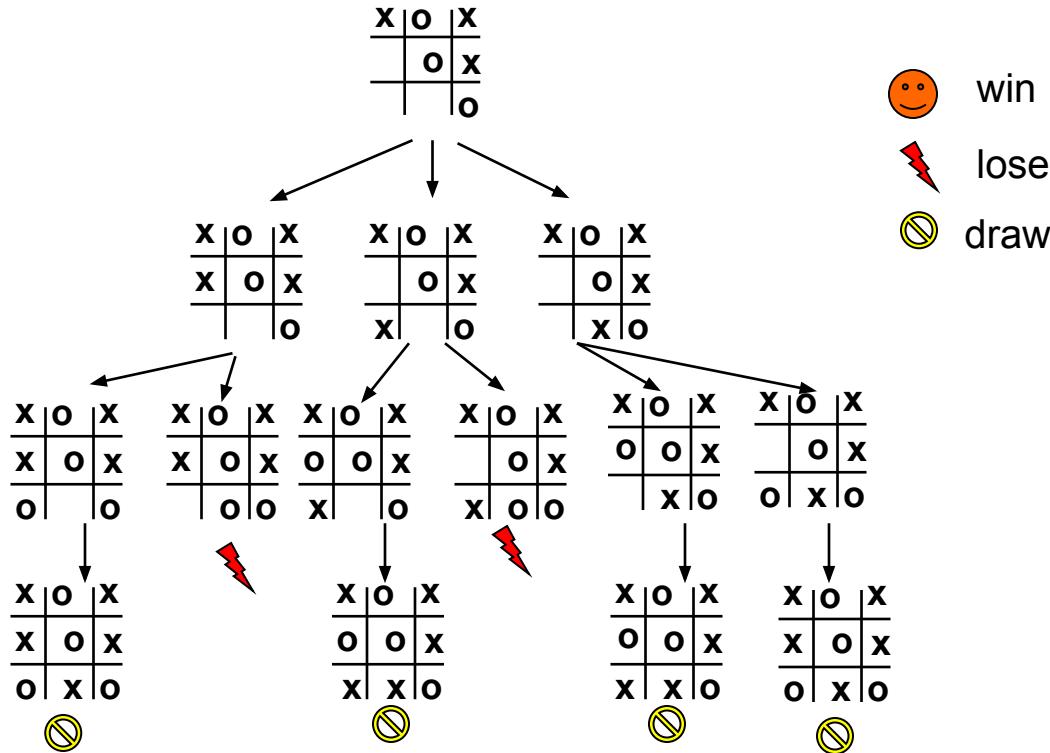
Generate Game Tree



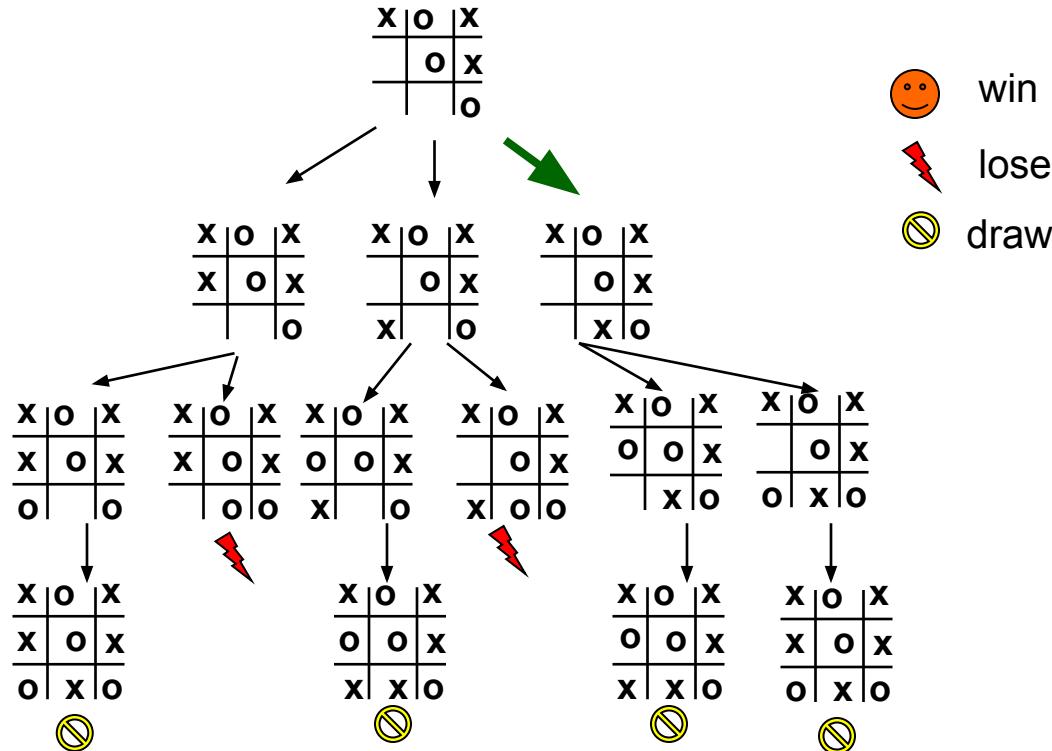
Generate Game Tree



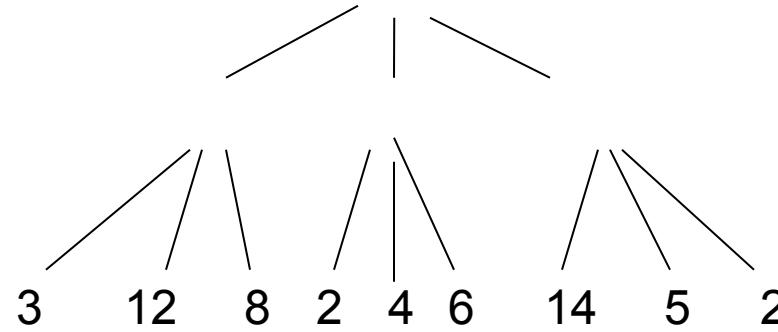
A Subtree



What is a Good Move?

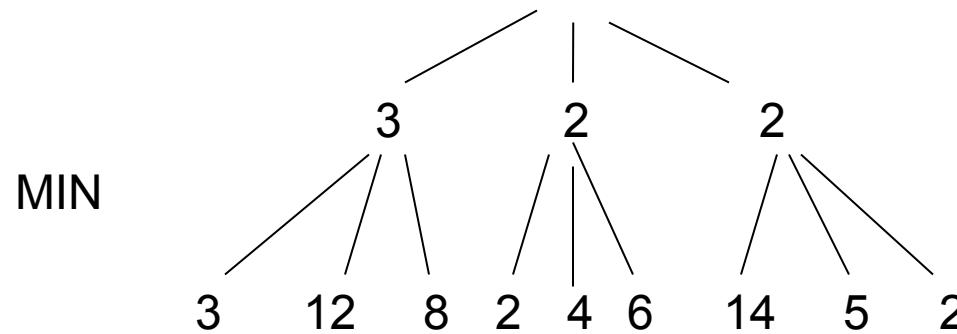


Minimax



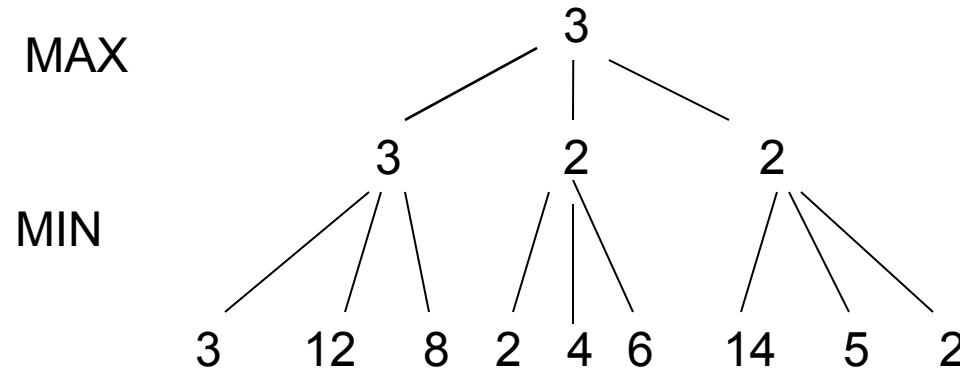
- Minimize opponent's chance
- Maximize your chance

Minimax



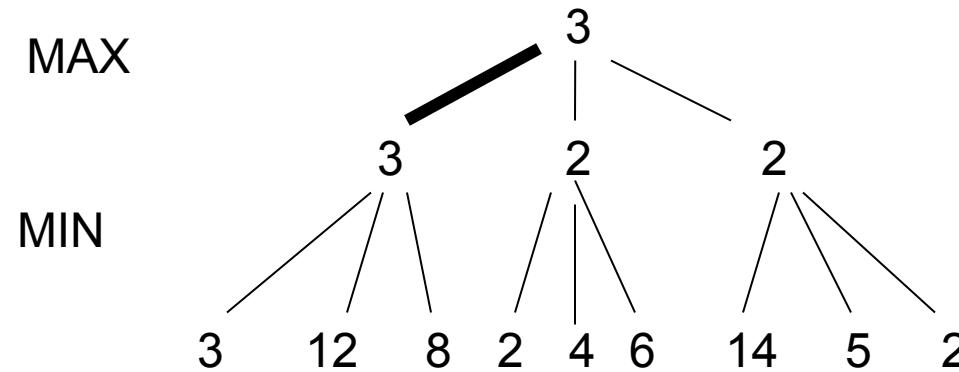
- Minimize opponent's chance
- Maximize your chance

Minimax



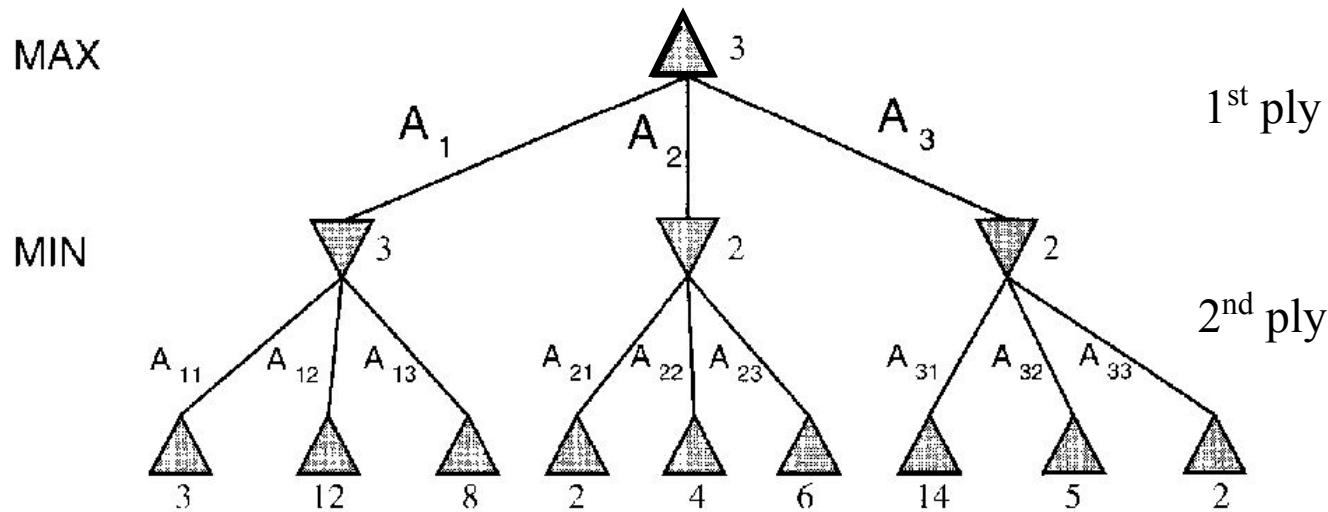
- Minimize opponent's chance
- Maximize your chance

Minimax



- Minimize opponent's chance
- Maximize your chance

minimax = maximum of the minimum



Minimax: Recursive implementation

```
function MINIMAX-DECISION(state) returns an action
    return arg maxa ∈ ACTIONS(s) MIN-VALUE(RESULT(state, a))
```

```
function MAX-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← −∞
    for each a in ACTIONS(state) do
        v ← MAX(v, MIN-VALUE(RESULT(s, a)))
    return v
```

```
function MIN-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← ∞
    for each a in ACTIONS(state) do
        v ← MIN(v, MAX-VALUE(RESULT(s, a)))
    return v
```

Complete: ?
Optimal: ?

Time complexity: ?
Space complexity: ?

Minimax: Recursive implementation

```
function MINIMAX-DECISION(state) returns an action
    return arg maxa ∈ ACTIONS(s) MIN-VALUE(RESULT(state, a))
```

```
function MAX-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← −∞
    for each a in ACTIONS(state) do
        v ← MAX(v, MIN-VALUE(RESULT(s, a)))
    return v
```

```
function MIN-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← ∞
    for each a in ACTIONS(state) do
        v ← MIN(v, MAX-VALUE(RESULT(s, a)))
    return v
```

Complete: Yes, for finite state-space **Time complexity:** $O(b^m)$

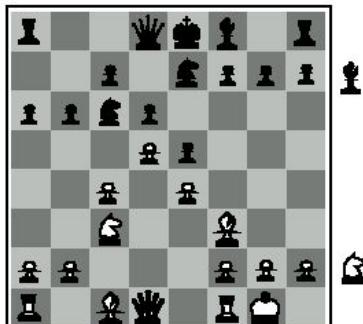
Optimal: Yes

Space complexity: $O(bm)$ (= DFS
Does not keep all nodes in memory.)

Move Evaluation without Complete Search

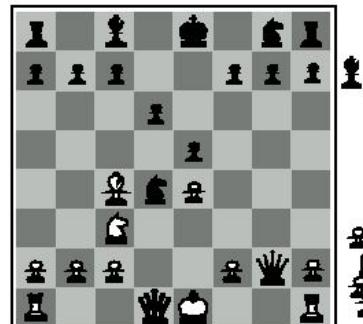
- Complete search is too complex and impractical
- **Evaluation function:** evaluates value of state using **heuristics** and cuts off search
- **New MINIMAX:**
 - **CUTOFF-TEST:** cutoff test to replace the termination condition (e.g., deadline, depth-limit, etc.)
 - **EVAL:** evaluation function to replace utility function (e.g., number of chess pieces taken)

Evaluation Functions



Black to move

White slightly better



White to move

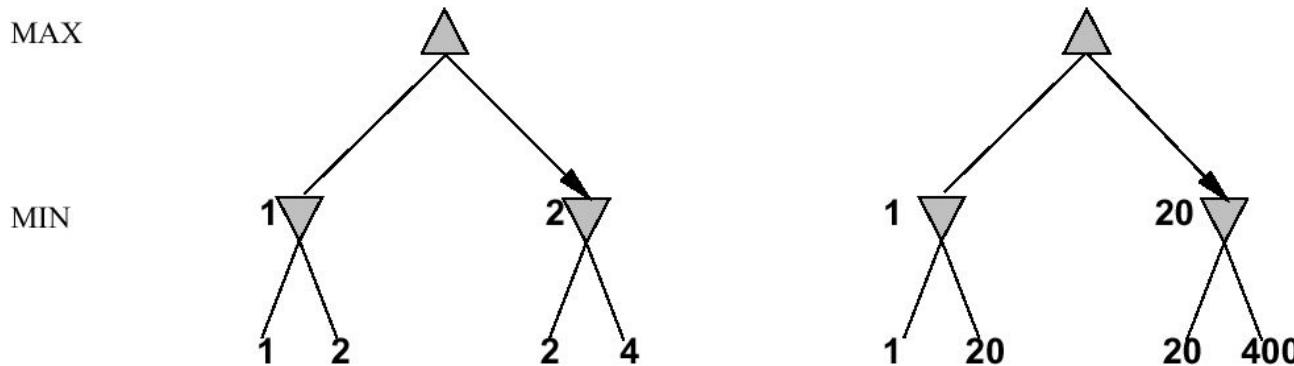
Black winning

- **Weighted linear evaluation function:** to combine n heuristics

$$f = w_1 f_1 + w_2 f_2 + \dots + w_n f_n$$

E.g., w 's could be the values of pieces (1 for prawn, 3 for bishop etc.)
 f 's could be the number of type of pieces on the board

Note: Exact Values do not Matter (Relative Orders are important)



Behaviour is preserved under any *monotonic* transformation of EVAL

Only the order matters:

payoff in deterministic games acts as an *ordinal utility* function

Minimax with Cutoff: Viable Algorithm?

MINIMAXCUTOFF is identical to MINIMAXVALUE except

1. TERMINAL? is replaced by CUTOFF?
2. UTILITY is replaced by EVAL

Does it work in practice?

$$b^m = 10^6, \quad b = 35 \quad \Rightarrow \quad m = 4$$

4-ply lookahead is a hopeless chess player!

4-ply \approx human novice

8-ply \approx typical PC, human master

12-ply \approx Deep Blue, Kasparov

Assume we have
100 seconds,
evaluate 10^4
nodes/s; can
evaluate 10^6
nodes/move

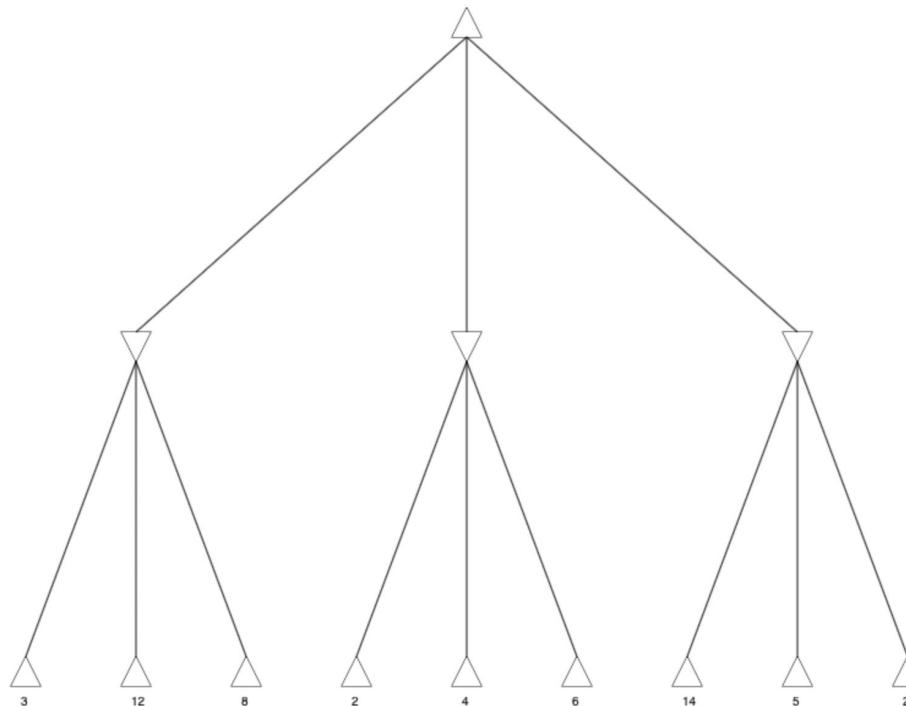
Reduce Search: α - β pruning for search cutoff

- **Pruning:** eliminating a branch of the search tree from consideration without exhaustive examination of each node
- **α - β pruning:** the basic idea is to prune portions of the search tree that cannot improve the utility value of the max or min node, by just considering the values of nodes seen so far.
- Does it work? Yes, it roughly cuts the branching factor from b to \sqrt{b} resulting in double the look-ahead than pure minimax

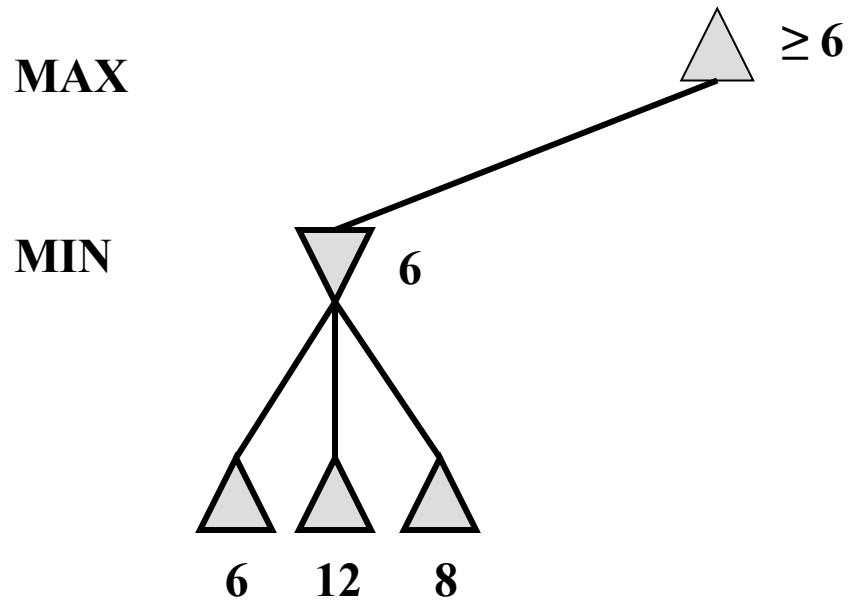
Demo

Demo: minimax game search algorithm with alpha-beta pruning (using html5, canvas, javascript, css)

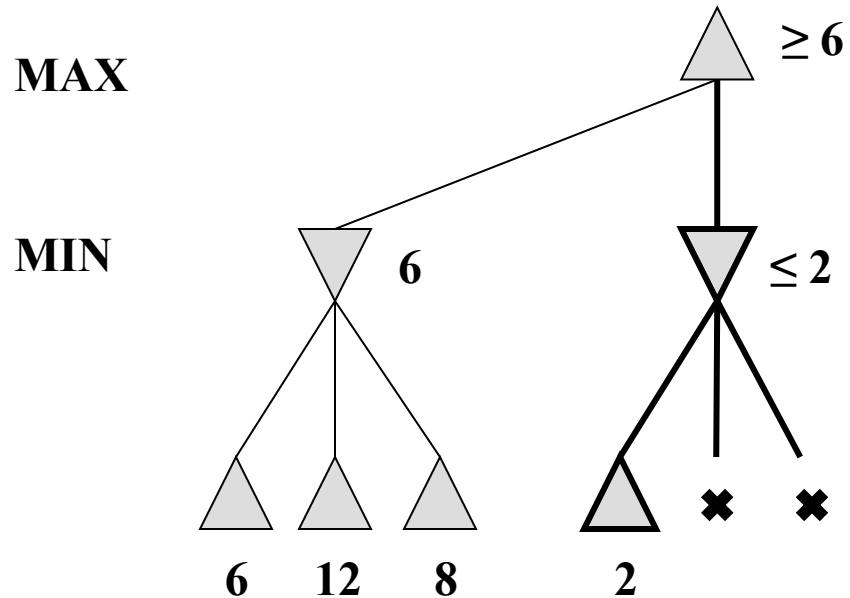
Enter the game tree structure:	<input type="text" value="3 3 3 3"/>	(hint: Insert the game tree structure composed by a list with the number of child nodes for each internal node, ordered by level and left to right)
Enter the game tree terminal values:	<input type="text" value="3 12 8 2 4 6 14 5 2"/>	(hint: Insert the utility values for the game tree terminal/leaf nodes ordered left to right)
<input type="button" value="Create new game tree"/> Messages:  <pre>tree.height=2;tree.number_nodes=13;Tree>0,0,3,1,-1000,450,30,-1000,1000,-1,-1,undefined,1,1,3,4,1000,180,295,undefined,undefined,0,-1,undefined,2,4,0,-1,3,90,560,undefined,undefined,1,undefined,undefined,5,2,0,-1,12,180,560,undefined,undefined,1,undefined,undefined,6,2,0,-1,8,270,560,undefined,undefined,1,undefined,undefined,7,2,0,-1,2,360,560,undefined,undefined,2,undefined,undefined,8,2,0,-1,4,450,560,undefined</pre>		
<input type="button" value="Step minimax"/> <input type="button" value="Run minimax"/>		



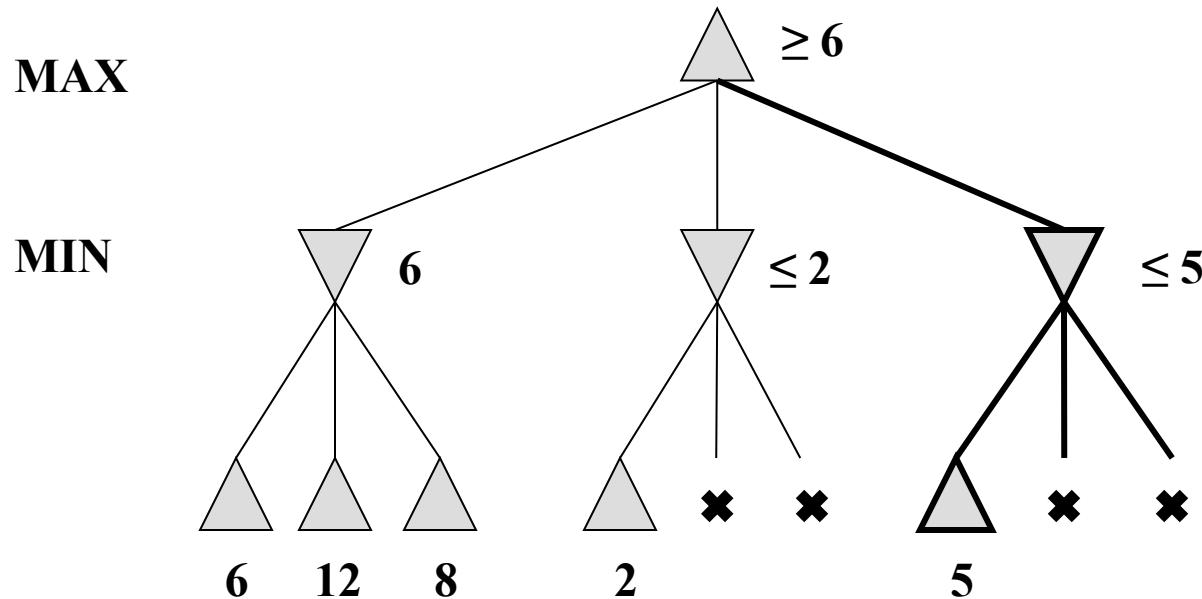
α - β pruning: example



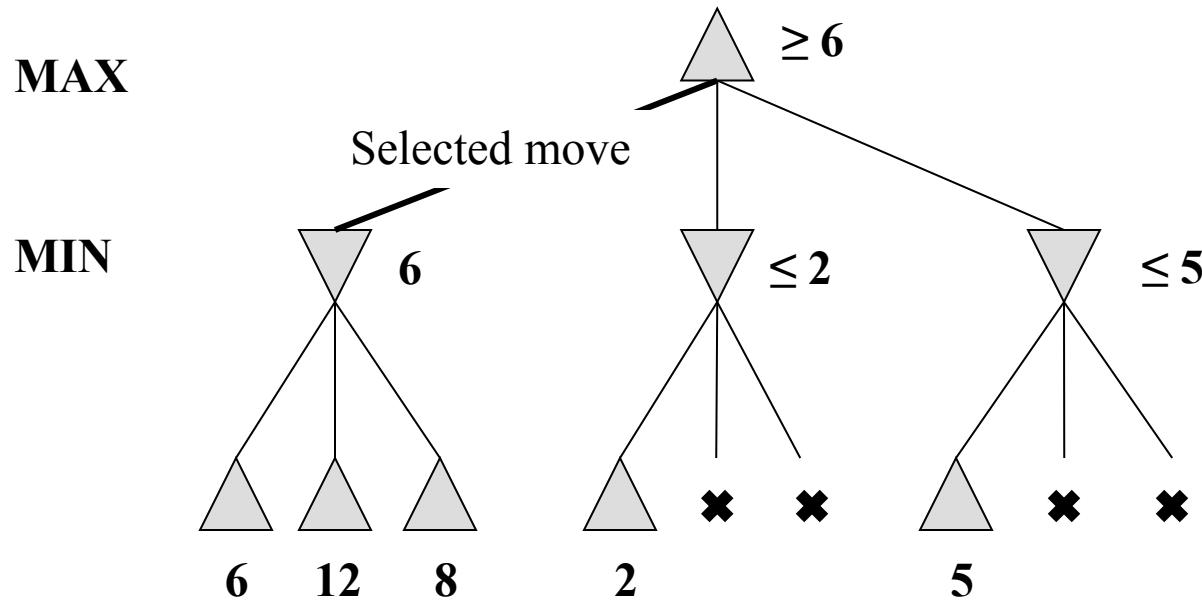
α - β pruning: example



α - β pruning: example



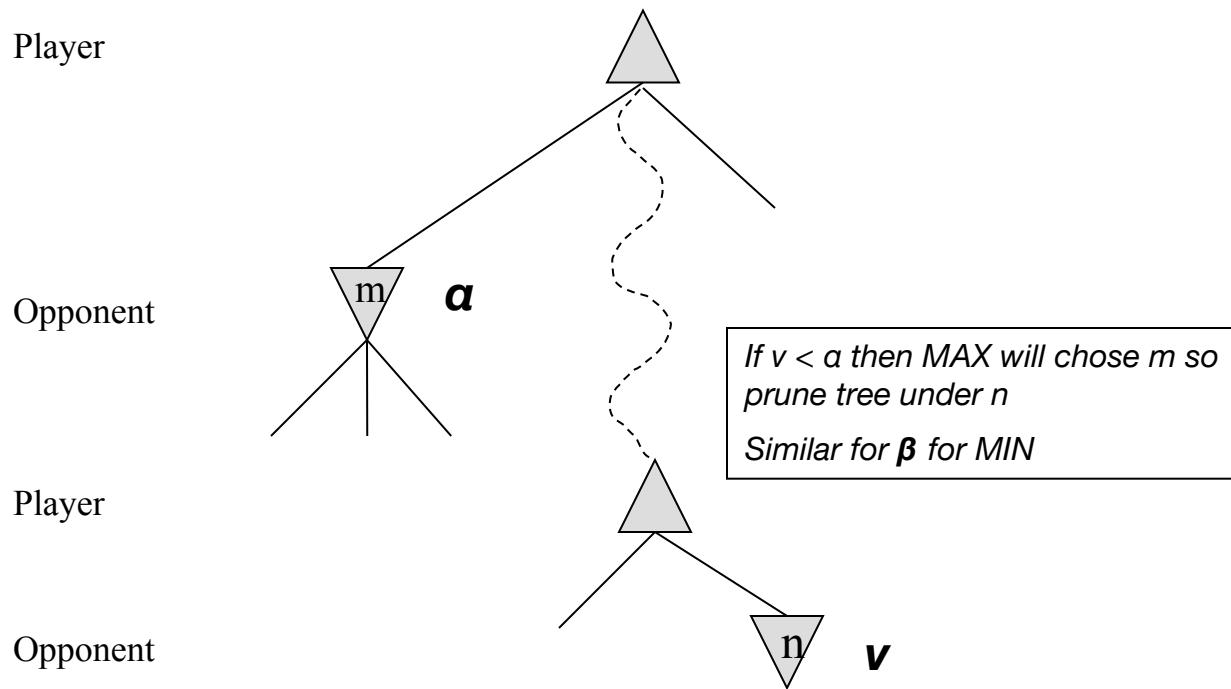
α - β pruning: example



Interactive demo:

<https://www.yosenspace.com/posts/computer-science-game-trees.html>

α - β pruning: general principle



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* depth of search

⇒ can easily reach depth 8 and play good chess

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

The α - β algorithm

function ALPHA-BETA-SEARCH(*state*) **returns** an action

$v \leftarrow \text{MAX-VALUE}(\textit{state}, -\infty, +\infty)$

return the *action* in ACTIONS(*state*) with value *v*

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

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

$v \leftarrow -\infty$

for each *a* in ACTIONS(*state*) **do**

$v \leftarrow \text{MAX}(v, \text{MIN-VALUE}(\text{RESULT}(s, a), \alpha, \beta))$

if $v \geq \beta$ **then return** *v*

$\alpha \leftarrow \text{MAX}(\alpha, v)$

return *v*

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

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

$v \leftarrow +\infty$

for each *a* in ACTIONS(*state*) **do**

$v \leftarrow \text{MIN}(v, \text{MAX-VALUE}(\text{RESULT}(s, a), \alpha, \beta))$

if $v \leq \alpha$ **then return** *v*

$\beta \leftarrow \text{MIN}(\beta, v)$

return *v*

More on the α - β algorithm

- Same basic idea as minimax, but prune (cut away) branches of the tree that we know will not contain the solution.
- We know a branch will not contain a solution once we know a better outcome has already been discovered in a previously explored branch.

Remember: Minimax: Recursive implementation

```
function MINIMAX-DECISION(state) returns an action
    return arg maxa ∈ ACTIONS(s) MIN-VALUE(RESULT(state, a))
```

```
function MAX-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← −∞
    for each a in ACTIONS(state) do
        v ← MAX(v, MIN-VALUE(RESULT(s, a)))
    return v
```

```
function MIN-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← ∞
    for each a in ACTIONS(state) do
        v ← MIN(v, MAX-VALUE(RESULT(s, a)))
    return v
```

Complete: Yes, for finite state-space **Time complexity:** $O(b^m)$

Optimal: Yes

Space complexity: $O(bm)$ (= DFS
Does not keep all nodes in memory.)

The α - β algorithm

function ALPHA-BETA-SEARCH(*state*) **returns** an action

$v \leftarrow \text{MAX-VALUE}(\textit{state}, -\infty, +\infty)$

return the *action* in ACTIONS(*state*) with value *v*

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

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

$v \leftarrow -\infty$

for each *a* in ACTIONS(*state*) **do**

$v \leftarrow \text{MAX}(v, \text{MIN-VALUE}(\text{RESULT}(s, a), \alpha, \beta))$

if $v \geq \beta$ **then return** *v*

$\alpha \leftarrow \text{MAX}(\alpha, v)$

return *v*

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

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

$v \leftarrow +\infty$

for each *a* in ACTIONS(*state*) **do**

$v \leftarrow \text{MIN}(v, \text{MAX-VALUE}(\text{RESULT}(s, a), \alpha, \beta))$

if $v < \alpha$ **then return** *v*

$\beta \leftarrow \text{MIN}(\beta, v)$

return *v*

More on the α - β algorithm

- Same basic idea as minimax, but prune (cut away) branches of the tree that we know will not contain the solution.
- Because minimax is depth-first, let's consider nodes along a given path in the tree. Then, as we go along this path, we keep track of:
 - a : Best choice so far for MAX
 - β : Best choice so far for MIN

The α - β algorithm

```
function ALPHA-BETA-SEARCH(state) returns an action
    v  $\leftarrow$  MAX-VALUE(state,  $-\infty$ ,  $+\infty$ )
    return the action in ACTIONS(state) with value v
```

```
function MAX-VALUE(state,  $\alpha$ ,  $\beta$ ) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v  $\leftarrow -\infty$ 
    for each a in ACTIONS(state) do
        v  $\leftarrow \text{MAX}(v, \text{MIN-VALUE}(\text{RESULT}(s,a), \alpha, \beta))
        if v  $\geq \beta$  then return v
         $\alpha \leftarrow \text{MAX}(\alpha, v)$ 
    return v$ 
```

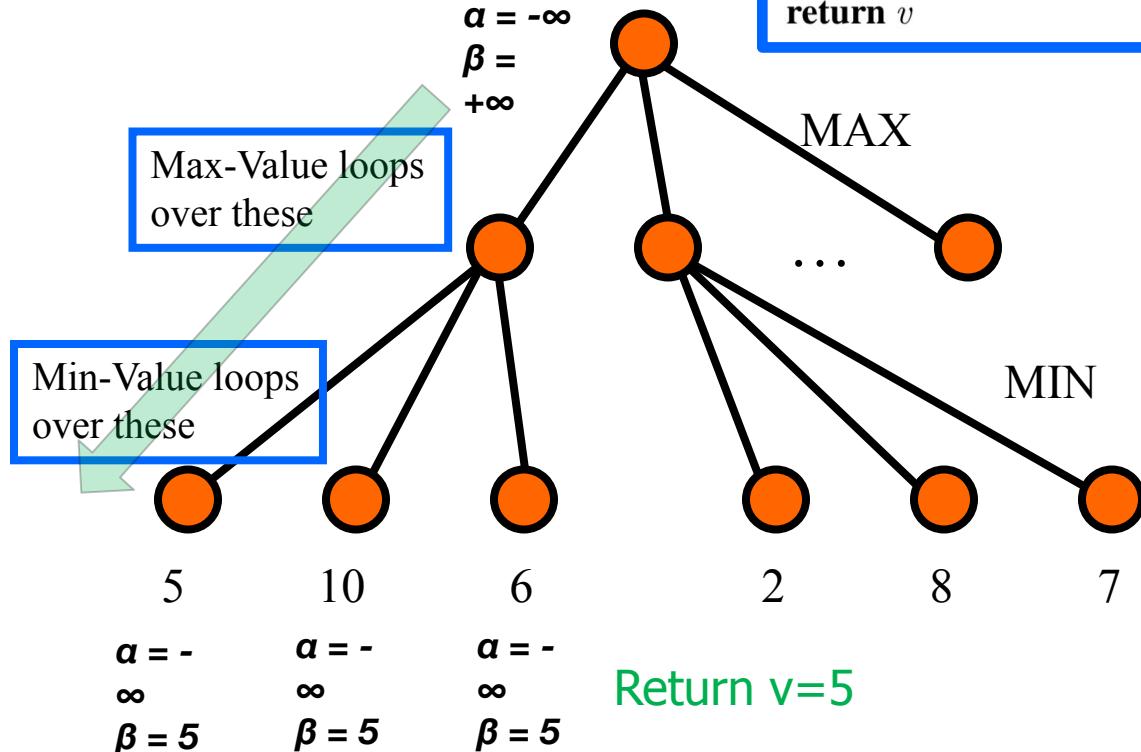
```
function MIN-VALUE(state,  $\alpha$ ,  $\beta$ ) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v  $\leftarrow +\infty$ 
    for each a in ACTIONS(state) do
        v  $\leftarrow \text{MIN}(v, \text{MAX-VALUE}(\text{RESULT}(s,a), \alpha, \beta))
        if v  $\leq \alpha$  then return v
         $\beta \leftarrow \text{MIN}(\beta, v)$ 
    return v$ 
```

Note: α and β are both Local variables. At the Start of the algorithm, We initialize them to $\alpha = -\infty$ and $\beta = +\infty$

More on the α - β algorithm

In Min-Value:

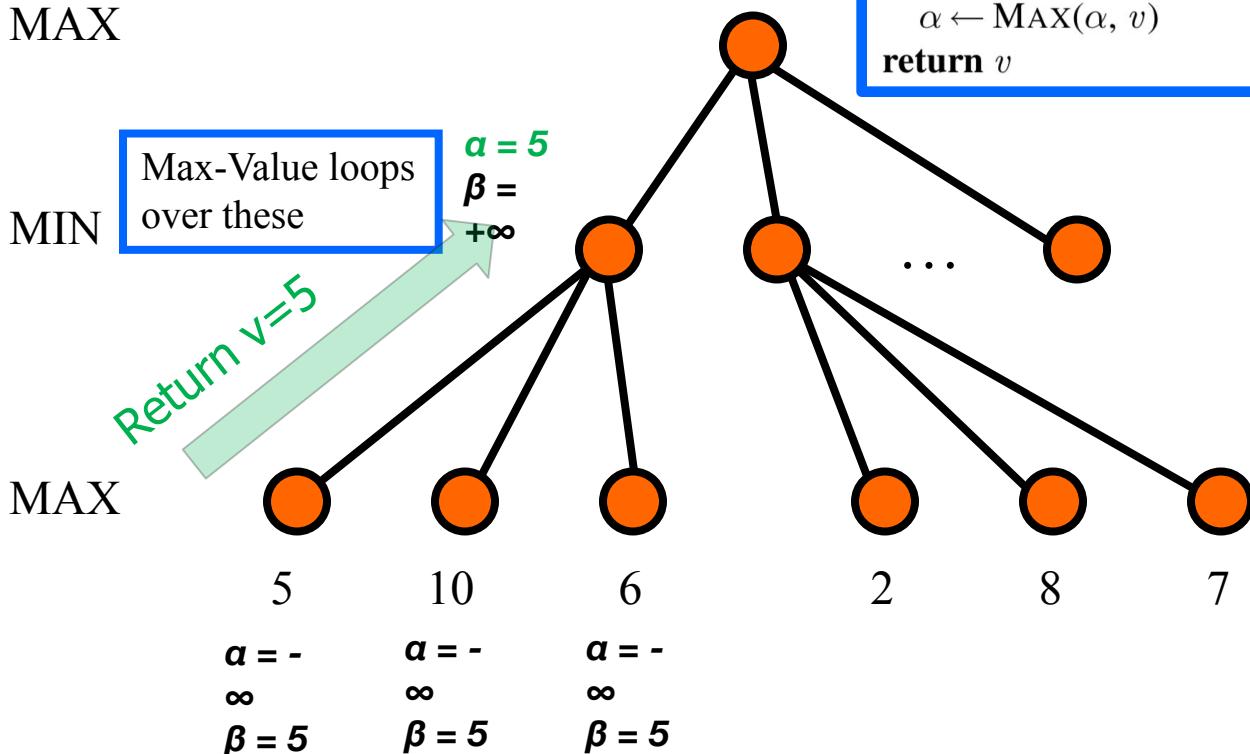
```
v ← +∞  
for each  $a$  in ACTIONS(state) do  
     $v \leftarrow \text{MIN}(v, \text{MAX-VALUE}(\text{RESULT}(s, a), \alpha, \beta))$   
    if  $v \leq \alpha$  then return  $v$   
     $\beta \leftarrow \text{MIN}(\beta, v)$   
return  $v$ 
```



More on the α - β algorithm

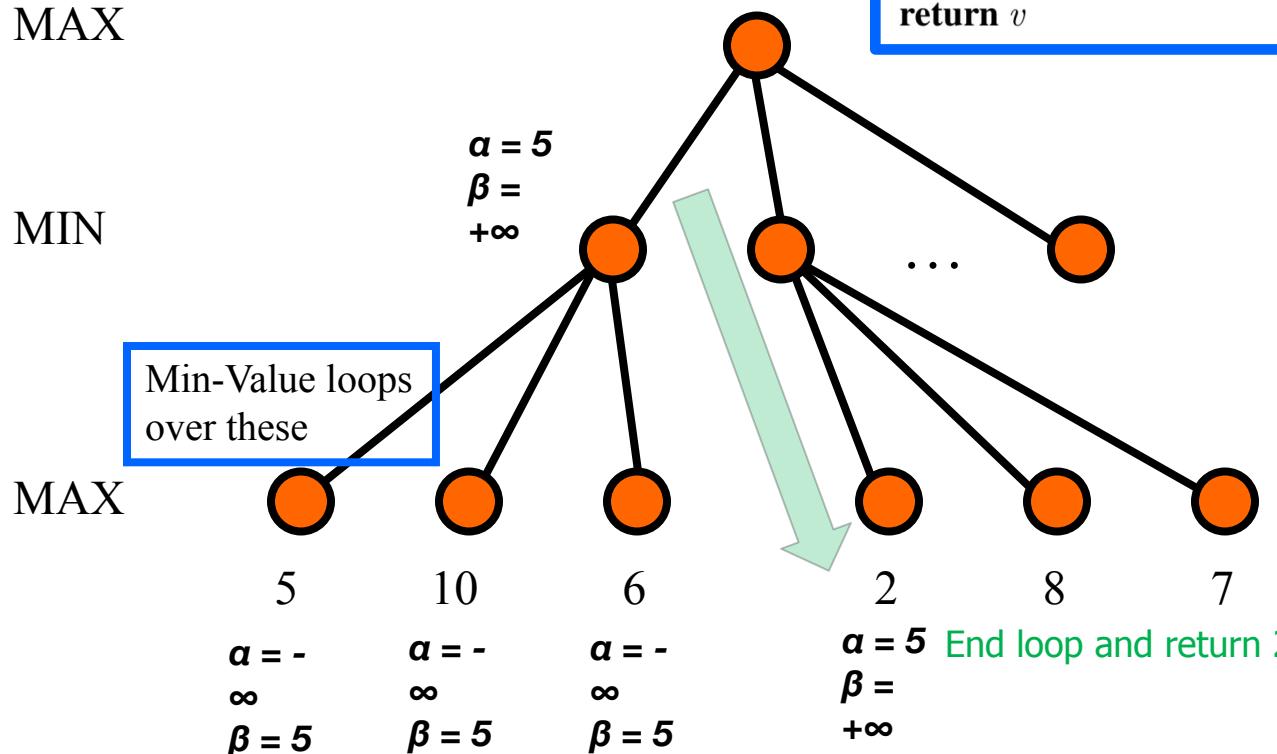
In Max-Value:

```
v ← -∞
for each  $a$  in ACTIONS(state) do
     $v \leftarrow \text{MAX}(v, \text{MIN-VALUE}(\text{RESULT}(s,a), \alpha, \beta))$ 
    if  $v \geq \beta$  then return  $v$  // or  $v \leq \alpha$ 
     $\alpha \leftarrow \text{MAX}(\alpha, v)$ 
return  $v$ 
```



In Min-Value:

More on the α - β algorithm



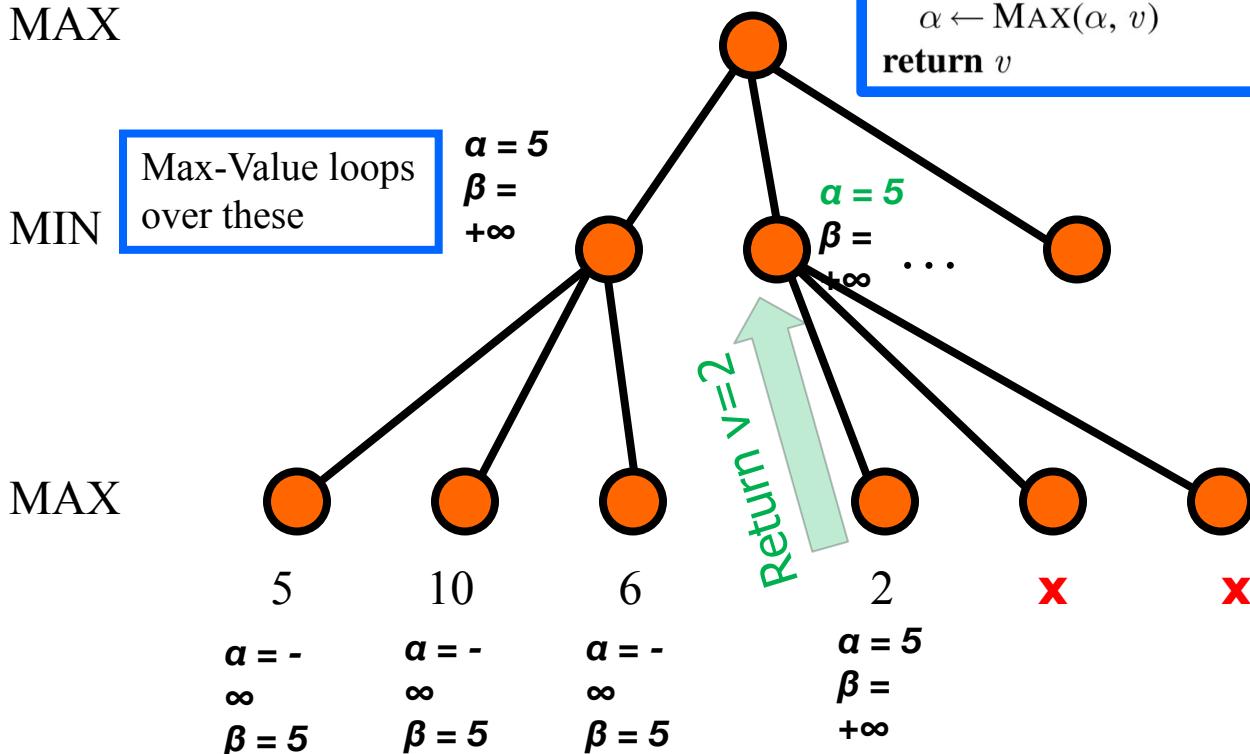
More on the α - β algorithm

In Max-Value:

```

 $v \leftarrow -\infty$ 
for each  $a$  in ACTIONS( $state$ ) do
     $v \leftarrow \text{MAX}(v, \text{MIN-VALUE}(\text{RESULT}(s,a), \alpha, \beta))$ 
    if  $v \geq \beta$  then return  $v$  // or  $v \leq \alpha$ 
     $\alpha \leftarrow \text{MAX}(\alpha, v)$ 
return  $v$ 

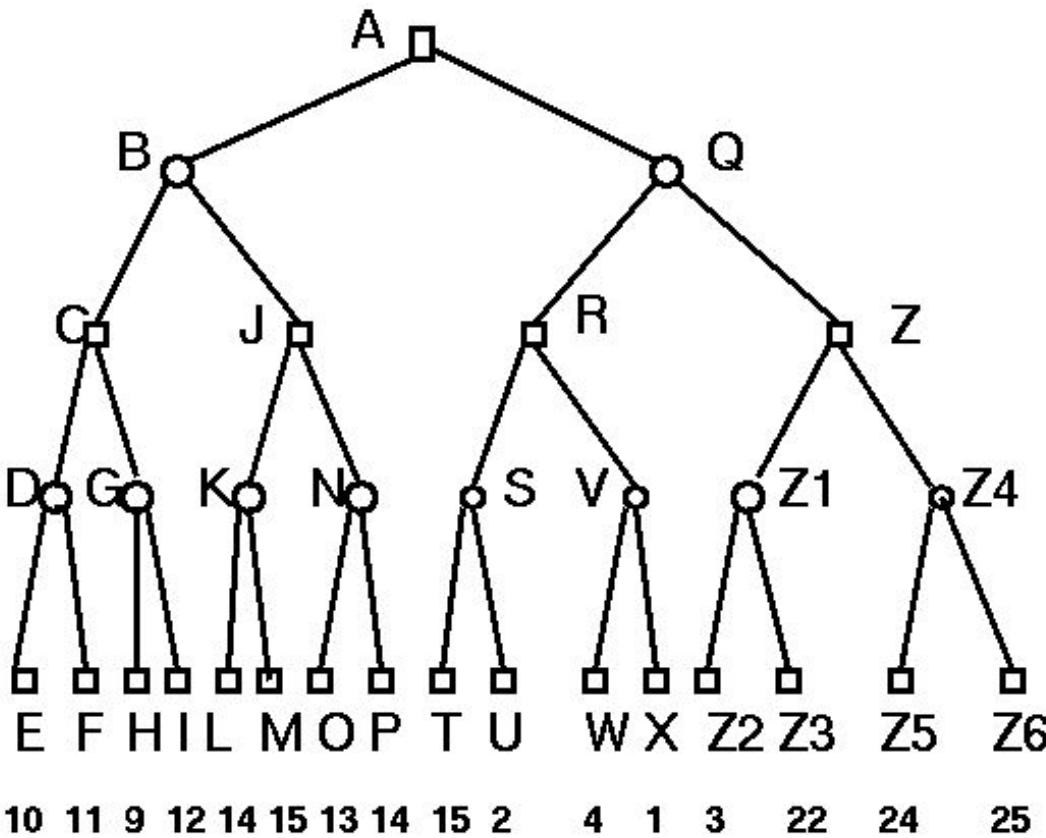
```



Another way to understand the algorithm

- For a given node N,
 - a is the value of N to MAX
 - β is the value of N to MIN

Example



Minimax
+
Alpha-Beta

- ARE MAX NODES
- ARE MIN NODES

α - β algorithm: slight variant (from earlier version of textbook)

Basically MINIMAX + keep track of α, β + prune

```
function MAX-VALUE(state, game, α, β) returns the minimax value of state
    inputs: state, current state in game
            game, game description
             $\alpha$ , the best score for MAX along the path to state
             $\beta$ , the best score for MIN along the path to state
    if CUTOFF-TEST(state) then return EVAL(state)
    for each s in SUCCESSORS(state) do
         $\alpha \leftarrow \text{MAX}(\alpha, \text{MIN-VALUE}(s, \text{game}, \alpha, \beta))$ 
        if  $\alpha \geq \beta$  then return  $\beta$ 
    end
    return  $\alpha$ 
```

```
function MIN-VALUE(state, game, α, β) returns the minimax value of state
    if CUTOFF-TEST(state) then return EVAL(state)
    for each s in SUCCESSORS(state) do
         $\beta \leftarrow \text{MIN}(\beta, \text{MAX-VALUE}(s, \text{game}, \alpha, \beta))$ 
        if  $\beta \leq \alpha$  then return  $\alpha$ 
    end
    return  $\beta$ 
```

Is this wrong
compared to latest
version of textbook?

Please always use
latest version of the
algorithm as in 4th
edition of textbook.

Solution

NODE	TYPE	ALPHA	BETA	SCORE
A	MAX	-Inf	Inf	
B	MIN	-Inf	Inf	
C	MAX	-Inf	Inf	
D	MIN	-Inf	Inf	
E	MAX	10	10	10
D	MIN	-Inf	10	
F	MAX	11	11	11
D	MIN	-Inf	10	10
C	MAX	10	Inf	
G	MIN	10	Inf	
H	MAX	9	9	9
G	MIN	10	9	9
C	MAX	10	Inf	10
B	MIN	-Inf	10	
J	MAX	-Inf	10	
K	MIN	-Inf	10	
L	MAX	14	14	14
K	MIN	-Inf	10	
M	MAX	15	15	15
K	MIN	-Inf	10	10
...				

NODE	TYPE	ALPHA	BETA	SCORE
...				
J	MAX	10	10	10
B	MIN	-Inf	10	10
A	MAX	10	Inf	
Q	MIN	10	Inf	
R	MAX	10	Inf	
S	MIN	10	Inf	
T	MAX	15	15	15
S	MIN	10	15	
U	MAX	2	2	2
S	MIN	10	2	2
R	MAX	10	Inf	
V	MIN	10	Inf	
W	MAX	4	4	4
V	MIN	10	4	4
R	MAX	10	Inf	10
Q	MIN	10	10	10
A	MAX	10	Inf	10

State-of-the-art for deterministic games

Checkers: Chinook ended 40-year-reign of human world champion Marion Tinsley in 1994. Used an endgame database defining perfect play for all positions involving 8 or fewer pieces on the board, a total of 443,748,401,247 positions.

Chess: Deep Blue defeated human world champion Gary Kasparov in a six-game match in 1997. Deep Blue searches 200 million positions per second, uses very sophisticated evaluation, and undisclosed methods for extending some lines of search up to 40 ply.

Othello: human champions refuse to compete against computers, who are too good.

Go: human champions refuse to compete against computers, who are too bad. In go, $b > 300$, so most programs use pattern knowledge bases to suggest plausible moves.

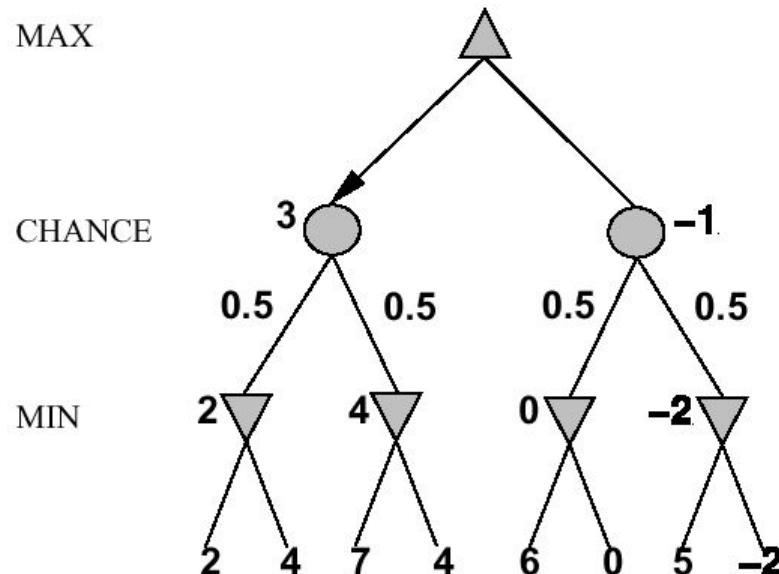
Before
2020

After 2020:
Alpha-GO
Win !!

Nondeterministic games

E.g., in backgammon, the dice rolls determine the legal moves

Simplified example with coin-flipping instead of dice-rolling:



Algorithm for nondeterministic games

EXPECTIMINIMAX gives perfect play

Just like MINIMAX, except we must also handle chance nodes:

...

if *state* is a chance node **then**

return average of EXPECTIMINIMAX-VALUE of SUCCESSORS(*state*)

...

A version of α - β pruning is possible

but only if the leaf values are bounded. Why??

Remember: Minimax algorithm

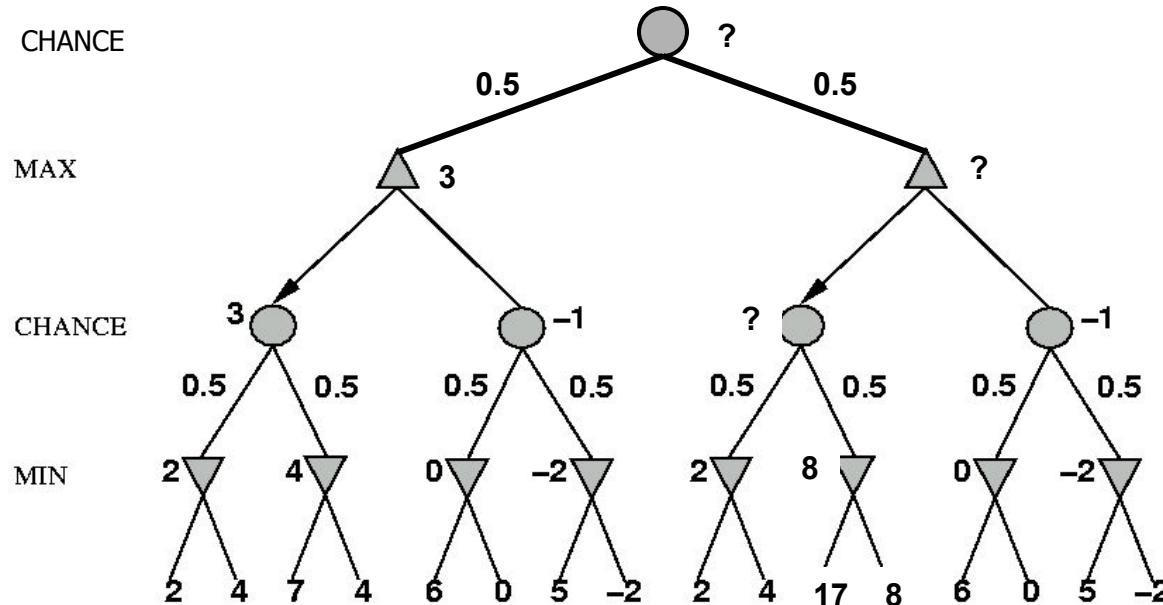
```
function MINIMAX-DECISION(state) returns an action
    return  $\arg \max_{a \in \text{ACTIONS}(s)} \text{MIN-VALUE}(\text{RESULT}(s, a))$ 
```

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

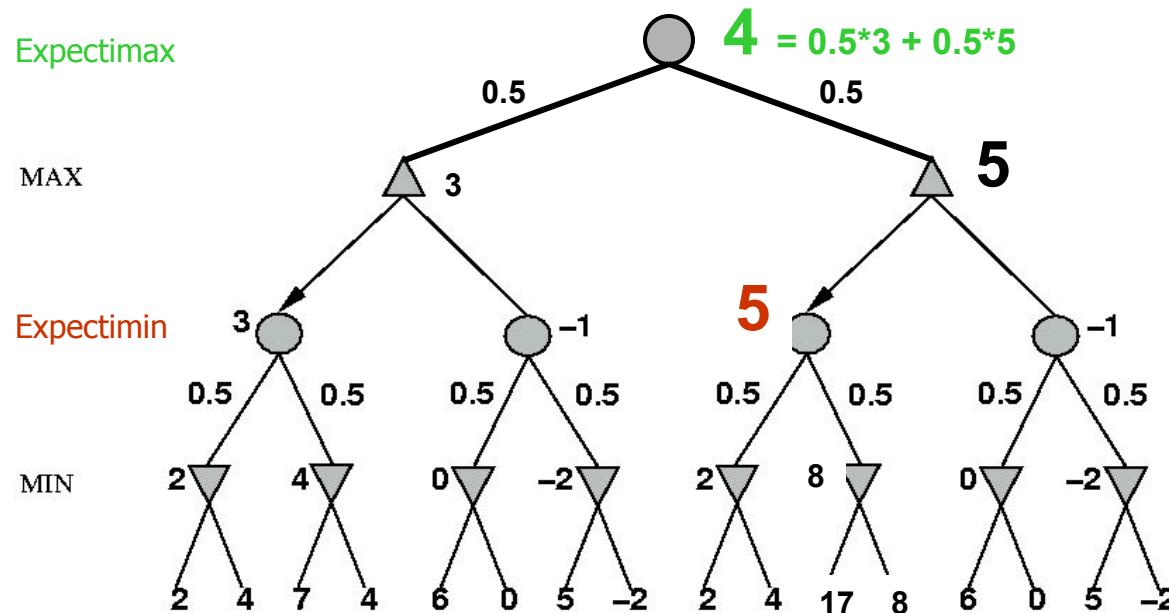
```
function MIN-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
     $v \leftarrow \infty$ 
    for each a in ACTIONS(state) do
         $v \leftarrow \text{MIN}(v, \text{MAX-VALUE}(\text{RESULT}(s, a)))$ 
    return v
```

Nondeterministic games: the element of chance

expectimax and **expectimin**, expected values over all possible outcomes

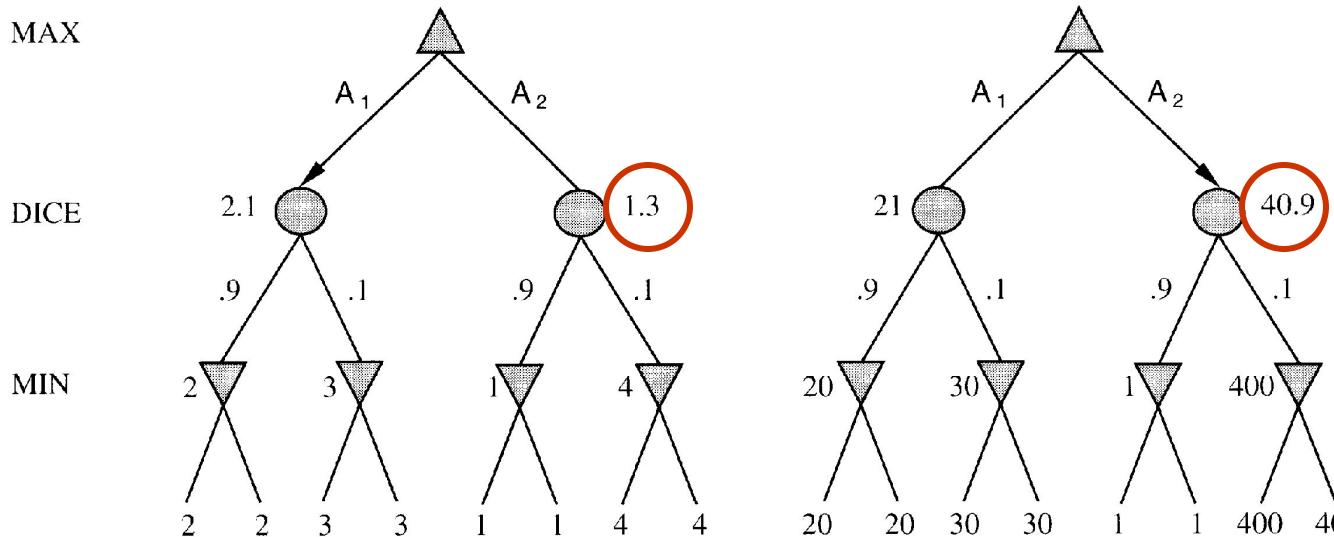


Nondeterministic games: the element of chance



Evaluation functions: Exact values DO matter

Order-preserving transformation do not necessarily behave the same!



State-of-the-art for nondeterministic games

Dice rolls increase b : 21 possible rolls with 2 dice

Backgammon ≈ 20 legal moves (can be 6,000 with 1-1 roll)

$$\text{depth } 4 = 20 \times (21 \times 20)^3 \approx 1.2 \times 10^9$$

As depth increases, probability of reaching a given node shrinks

\Rightarrow value of lookahead is diminished

$\alpha\text{-}\beta$ pruning is much less effective

Summary

Games are fun to work on! (and dangerous)

They illustrate several important points about AI

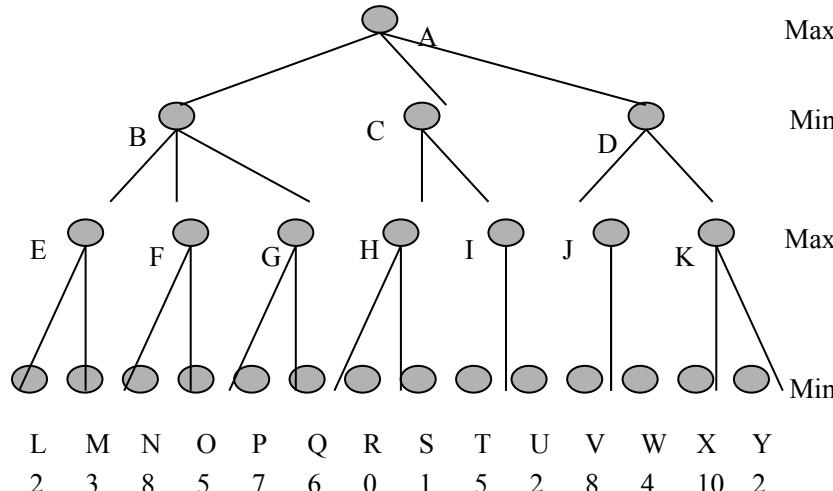
- ◊ perfection is unattainable \Rightarrow must approximate
- ◊ good idea to think about what to think about
- ◊ uncertainty constrains the assignment of values to states

Games are to AI as grand prix racing is to automobile design

Exercise: Game Playing

Consider the following game tree in which the evaluation function values are shown below each leaf node. Assume that the root node corresponds to the maximizing player. Assume the search always visits children left-to-right.

- (a) Compute the backed-up values computed by the minimax algorithm. Show your answer by writing values at the appropriate nodes in the above tree.
- (b) Compute the backed-up values computed by the alpha-beta algorithm. What nodes will not be examined by the alpha-beta pruning algorithm?
- (c) What move should Max choose once the values have been backed-up all the way?



CSCI 561 - Foundation for Artificial Intelligence

Discussion Section (Week 3)

PROF WEI-MIN SHEN WMSHEN@USC.EDU

Outline

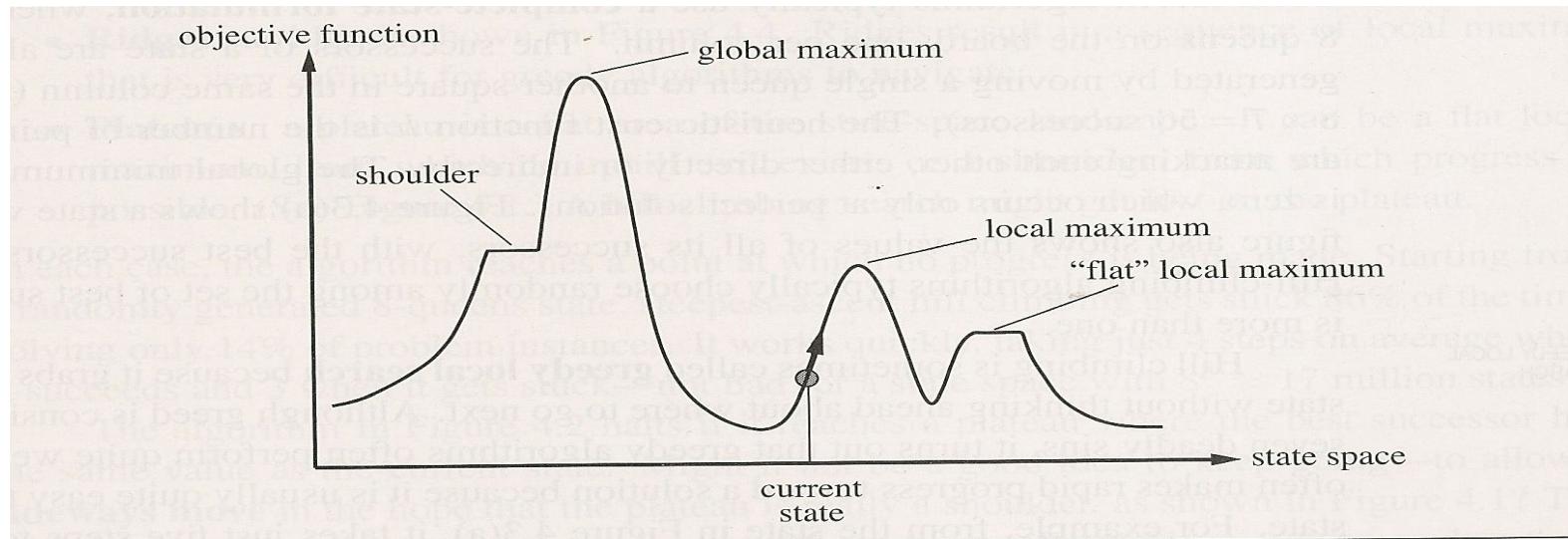
- 1. Function Optimization**
- 2. Constraint Satisfaction**
- 3. Game Playing**

What Is Optimization? So “Simple”?

Given: a hidden objective function $F(x)$

Find: a x^* such that $F(x^*)$ is the global extreme:

$$F(x^*) = \max\{ F(x) \} \text{ or } F(x^*) = \min\{ F(x) \}$$



Why Is It So Important?

All engineering problems are optimizations!

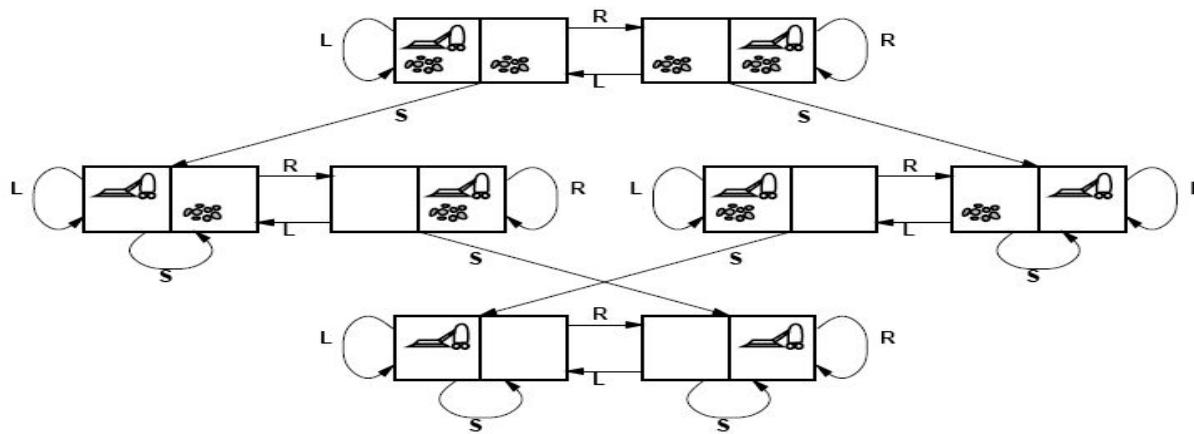
- Design: architecture, software, robots, website, ...
- AI systems: industry, agriculture, military, finance, ...
- For yourself: Getting good grades ☺

The key challenge is: How to represent them properly

For Search, you can represent your desires as the objective function

- Choice 1: **x as a state**, $F(x)$ as the “rewards” of states
- Choice 2: **x as a path**, $F(x)$ as the “rewards” of paths
- Degree of “goodness”: goal related, cost related, etc.

Represent Problems as Optimization



How to represent this problem as an optimization problem?

$x: 00, 01, 10, 11$

- State: LeftRoom_RightRoom, (dirty=0, clean=1)

$F(x)=x$; “the cleaner the room, the higher $F(x)$ ”

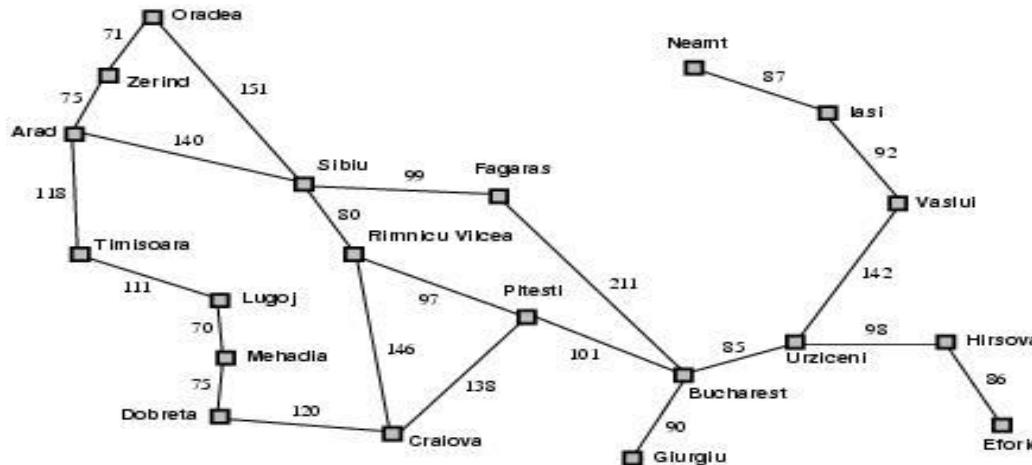
Representing as optimization

Choice 1: x as a city

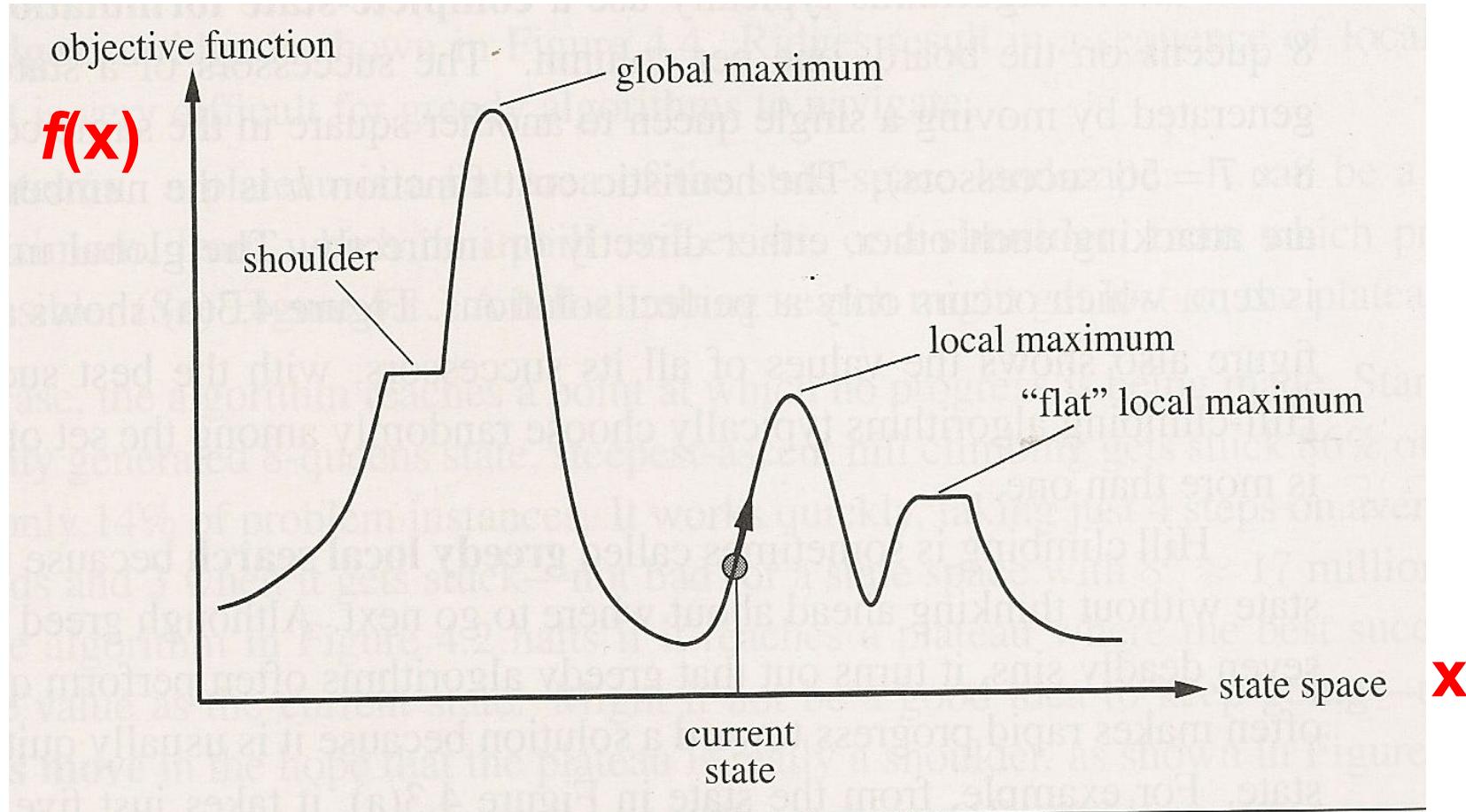
- $F(x)$ is the distance from x to Budapest

Choice 2: x is a sequence of cities starting from Arad

- $F(x)$ has a higher value if the path ends at Budapest
- What about “leads to Budapest”? What about the “cost”?



Why is optimization so hard?



Why is it so hard?

Which way to go next?

How much can you see? (local/partial vs global/complete sensors)

How many points can you remember (incremental)?

How small is your step? (not to skip x^*)

How to get from one x to another (bound by actions)?

How well can you guess (the next x)?

What do you know about the function (continuous)?

How to check if you are done (avoid local extremes)?

Will the function $F(x)$ change by itself (often does)?

How to design the objective function?

In AI, we call it the Representation Challenge

We still don't have a general solution for all

We will illustrate all these using the following story

Representation Challenge for AI

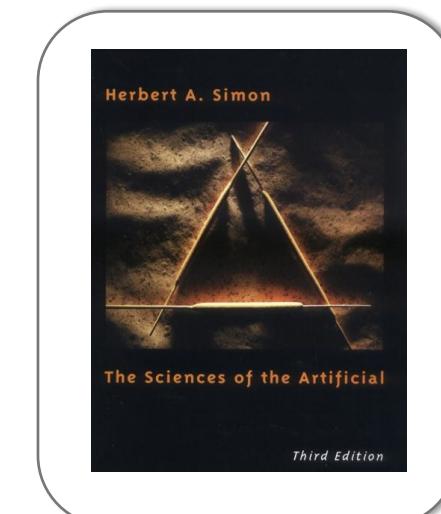
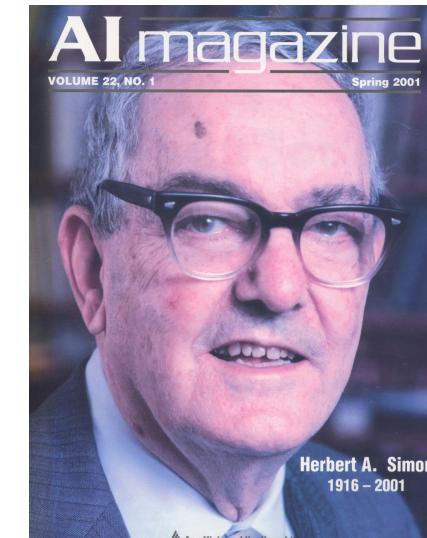
Why is “representation” so important?

A great example from Herbert A. Simon

- Game: Make “a book of 15” from 9 cards
- Goal: first does that wins (can you always win?)
- “The Science of the Artificial” p131

How to find a good representation

- is still an open challenge for AI



4	9	2
3	5	7
8	1	6

Some Optimization Methods

Dynamic programming

Hill climbing

- Idea: Use local gradient(x) to determine direction, always heads to the better
- Pros: simple, local, incremental, no memory
- Cons: may be trapped in local extreme

Simulated Annealing

- Ideas: long and random jumps when temperature is high
- Pros: may avoid local extreme
- Cons: expensive, not always find x^* ,

Genetic algorithms

Sampling techniques (e.g., random walks)

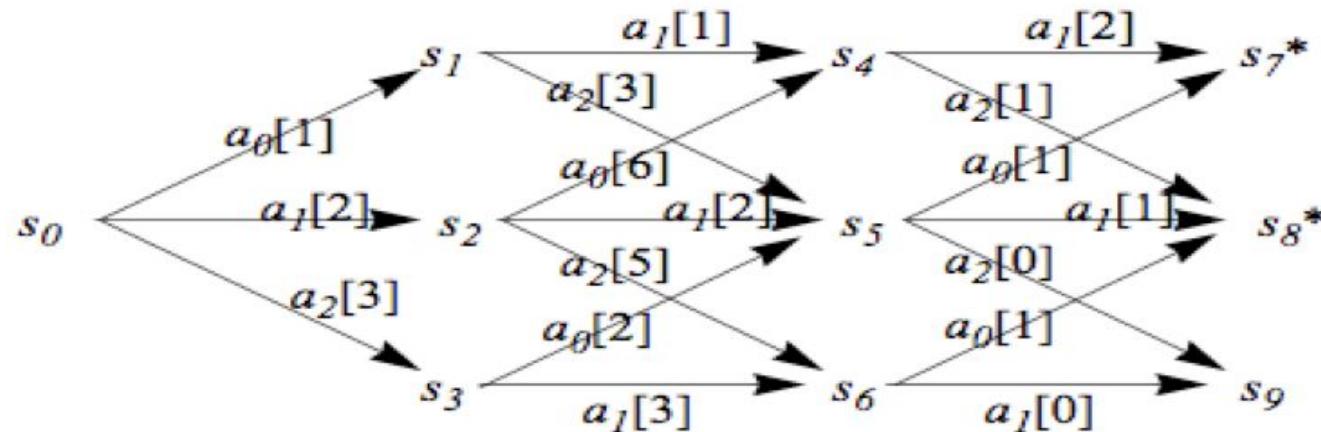
Online (incremental) search

Dynamic Programming

Non-stationary search techniques

Many more “new” methods are being invented as we speak

Dynamic Programming for Optimization (see ALFE 6.1.1)



Note: x is a state, but $f(x)$ is about the most rewarding path from x to a goal state

$$V(s_1) = \max\{R(s_1, a_1) + V(s_4), R(s_1, a_2) + V(s_5)\} = \max\{1 + 2, 3 + 1\} = 4$$

$$\begin{aligned} V(s_2) &= \max\{R(s_2, a_0) + V(s_4), R(s_2, a_1) + V(s_5), R(s_2, a_2) + V(s_6)\} \\ &= \max\{6 + 2, 2 + 1, 5 + 1\} = 8 \end{aligned}$$

$$V(s_3) = \max\{R(s_3, a_0) + V(s_5), R(s_3, a_1) + V(s_6)\} = \max\{2 + 1, 3 + 1\} = 4$$

$$V(s_i) = \max_a \{R(s_i, a) + V(s_j)\} \text{ where } s_i \xrightarrow{a} s_j$$

Iterative Improvement

In many optimization problems, **path** is irrelevant;
the goal state itself is the solution.

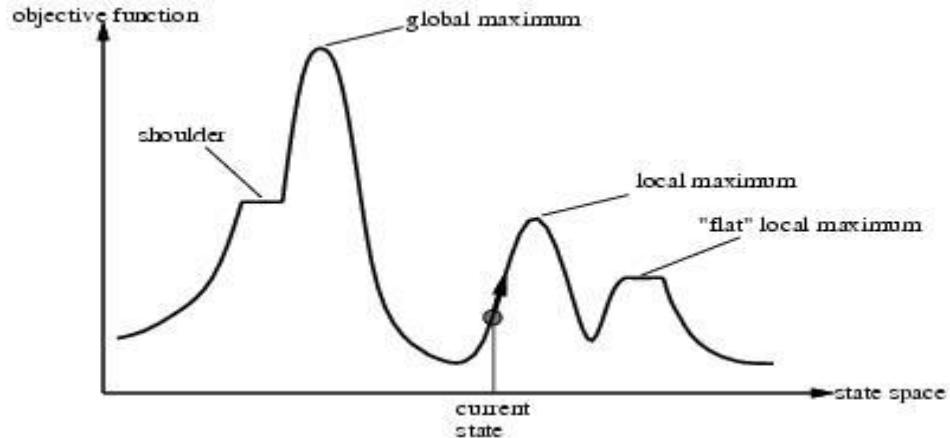
Then, state space = space of “**complete**” configurations.

Algorithm goal:

- find optimal configuration (e.g., TSP), or,
- find configuration satisfying constraints
(e.g., n-queens)

In such cases, can use **iterative improvement algorithms**: keep a single “**current**” state, and try to improve it.

Hill Climbing



Move continuously in the direction of increasing value

- Go to best successor of current state, based on evaluation
 - If more than one best successor, pick randomly among them

Terminate when reach a peak

- May only find a *local maximum*

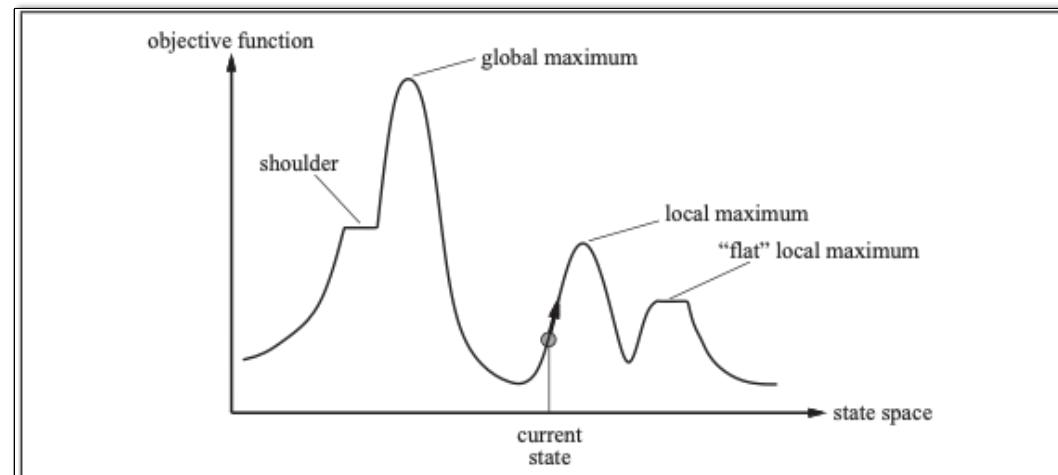
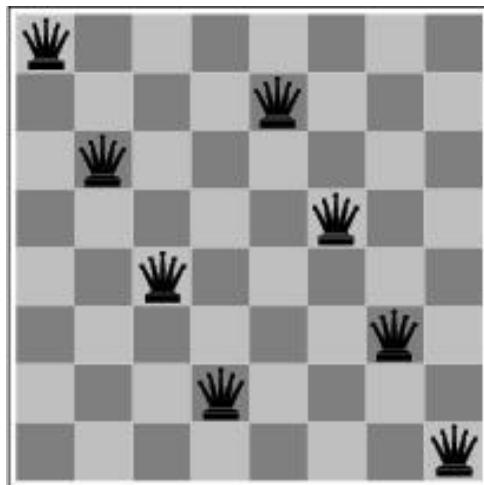
This form of hill climbing is also referred to as

- Steepest-ascent hill climbing
- Greedy local search
- Continuous analogue is *gradient ascent*

Hill Climbing

- What are the advantages and disadvantages of classical hill-climbing?
- Can you think of real-world examples where hill-climbing would be particularly good?

- What about bad?



Key Questions

How big is your step?

You “walk”, but do you “jump”? When? How far?

When do you stop?

Is your technique deterministic and complete?

Simulated Annealing

Escape local maxima by allowing “jump” moves

- When to jump?

- If you always jump too much, then you never settle down
- Gradually decrease the likelihood to jump over time
- Use temperature to determine the likelihood to jump
- Reduce temperature T slowly over the time

- How far to jump?

- Too close? Too far? Random?

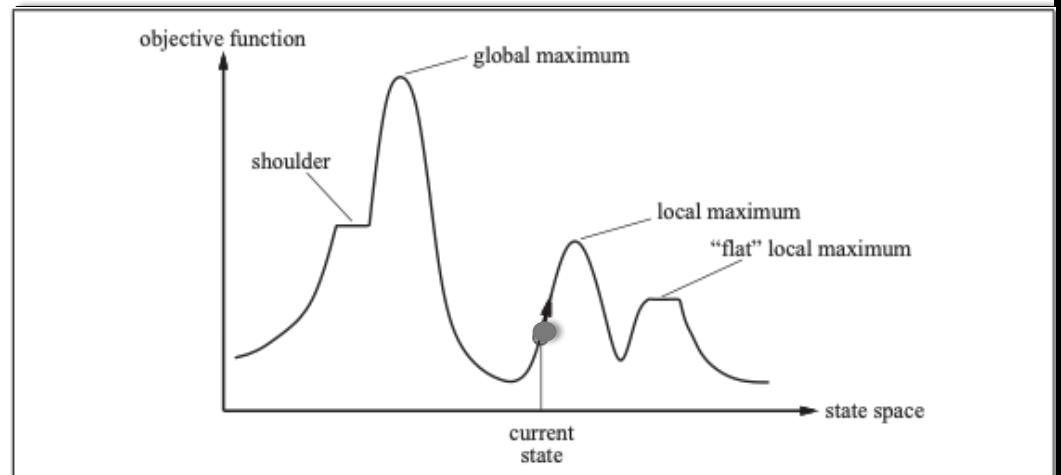
If T decreases slowly “enough,” then you can find the optimal extreme

1. How slow? Infinitely slow
2. Theoretically OK, but infeasible in practice

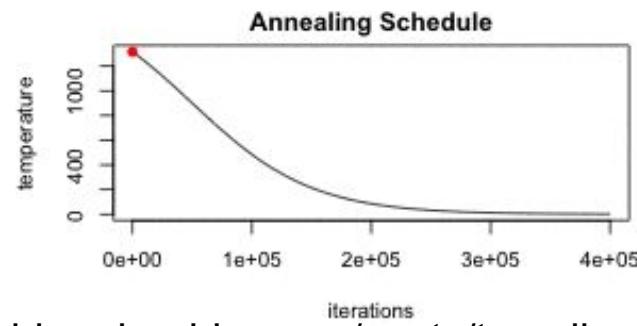
Simulated Annealing

The AIMA book says that simulated annealing is complete (page 125).

- In real-world applications will that be true?
- Why or why not?

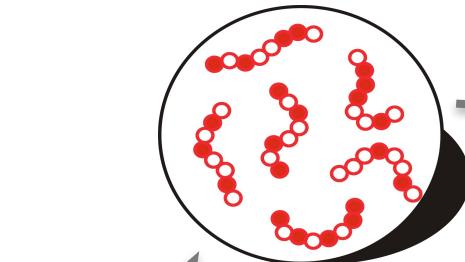


Distance: 43,499 miles
Temperature: 1,316
Iterations: 0

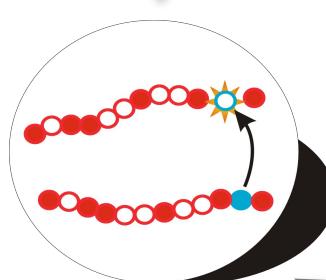
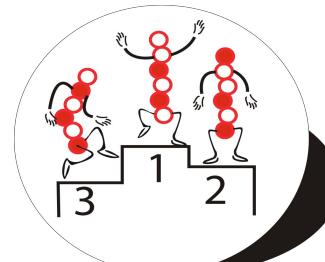


The Genetic Algorithm Cycle

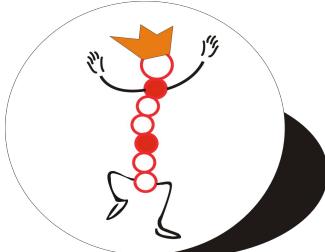
New Population



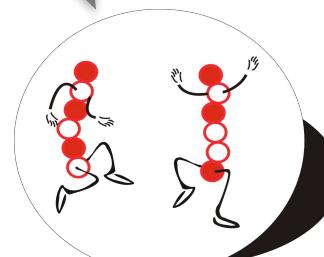
Fitness



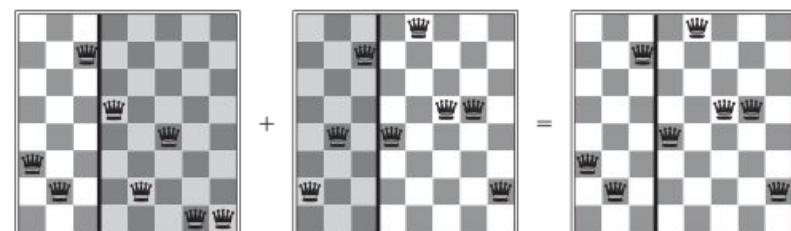
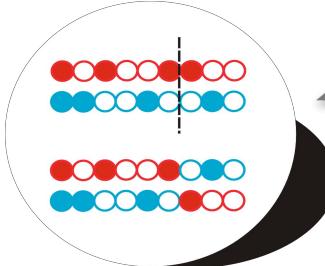
Mutation



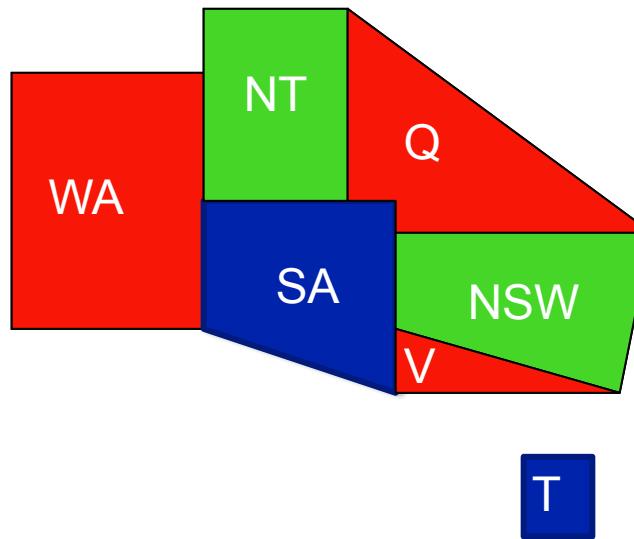
Selection



Crossover

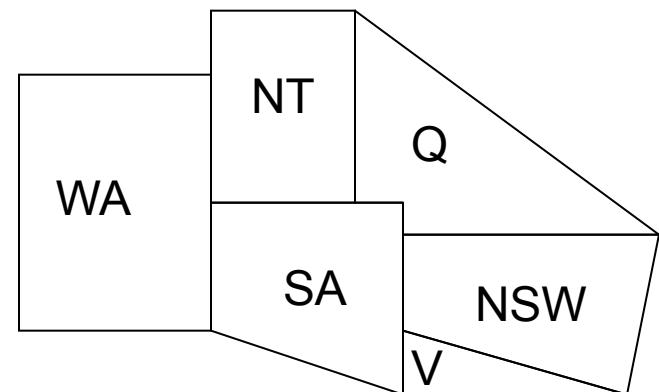
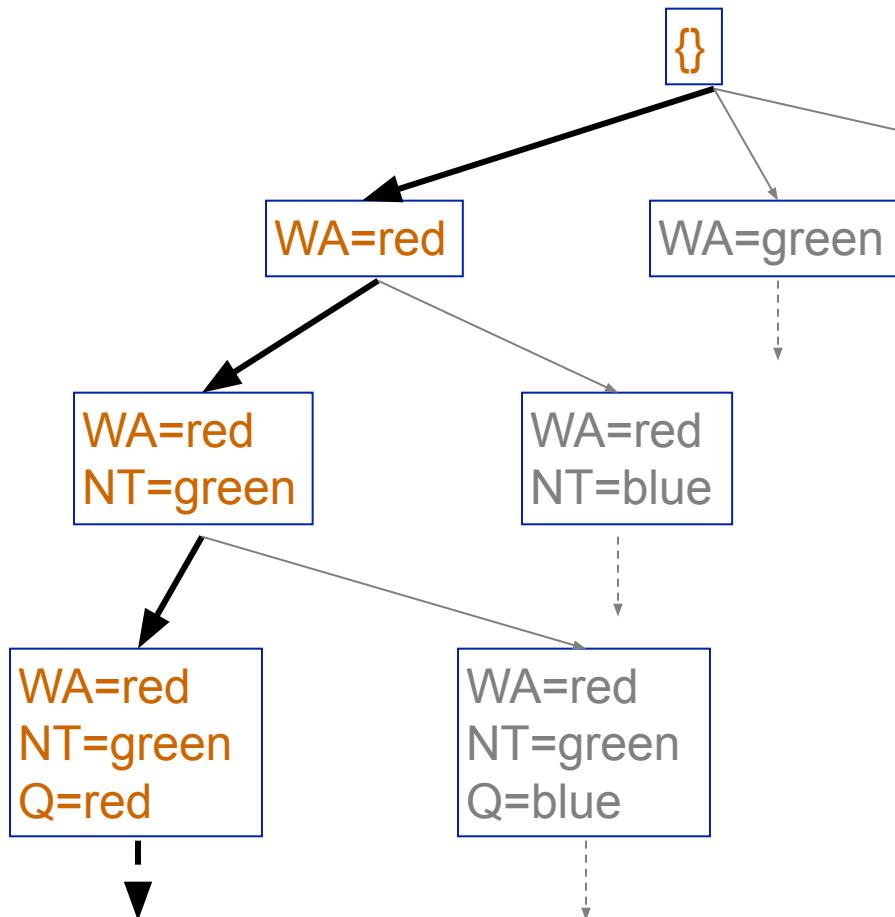


CSP Example: Map Coloring



- 7 variables {WA, NT, SA, Q, NSW, V, T}
- Each variable has the same domain {red, green, blue}
- No two adjacent variables have the same value:
WA ≠ NT, WA ≠ SA, NT ≠ SA, NT ≠ Q, SA ≠ Q, SA ≠ NSW, SA ≠ V, Q ≠ NSW, NSW ≠ V

Backtracking Search: Map Coloring



T

Constraint Propagation

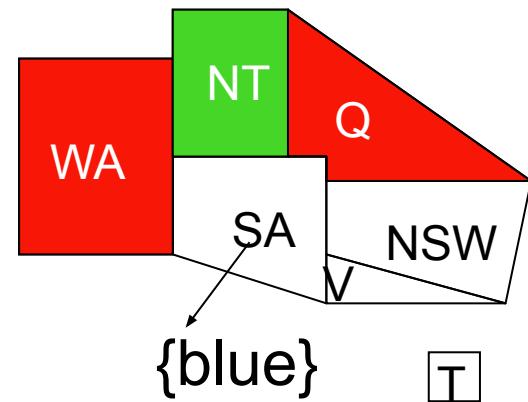
- Which variable X should be assigned a value next?
- In which order should its values be tried?

• **Variable Selection:**

- “Most constrained variable” or “Minimum Remaining Values”
- Degree: variable involved in most constraints on others (tiebreaker)

• **Value Selection:**

- Least constraining value



When to use CSP Techniques?

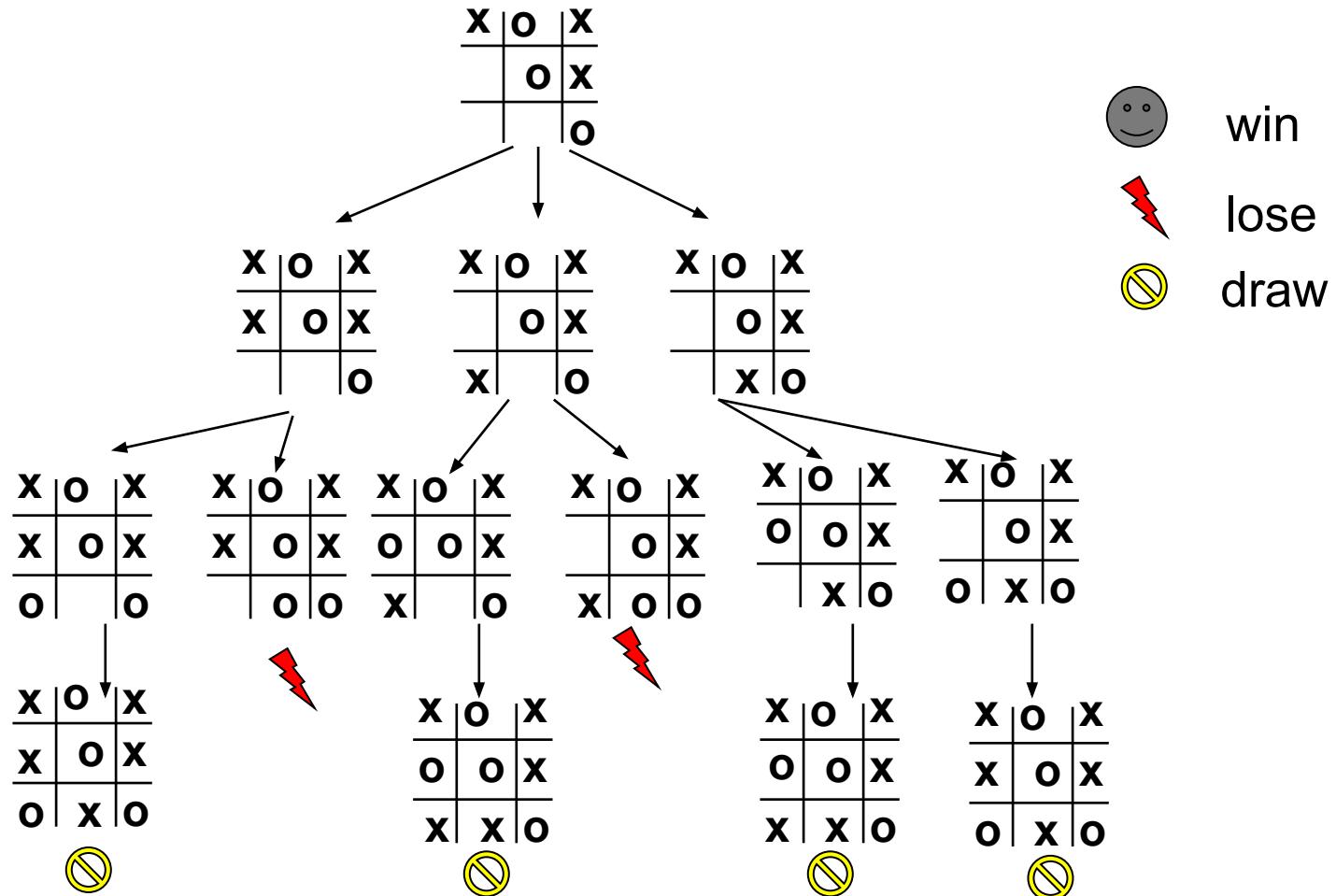
- When the problem can be expressed by a set of variables with constraints on their values
- When constraints are relatively simple (e.g., binary)
- When constraints propagate well (AC3 eliminates many values)
- Local Search: when the solutions are “densely” distributed in the space of possible assignments

Game Playing -- Adversarial Search

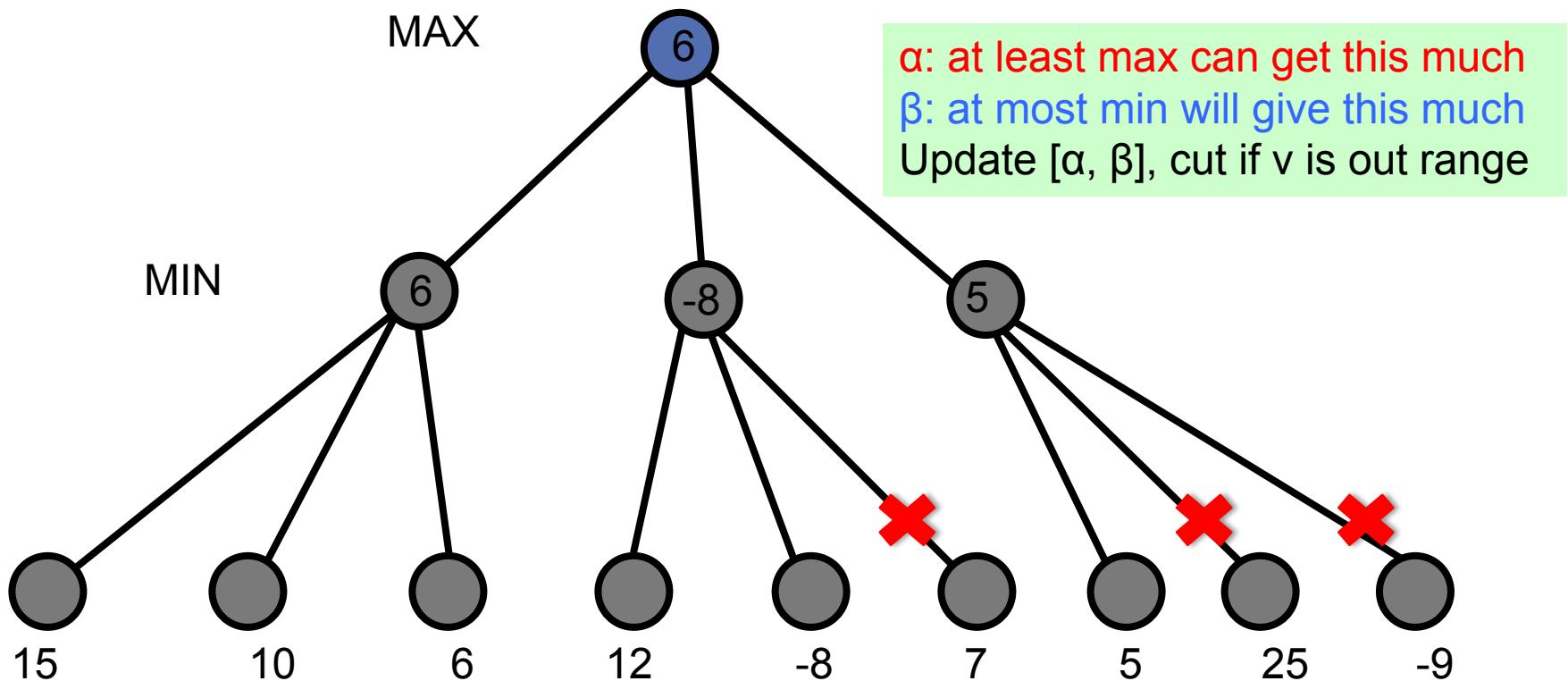
MAX

MIN

MAX

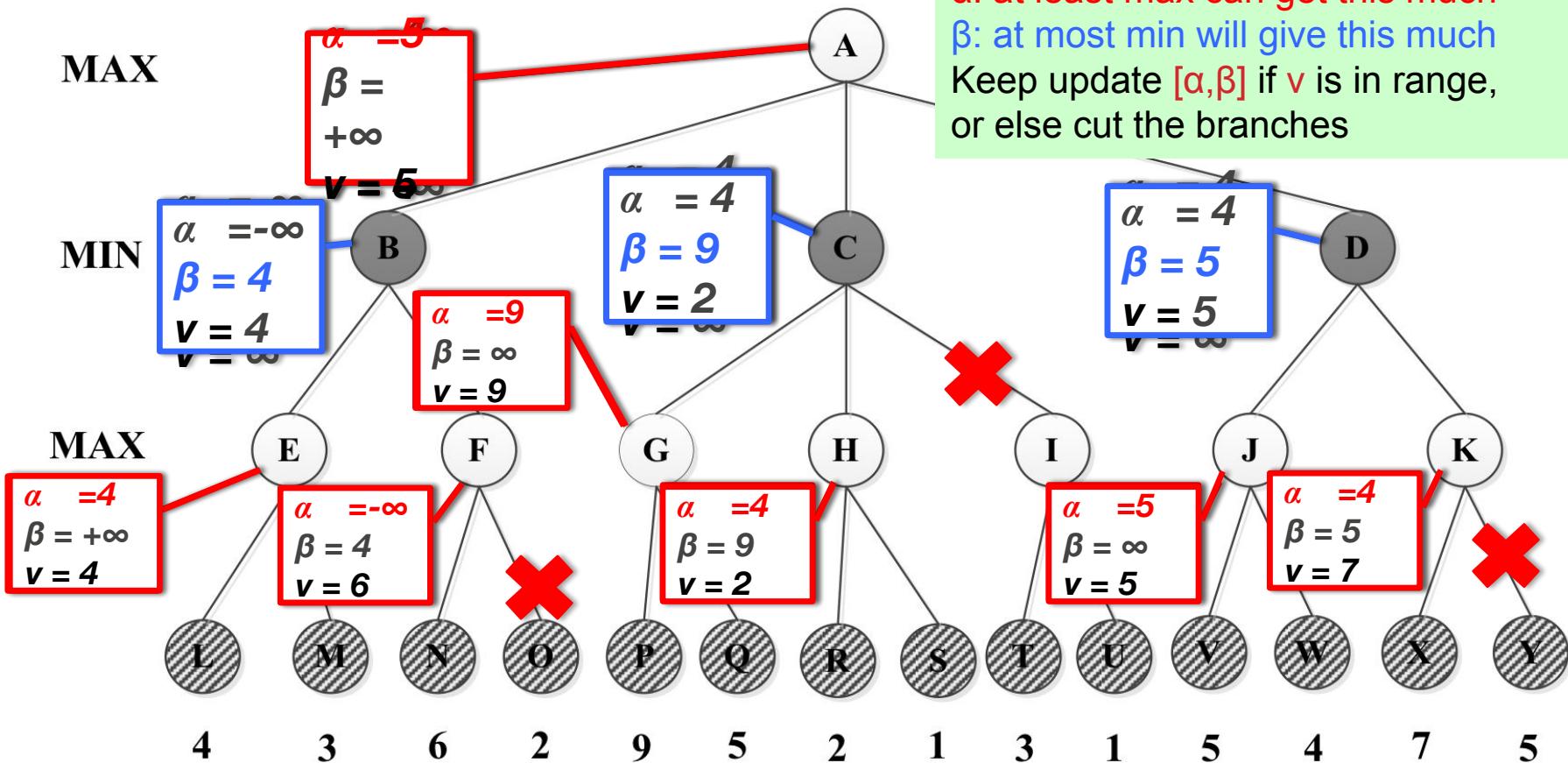


Minimax Algorithm



MAX

α : at least max can get this much
 β : at most min will give this much
 Keep update $[\alpha, \beta]$ if v is in range,
 or else cut the branches

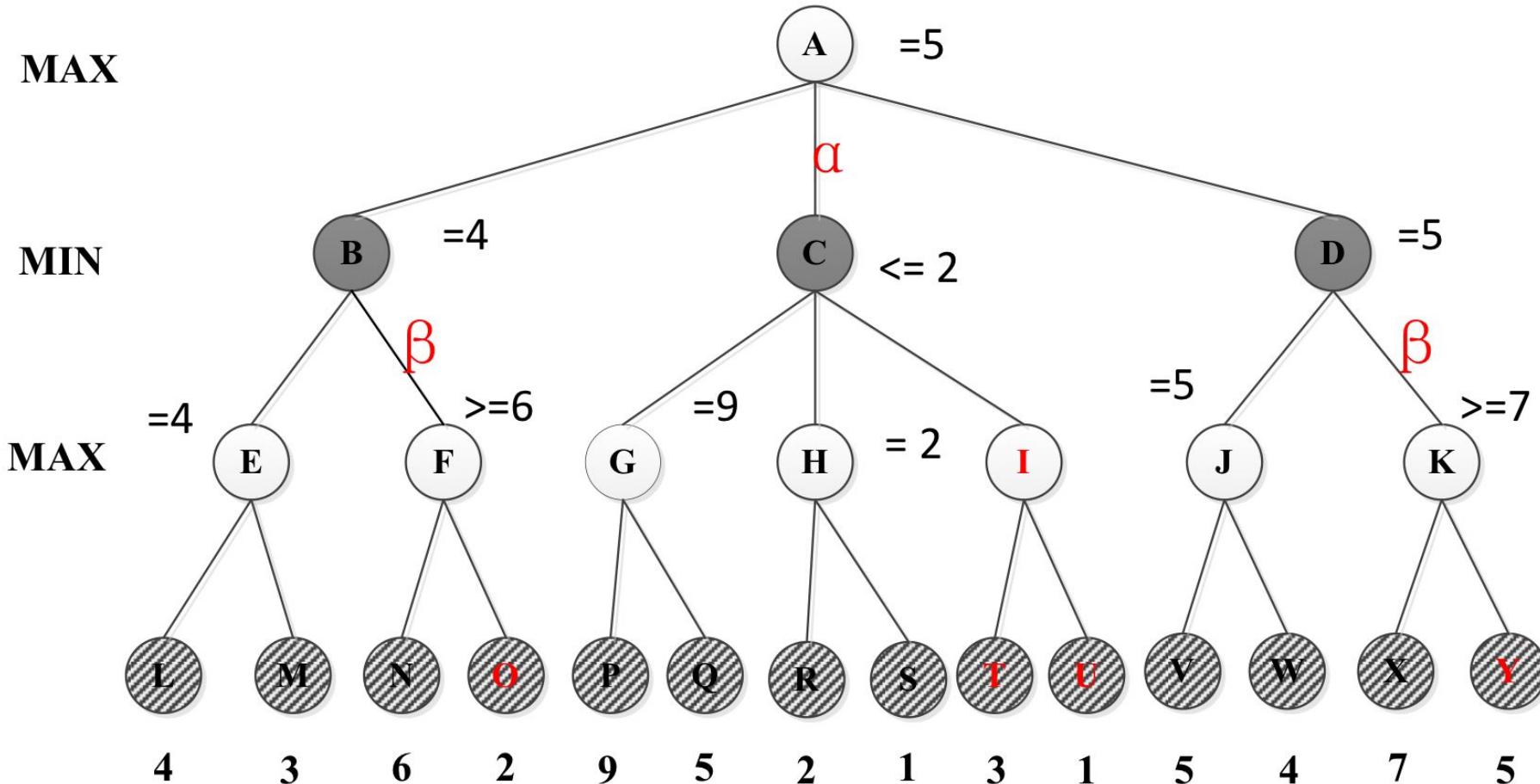


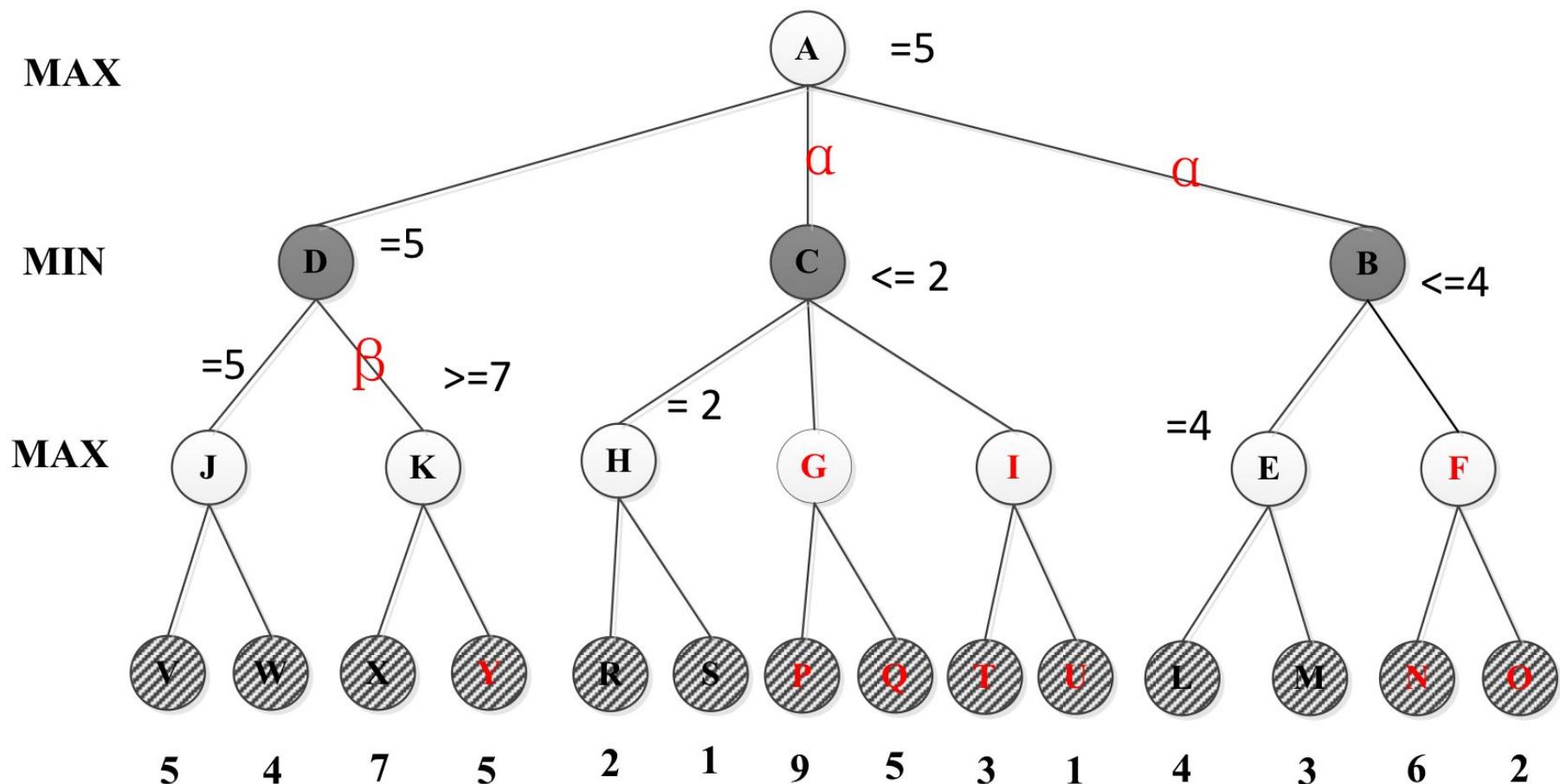
Max Node

```
for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if  $v \geq \beta$  then return v
    α ← MAX(α, v)
return v
```

Min Node

```
for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if  $v \leq \alpha$  then return v
    β ← MIN(β, v)
return v
```





What you should know

- What are the characteristics of local search? Is it complete? Optimal? Time and Space complexity?
- What problem domains are well-suited to each type of search technique? Ill-suited? Why?
- Know how to compare their performance in general and in a specific domain or problem.
- What is the difference between uninformed and informed search? Which ones are optimal?
- What are the advantages and disadvantages of depth-first search?
- Why does a search heuristic need to be “admissible”?
- What are the characteristics of Adversarial Search?
- Why is *meta-reasoning* important in adversarial search?

Want more?

Check out these demos:

<http://toddwschneider.com/posts/traveling-salesman-with-simulated-annealing-r-and-shiny/>

<http://codecapsule.com/2010/04/06/simulated-annealing-traveling-salesman/>

<http://www.biostat.jhsph.edu/~iruczins/teaching/misc/annealing/animation.html>

<http://math.hws.edu/eck/jsdemo/jsGeneticAlgorithm.html>

http://rednuht.org/genetic_walkers/

http://rednuht.org/genetic_cars_2/

Alpha-beta search demo:

<https://www.yosenspace.com/posts/computer-science-game-trees.html>

A* and heuristics:

<http://www.briangrinstead.com/files/astar/>

Practice Exercises: Chapter 4: # 4.1, Chapter 6: # 6.1, 6.5

CSCI 561

Foundation for Artificial Intelligence

07 - Reinforcement Learning

08 – Learn to Search and Play Games

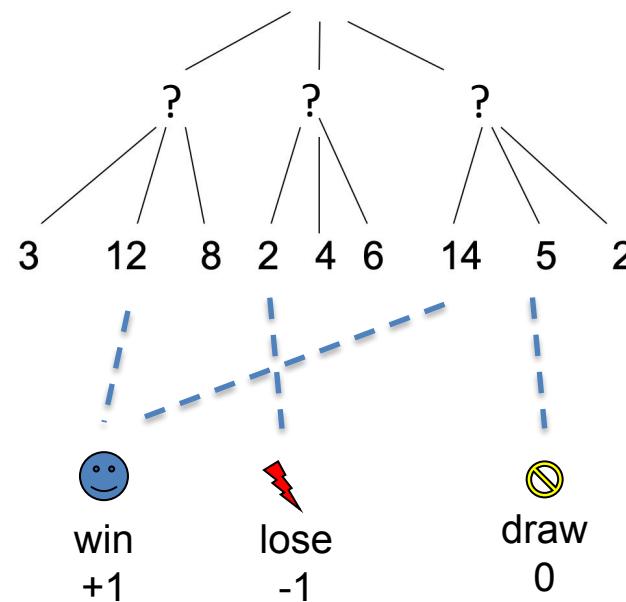
Professor Wei-Min Shen
University of Southern California

Outline

- Motivation
 - Agent and Environment (Search & Games)
- States, actions, utility, rewards, policy
- Utility value iteration
- Policy Iterations
- Reinforcement Learning
 - Model-based
 - Model-free
- Q-Learning
 - State space for advanced game playing

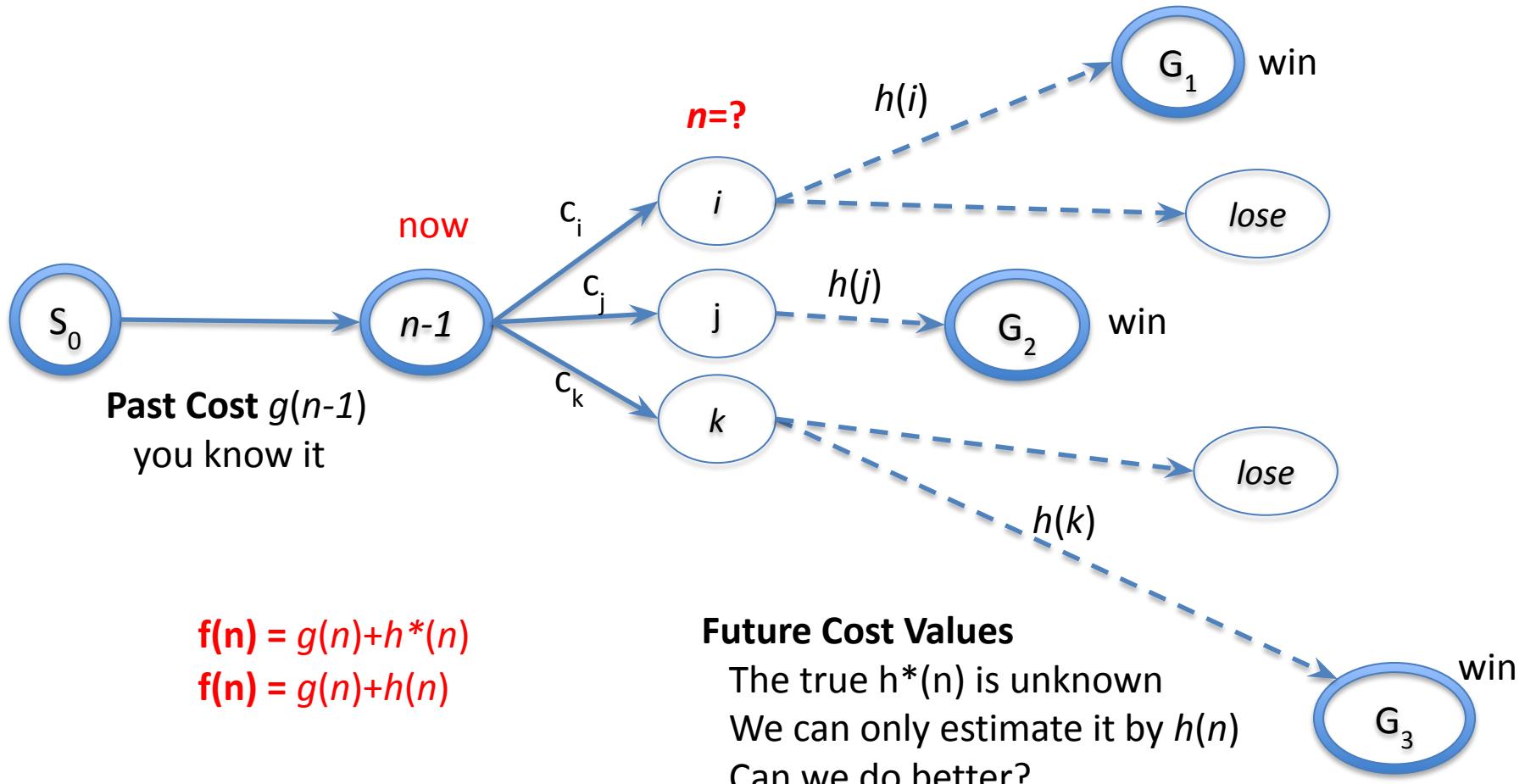
A Key Question

- In all search and game playing problems, what is the key information that you wish to have for finding the optimal solution?
 - What you wish to have but may not always have?

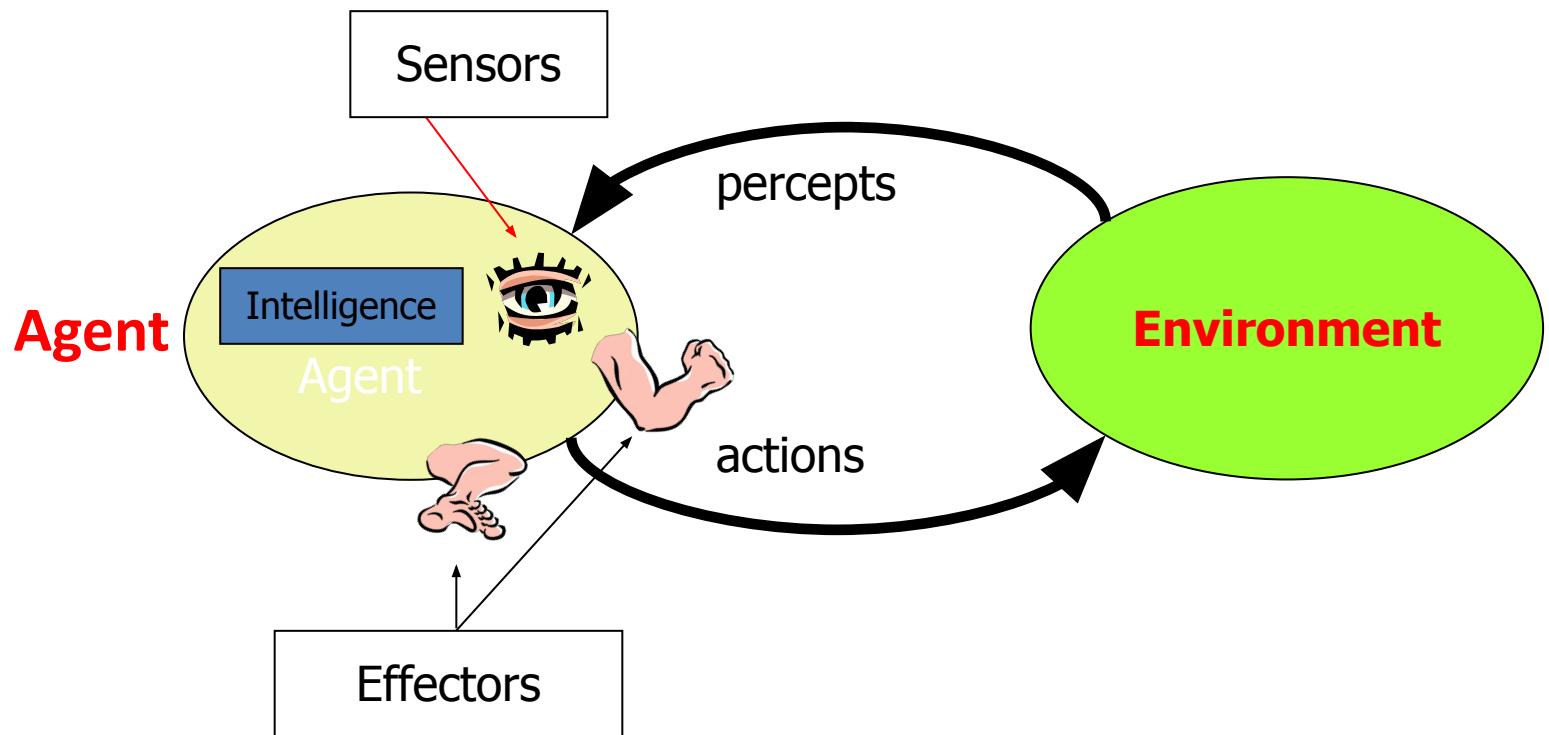


The True Future Values

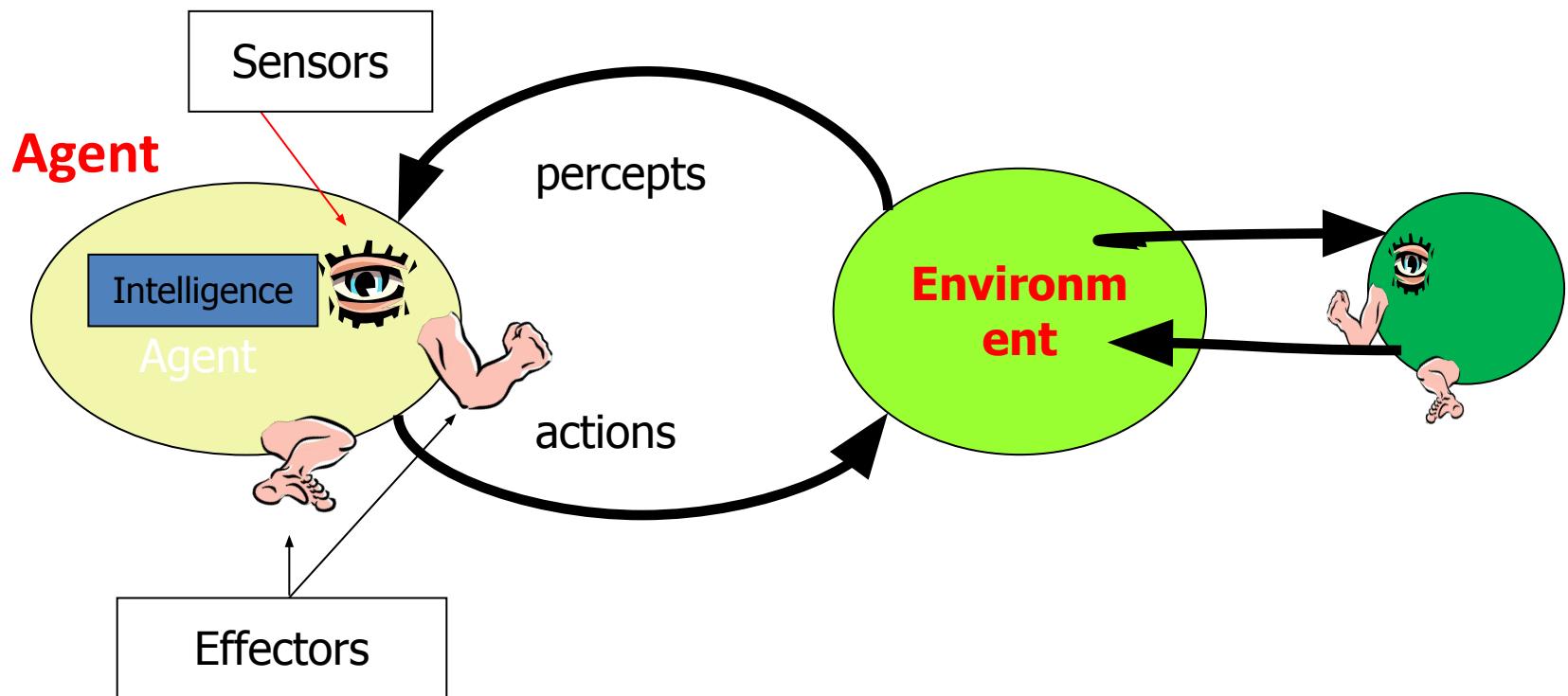
(Only if we would know them !)



Agent and Environment



Agent, Environment, Game



Agent and Environment (Star War)

1 	2	3	4 
5		6	7 
8	9	10	11

Agents:
R2D2
3PIO
Darth Vader

States: 1,..,11

Actions: ^,v,>,<

Percepts: ...

Goals: ...

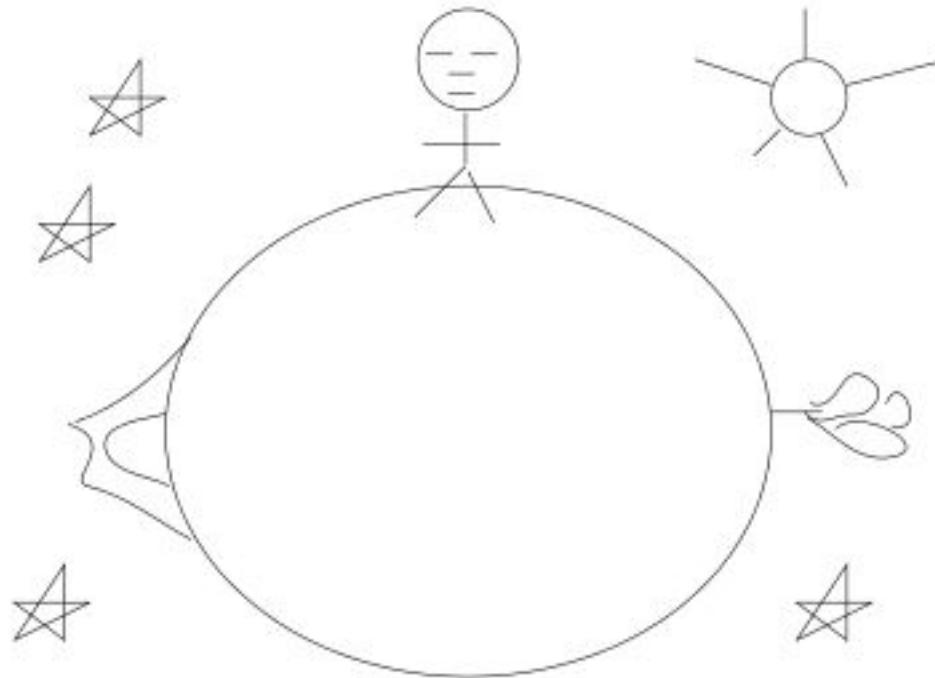
Rewards: ...

State Utility Values:
...
(your “crystal ball”)

Terminal States: 4 and 7. (nowhere to go next)

Agent and Environment

The Little Prince's Planet



- Percepts: {rose, volcano, nothing}
- Actions: {forward, backward, turn-around}
- States: {north, south, east, west}

The Little Prince Planet Environment

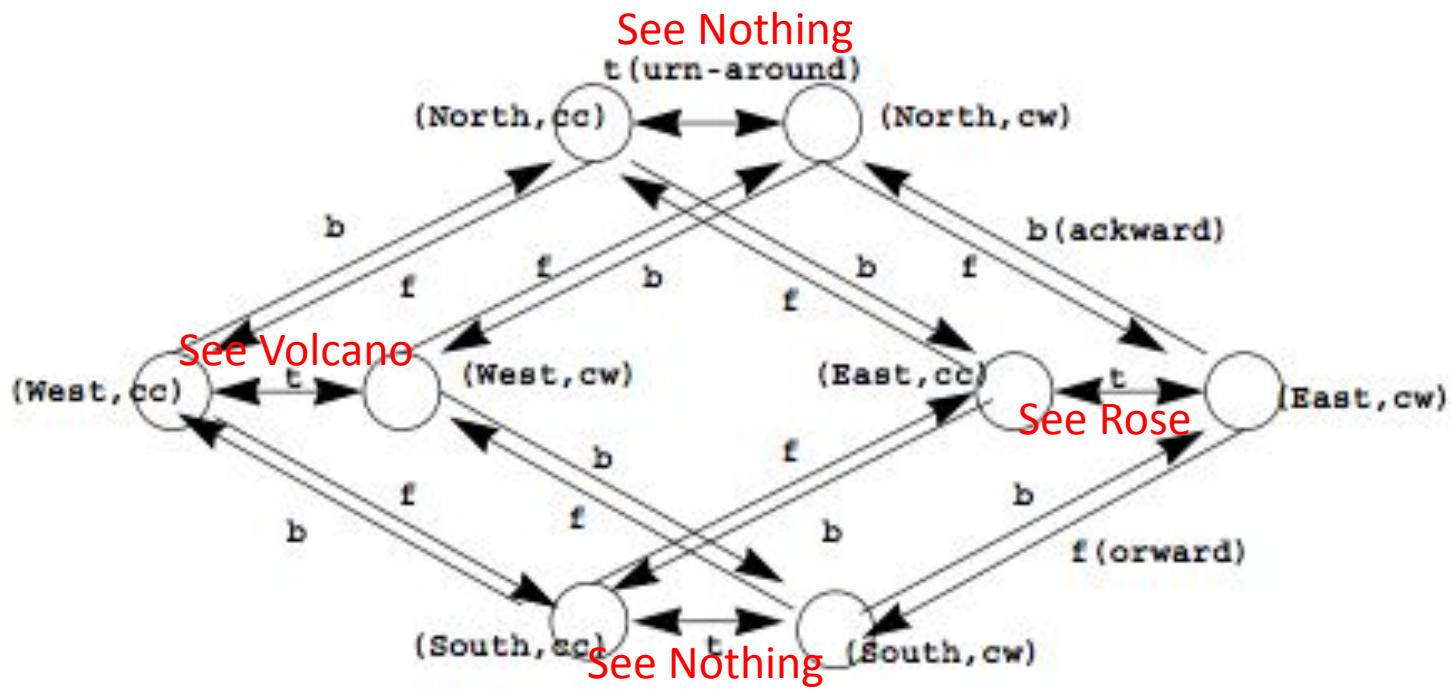


Figure 2.5: The little prince's actual environment.

A Model for Little Prince's Planet

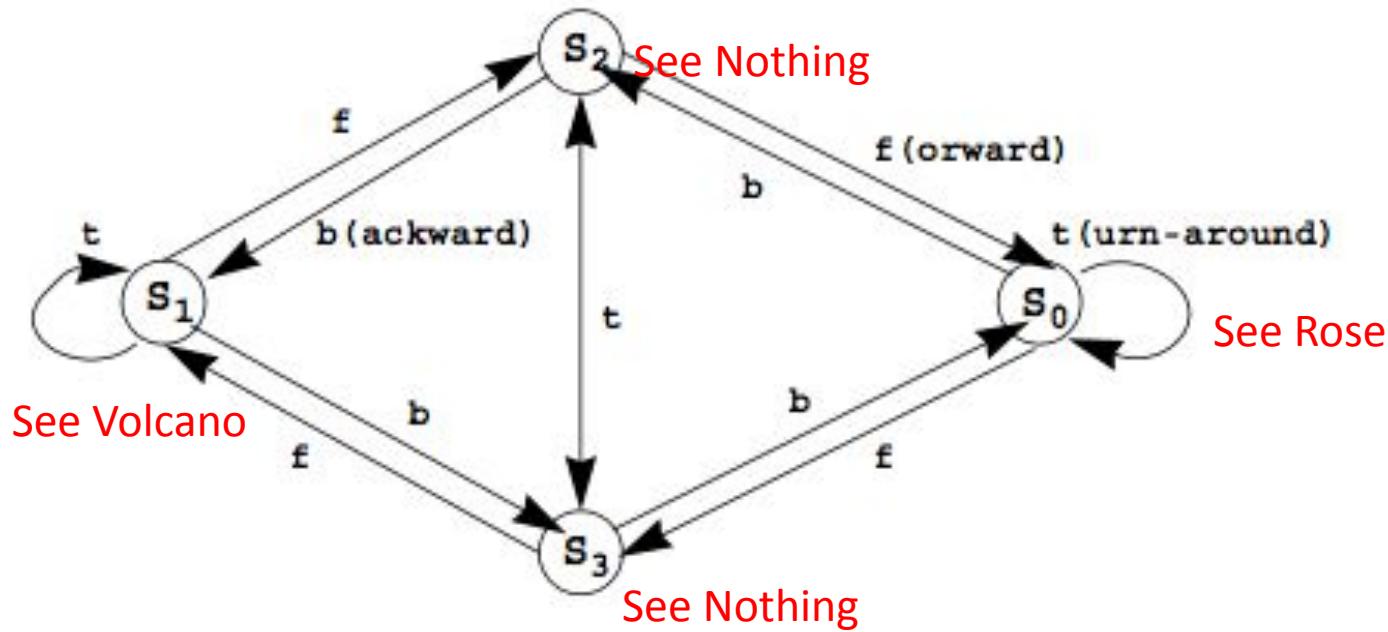


Figure 2.3: The little prince's model of his world.

His internal action model only needs 4 states! (why?)

State, Action and Sensor Models

We represent a model M of the environment \mathcal{E} as a machine $(A \ Z \ S, \phi, \theta, t)$, where

- A is a set of actions, which is the same as the set of input actions into \mathcal{E} ,
- Z is a set of percepts, which is the same as the set of output symbols of \mathcal{E} ,
- S is a set of model states,
- ϕ , a function from $S \times A$ to S , is the transition function of M ,
- θ , a function from S to Z is the appearance function of M , and
- t is the current model state of M . When a basic action a is applied to M , t is updated to be $\phi(t, a)$.

Little Prince's Model

$$A \equiv \{\text{forward, backward, turn-around}\}$$

$$P \equiv \{\text{rose, volcano}, \text{nothing}\}$$

$$Z \equiv \{s_1, s_2, s_3, s_4\}$$

$$\phi \equiv \phi(s_0, \text{forward}) = s_3, \phi(s_0, \text{backward}) = s_2, \dots$$

$$\theta \equiv \theta(s_1) = \{\text{volcano}\}, \theta(s_0) = \{\text{rose}\}, \theta(s_2) = \theta(s_3) = \{\}$$

Transition Model ϕ can be deterministic or probabilistic

Sensor Model θ can be deterministic or probabilistic

Are the transition model ϕ here deterministic?

Are the sensor model θ deterministic?

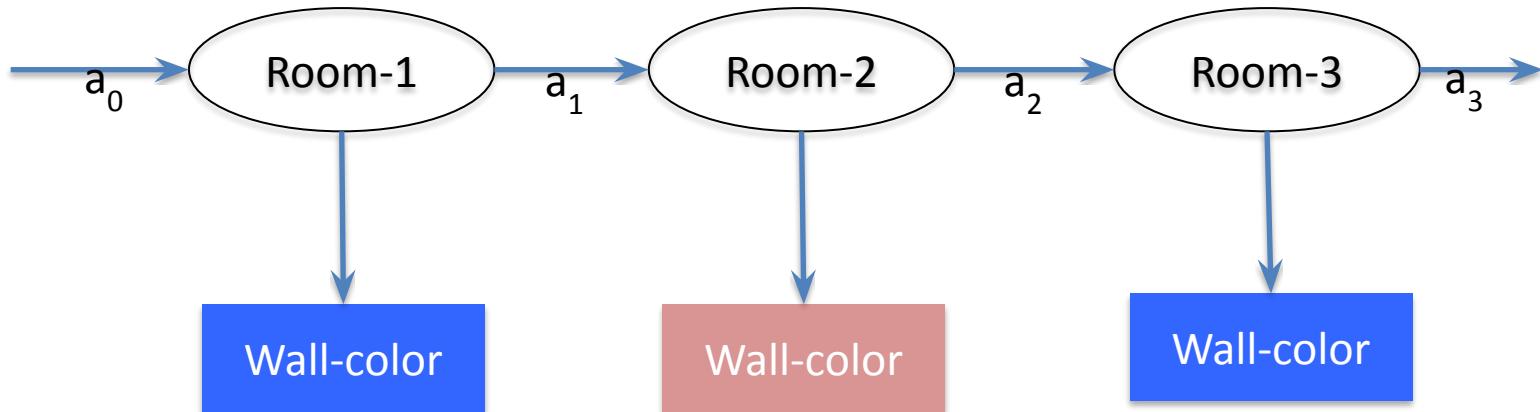
Is there any hidden states in this example?

The Environment may be Uncertain

- “Seeing may not always be believing”
 - States and observations are not always 1-1 mapping
 - One state may be seen in many different ways
 - The same observation may be seen in many different states
 - E.g., When the Prince sees nothing, where is he? Facing which way?
 - **Can you make the little prince’s sensors non-deterministically?**
- “Action may not always produce the same result”
 - Actions do not always take you to the same states
 - E.g., the Little Prince’s planet may have “planet-quakes” 😊
 - E.g., step forward on ice may not always succeed
 - Transitions between states by actions may be nondeterministic
 - E.g., Star War’s transitions are not deterministic
 - **Can you make the Little Prince’s actions non-deterministically?**

Sensors can be uncertain (in real world)

- Speech Recognition
 - “Listening is not always equal to hearing” 😊
- Traveling through rooms with colored walls
 - “Seeing does not always tell where you are” 😊

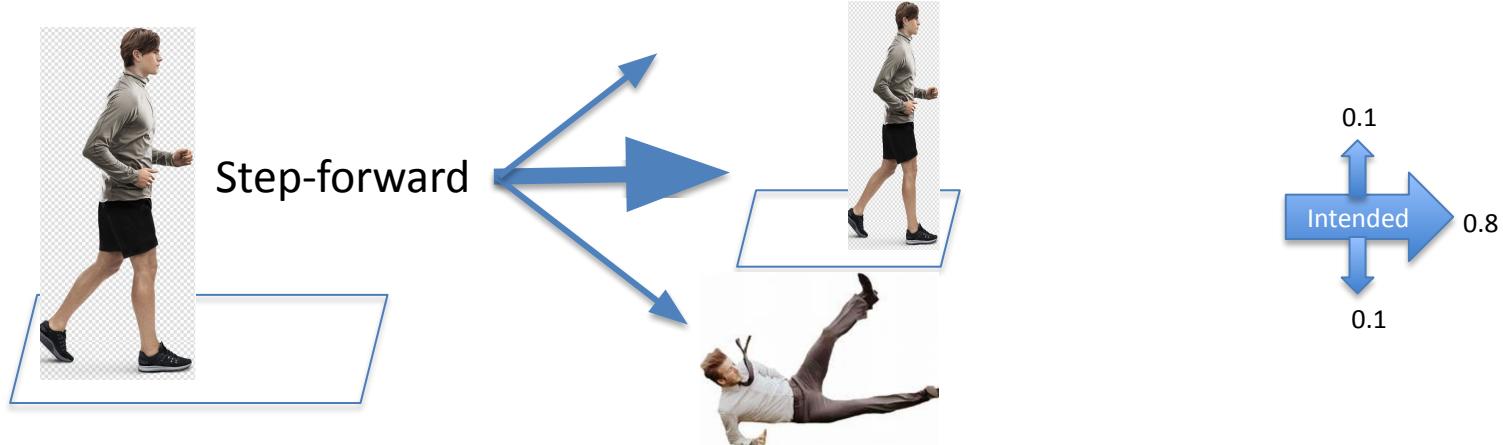


Actions can be uncertain (in real world)

- Deterministic actions



- Non-deterministic actions (e.g., walking on ice)



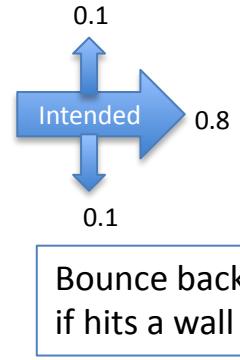
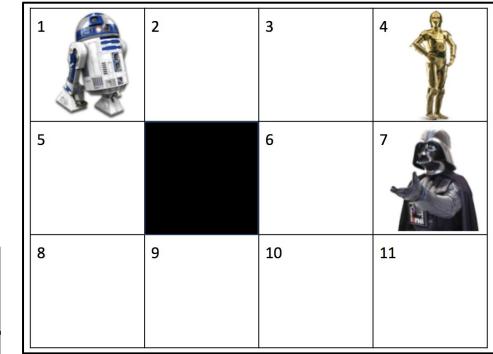
Other Uncertainties in the Real World

- Multiple Agents
 - The actions of other agents on the environment may be “uncertain” from your point of view !
- Advanced Game Playing
 - Your opponent’s moves are definitely “uncertain” from your point of view !
- When you driving on a highway
 - What are uncertain?

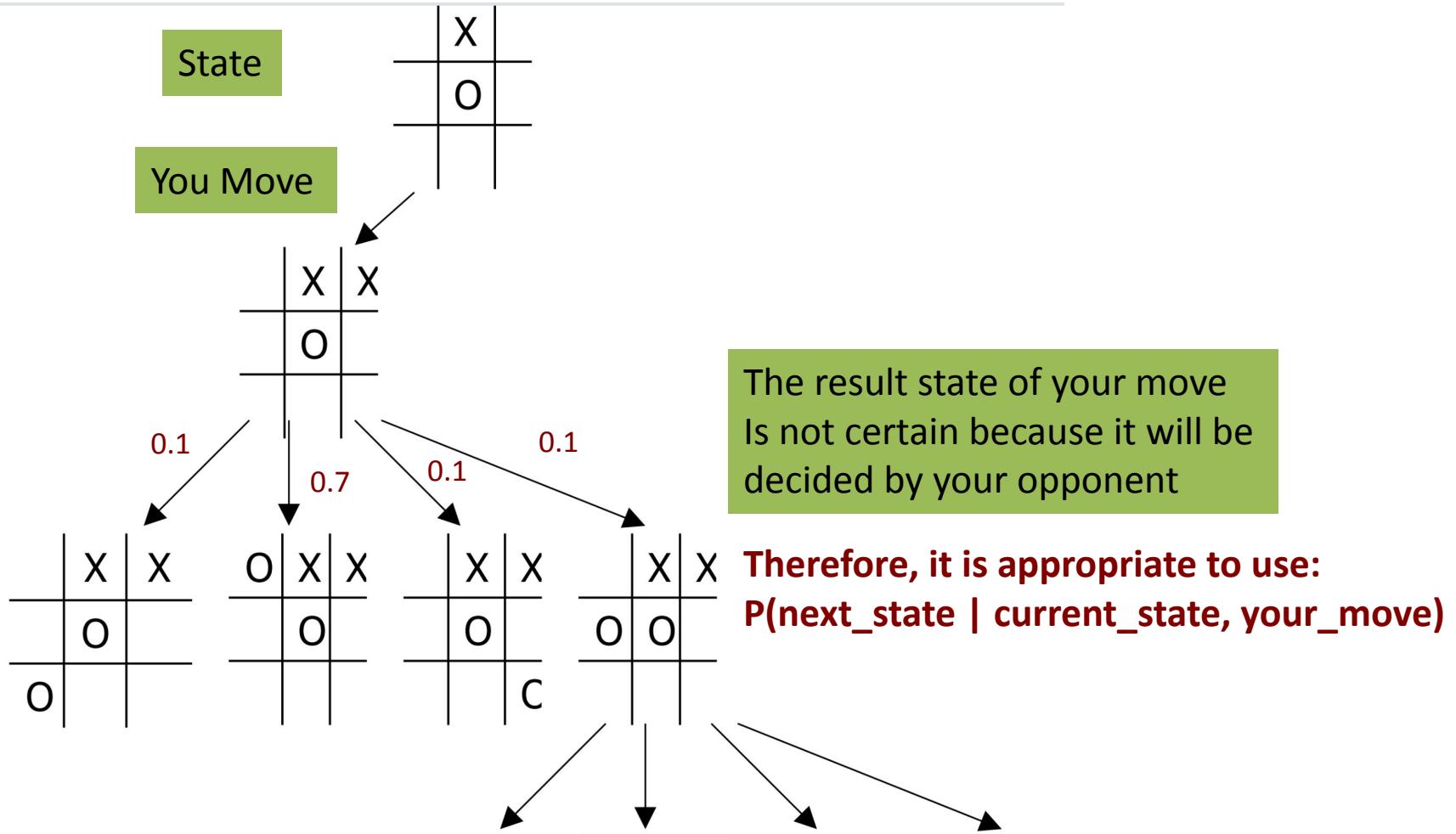
Star War's Probabilistic Transition Model

- Transitions can be probabilistic
 - $P(\text{state}_{t+1} \mid \text{state}_t, \text{action}_t)$

X_{t-1}	A_{t-1}	$P(X_t=1)$	$P(X_t=2)$	$P(X_t=3)$	$P(X_t=4)$	$P(X_t=5)$...
1	up	0.8+0.1	0.1	0.0	0.0	0.0	
1	down	0.1	0.1	0.0	0.0	0.8	
1	left	0.8+0.1	0.0	0.0	0.0	0.1	
1	right	0.1	0.8	0.0	0.0	0.1	
2	up	0.1	0.8	0.1	0.0	0.0	
...							



In Game Playing: Opponent's Actions are not certain

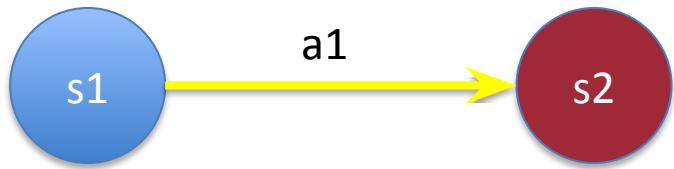


The Key Representations

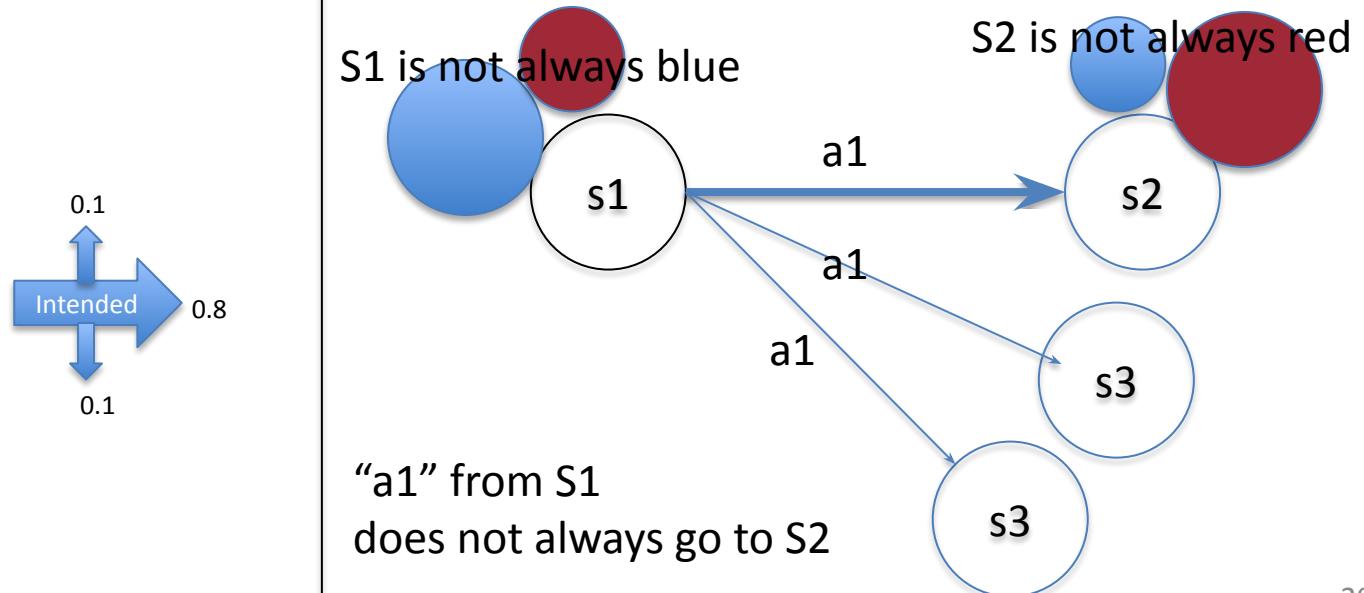
- Model the environment by **States, Actions, Percepts**, and
- Transition (action) model $\varphi = P(s_{t+1} | s_t, a_t)$ may be probabilistic
- **Sensor Model** $\theta = P(z | s)$ may be probabilistic
- States may have **Utility Value** $U(s)$ or $V(s)$
- Agents may receive **Reward r** (implicit “goals”) occasionally:
 - rewards may be given to agent in different format or time
- **Behaviors** may be represented as **Policy**: state -> action
- **Objective**: find the optimal policy based on utilities and rewards

Certainty vs. Uncertainty

The certain way

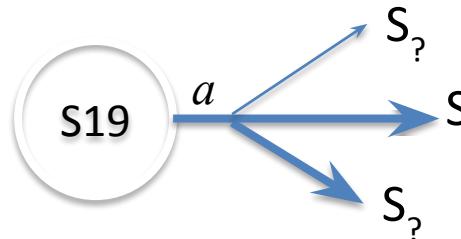


The uncertain way



Uncertainty in Sensor & Actions

State, action

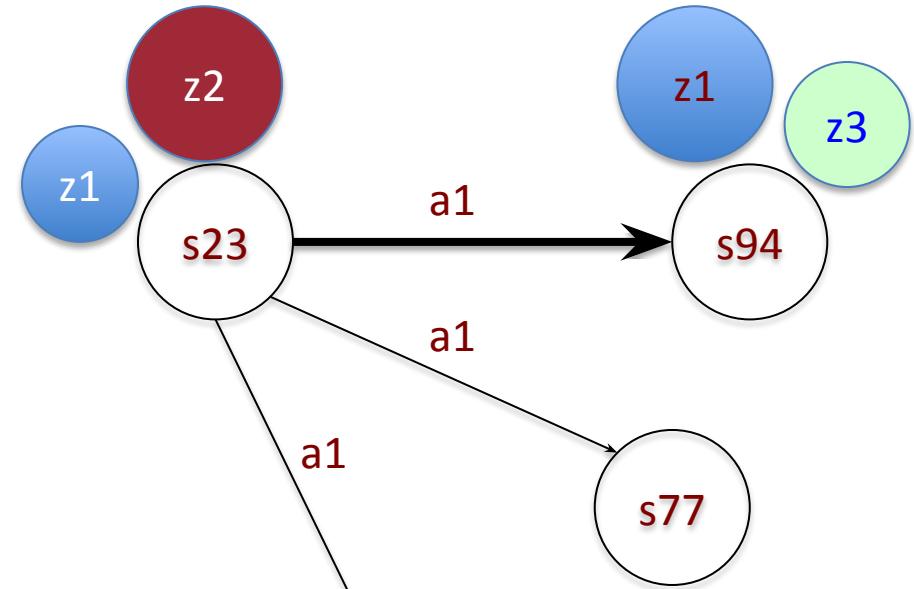


Observations



- Solution: let's use probabilities
 - Action/Transition Model (for states and actions)
Use Probabilistic Transitions
 $P(\text{NextState} \mid \text{CurrentState, Action})$
 - Sensor Model (for states and observations)
 $P(\text{Observations} \mid \text{State})$

Hidden Markov Model (with actions)



1. Actions
2. Percepts (observations)
3. States
4. Appearance: states -> observations
5. Transitions: (states, actions) -> states
6. Current State

Hidden Markov Model (1/2)

hidden Markov model M is a stochastic process $(B, Z, S, P, \theta, \pi)$ with the components defined as follows:

- B is a set of basic actions. As before, the actions can be applied to both the environment and the model.
- Z is a set of percepts and represents the output symbols of the environment that can be observed by the learner.
- S is a finite set of (internal) model states. We assume that at any single instant t , the current environmental state q_t corresponds to exactly one model state s_t , and the identity of s_t is sufficient to stochastically determine the effects of action in the environment. This is the *Markov assumption*.
- $\phi = \{P_{ij}[b]\}$ is a set of probabilities concerning model state transitions. For each basic action $b \in B$ and a pair of model states s_i and $s_j \in S$, the quantity $P_{ij}[b]$ specifies the probability that executing action b when the current model state is s_i will move the environment to an environmental state that corresponds to the model state s_j .

Action Model

Hidden Markov Model (2/2)

Sensor Model

- $\theta = \{\theta_i(k)\}$, where $\theta_i(k) = p(z_k|s_i)$, $z_k \in Z$, and $s_i \in S$, is a set of probability distributions of observation symbols in each model state. (This corresponds to the appearance function for the deterministic models defined at the beginning of this chapter.) For each observation symbol z_k , the quantity $\theta_i(k)$ specifies the probability of observing z_k if the current model state is s_i .
- $\pi(t) = \{\pi_i(t)\}$ is the probability distribution of the current model state at time t . That is, $\pi_i(t) = p(i_t = s_i)$, where i_t denotes the current model state and $s_i \in S$ specifies the probability of s_i being the current model state at time t .
This is also called “localization”

The HMM for Little Prince's Planet

(with uncertain actions and sensors)

$$A \equiv \{\text{forward, backward, turn-around}\} \quad \{f, b, t\}$$

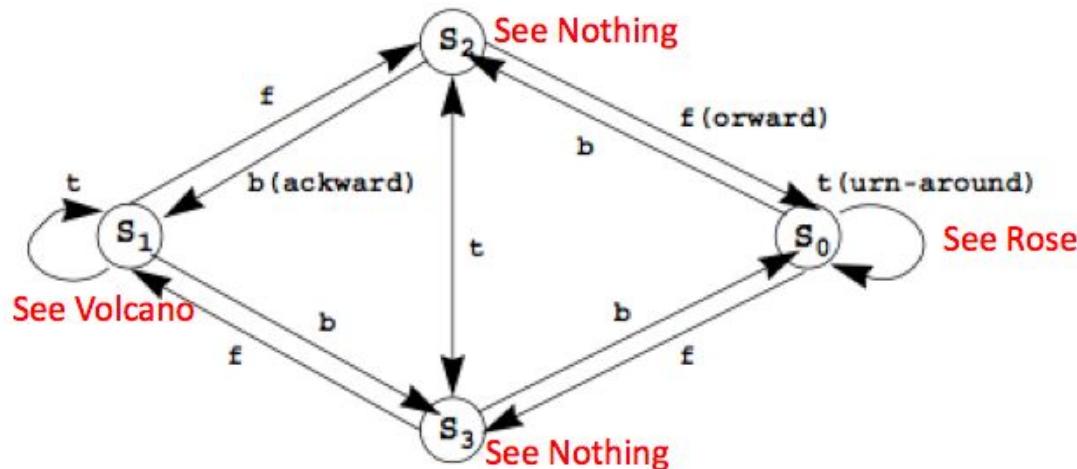
$$Z \equiv \{\text{rose, volcano, nothing}\}$$

$$S \equiv \{s_1, s_2, s_3, s_4\}$$

$$\phi \equiv \{P(s_3|s_0, f) = .51, P(s_2|s_1, b) = .32, P(s_4|s_3, t) = .89, \dots\}$$

$$\theta \equiv \{P(\text{rose}|s_0) = .76, P(\text{volcano}|s_1) = .83, P(\text{nothing}|s_3) = .42, \dots\}$$

$$\pi_1(0) = 0.25, \pi_2(0) = 0.25, \pi_3(0) = 0.25, \pi_4(0) = 0.25$$



Little Prince Example (may add Rewards)

- (action) Transition Probabilities φ (Fwd, Back, Turn)

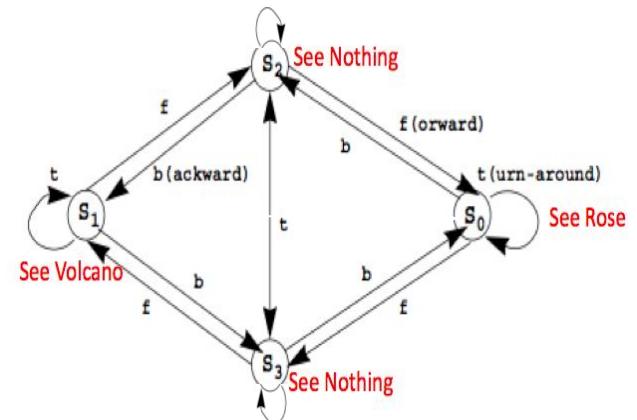
F	S0	S1	S2	S3	B	S0	S1	S2	S3	T	S0	S1	S2	S3
S0	0.1	0.1	0.1	0.7	S0	0.1	0.1	0.7	0.1	S0	0.7	0.1	0.1	0.1
S1	0.1	0.1	0.7	0.1	S1	0.1	0.1	0.1	0.7	S1	0.1	0.7	0.1	0.1
S2	0.7	0.1	0.1	0.1	S2	0.1	0.7	0.1	0.1	S2	0.1	0.1	0.1	0.7
S3	0.1	0.7	0.1	0.1	S3	0.7	0.1	0.1	0.1	S3	0.1	0.1	0.7	0.1

- (sensor) Appearance Probabilities θ

θ	Rose	Volcano	Nothing
S0	0.8	0.1	0.1
S1	0.1	0.8	0.1
S2	0.1	0.1	0.8
S3	0.1	0.1	0.8

- Initial State Probabilities π

π	S0	S1	S2	S3
	.25	.25	.25	.25



You may add rewards to states (e.g., prefer to see rose)

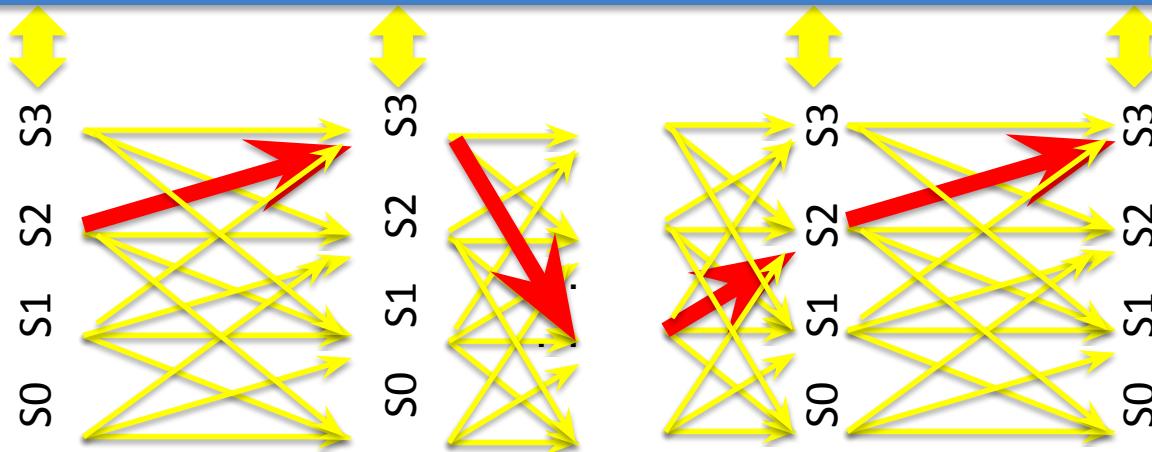
Experience and State Sequence

- $E_{1:T}$ is an experience which is a sequence of
 - {observe₁, act₁, observe₂, act₂, ..., act_{T-1}, observe_T}
 - $E_{1:T} = \{o_1, a_1, o_2, a_2, o_3, \dots, o_{T-1}, a_{T-1}, o_T\}$
- $X_{1:T}$ is a sequence of states that may be hidden but correspond to the experience
 - $X_{1:T} = \{X_1, X_2, X_3, \dots, X_{T-1}, X_T\}$

Little Prince Example

- From an “*experience*” (time₁ through time_t)
- Infer the most likely “*sequence of states*”

{rose}, forward, {.}, ..., turn, ..., {rose}, bwd, {volcano}

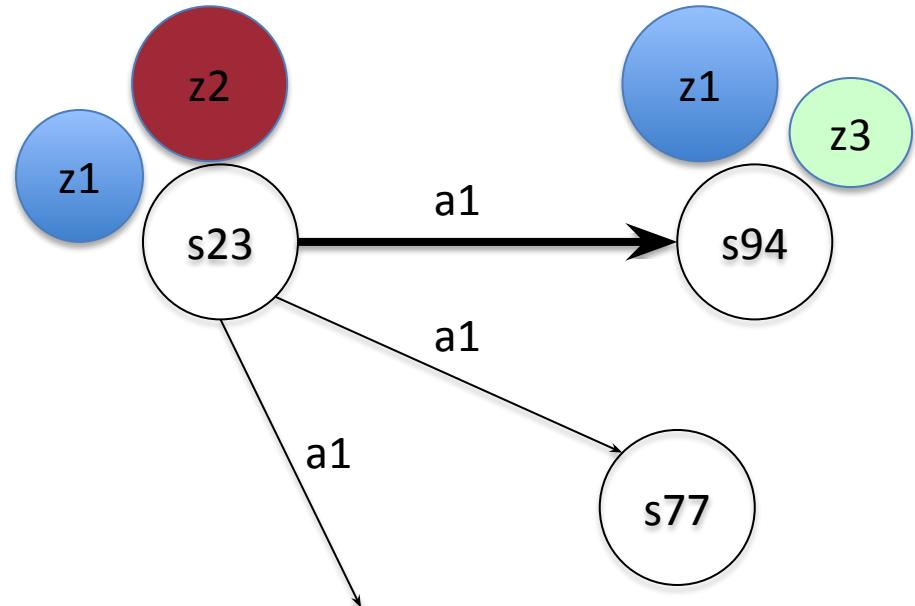


Find the “best sequence of (hidden) states” that support the experience!

Action and Sensor Models (review)

1. Actions
2. Percepts (observations)
3. States
4. Appearance: states -> observations
5. Transitions: (states, actions) -> states
6. Current State

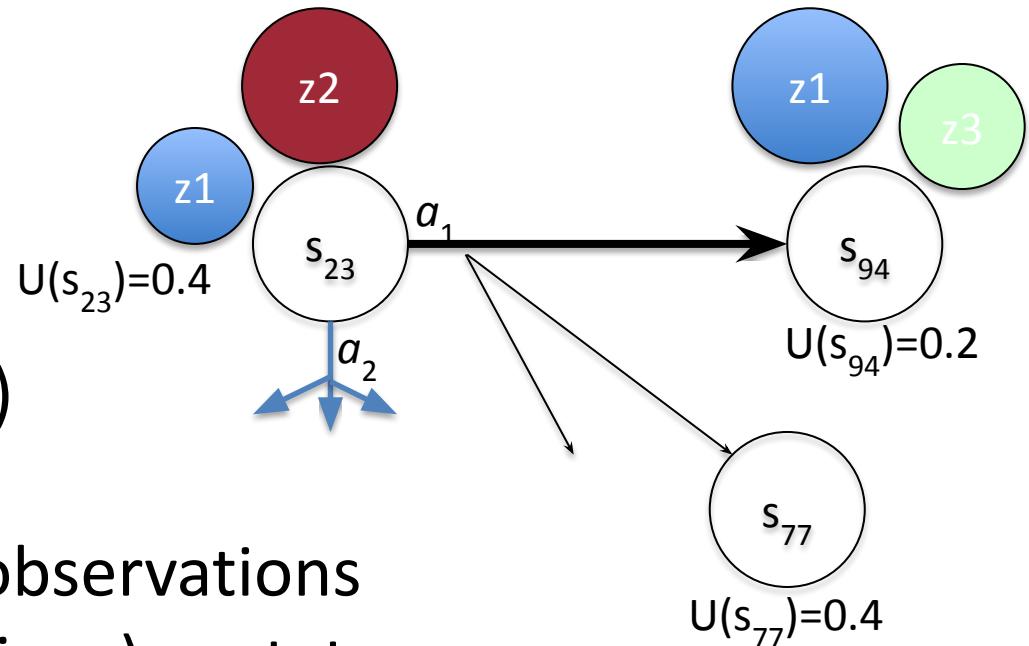
What about the goals?



Utility Value of States

Utility Value \Leftrightarrow Goal Information

1. Actions
2. Percepts (observations)
3. States
4. Appearance: states -> observations
5. Transitions: (states, actions) -> states
6. Current State
7. **Rewards:** $R(s)$ or $R(s,a)$ (related to the goals, given to the agent)
- (8) Utility Value of States:** $U(s)$ (how good for the goals, must compute)



Rewards and Utilities

- Three types of rewards for agents
 1. Being at a state $R(s)$, e.g., holding an ice cream
 2. Do an action on a state $R(s, a)$, e.g., eating the ice cream
 3. Making a transition $R(s, a, s')$, e.g., feeling good after eating
- Every state may have a Utility Value $U(s)$ or $V(s)$

1 	2	3	4 
5		6	7 
8	9	10	11

Rewards and Utility for R2D2:

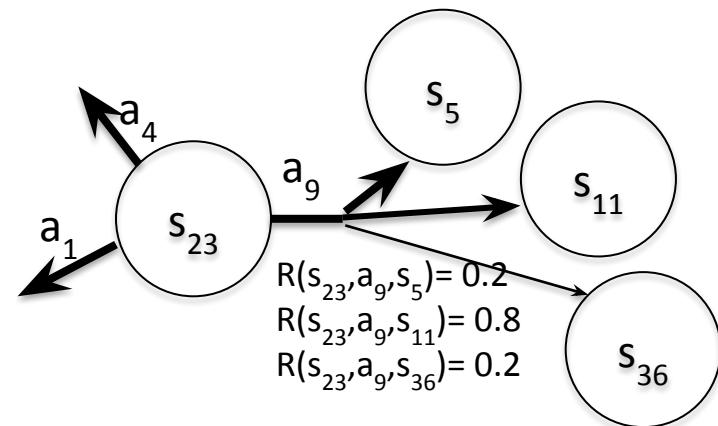
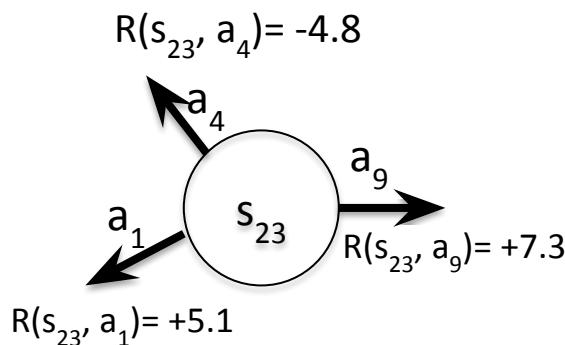
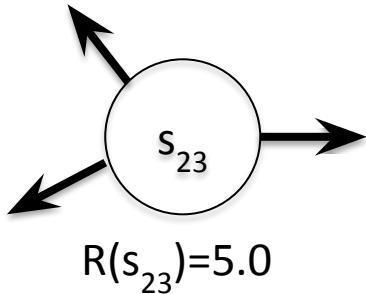
X_t	Reward	Utility
4	+1	0.0
7	-1	0.0
Else	-0.04	0.0

Given

To be learned

Rewards (caution, 3 types)

- Three types of rewards for agents
 1. Being at a state $R(s)$, e.g., holding an ice cream
 2. Do an action on a state $R(s, a)$, e.g., eating the ice cream
 3. Making a transition $R(s, a, s')$, e.g., feeling good after eating



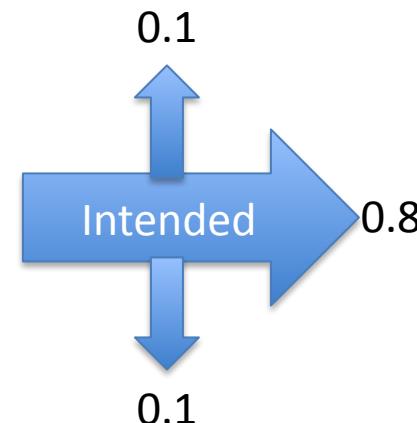
Type I: $R(s)$

Type II: $R(s, a)$

Type III: $R(s, a, s')$

Reward and Action Example

-0.04	-0.04	-0.04	+1
-0.04		-0.04	-1
-0.04 (start)	-0.04	-0.04	-0.04



(Except for the terminal/goal states)

Rewards (not utility) as shown:

Two terminal states: -1, +1 (goal)

-0.04 for all nonterminal states

Actions: >, <, ^, v,

outcome probability:

0.8 for the intended

0.2 sideways

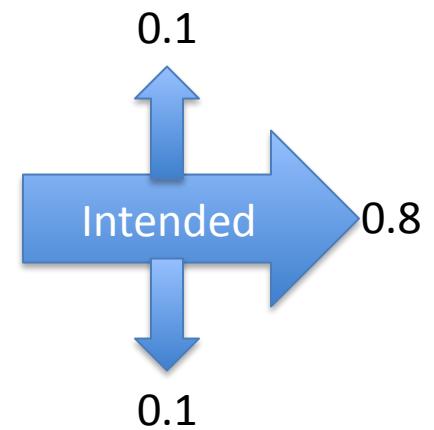
// bounce back if hits wall

// In a terminal state, the probability

// to go anywhere is 0.0

State Utility Value Example

?	?	?	?
?		?	?
?	?	?	?

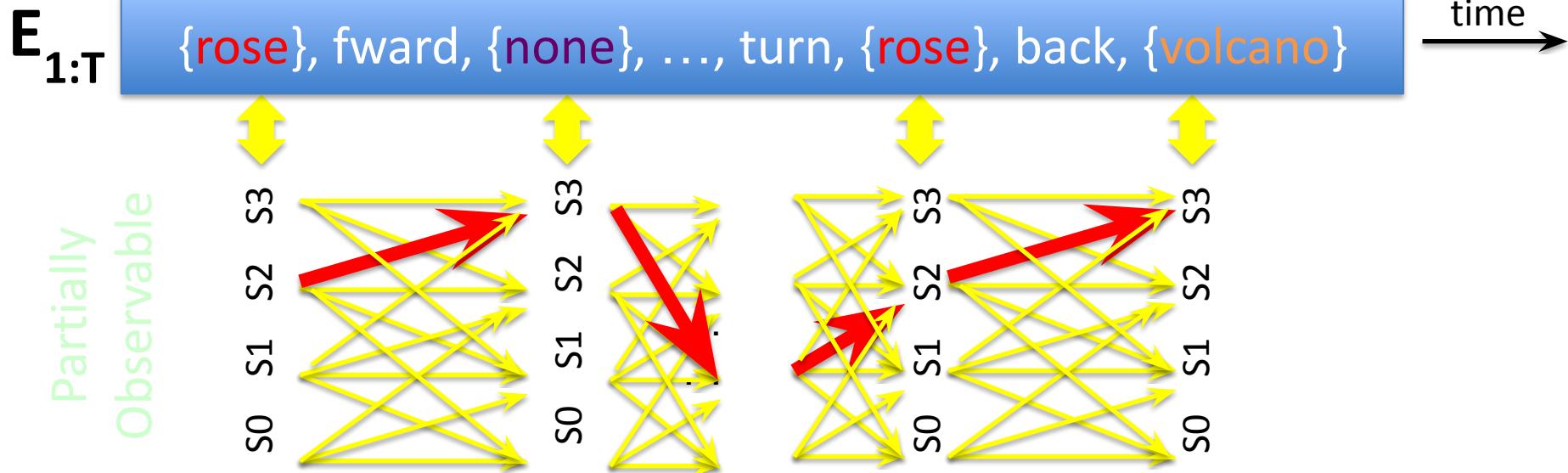


State Utility Values:

Must be learned or computed by the agent

Initially, could be all 0.0

Little Prince in Action (POMDP)



- Given: “experience” $E_{1:T}$ (time 1 through T)
- Definitions:
 - State S and actions A (Forward, Back, Turn)
 - (Action model) Transition Probabilities Φ
 - (Sensor model) Appearance Probabilities Θ (rose, volcano, none)
 - (localization) Initial/current State Probabilities π
- Partially Observable: agent sees only the percepts, not the states

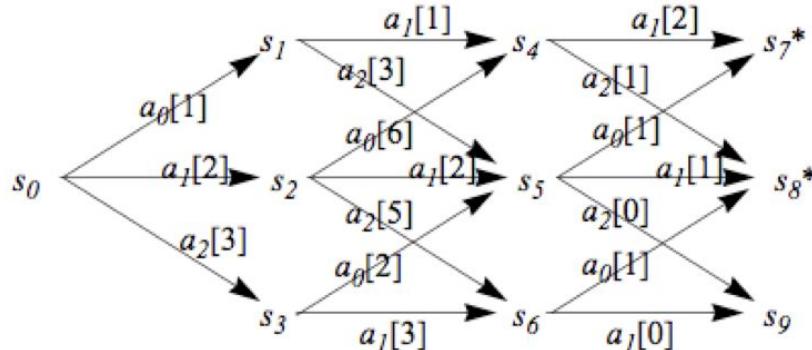
State Utility and Decision on Actions

- Combine the following together
 - Partially observable action/perception model
 - E.g., Little Prince Example
 - Compute state utility values from rewards (goals)
 - E.g., use Dynamic Programming (next slide)
 - Bellman equations, Bellman iterations (later slides)
 - Policy (decide actions based on state utility values)
 - To gain as much as rewards as you can
 - To go to the goals as close as you can

Compute State Values by Dynamic Programming

(review, from ALFE 6.1.1)

Backward recursion: compute utility/values by backing from the goal* stage by stage



Rewards are given
 $a_i[R] = R(s, a_i)$ from
an action a_i on a state s

are recorded. For example, the best value one can get from state s_4 to a goal state is 2, denoted as $V(s_4) = 2$. Similarly, $V(s_5) = 1$ and $V(s_6) = 1$. These

$$V(s_1) = \max\{R(s_1, a_1) + V(s_4), R(s_1, a_2) + V(s_5)\} = \max\{1 + 2, 3 + 1\} = 4$$

$$\begin{aligned} V(s_2) &= \max\{R(s_2, a_0) + V(s_4), R(s_2, a_1) + V(s_5), R(s_2, a_2) + V(s_6)\} \\ &= \max\{6 + 2, 2 + 1, 5 + 1\} = 8 \end{aligned}$$

$$V(s_3) = \max\{R(s_3, a_0) + V(s_5), R(s_3, a_1) + V(s_6)\} = \max\{2 + 1, 3 + 1\} = 4$$

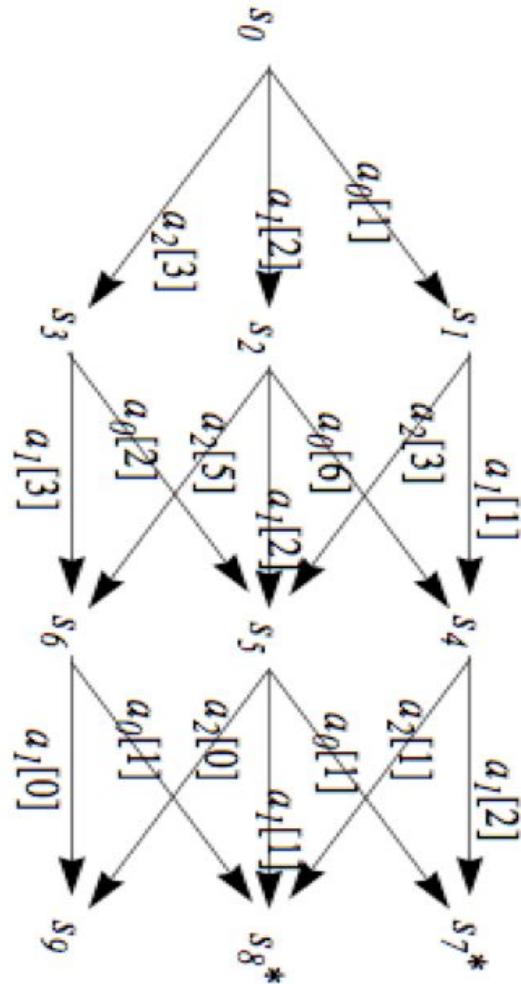
$$\begin{aligned} V(s_0) &= \max\{R(s_0, a_0) + V(s_1), R(s_0, a_1) + V(s_2), R(s_0, a_2) + V(s_3)\} \\ &= \max\{1 + 4, 2 + 8, 3 + 4\} = 10 \end{aligned}$$

Backward recursion equation: $V(s_i) = \max_a \{R(s_i, a) + V(s_j)\}$ where $s_i \xrightarrow{a} s_j$

Compute State Values by Dynamic Programming

(review, from ALFE 6.1.1)

Backward recursion: compute utility/values by backing from the goal* stage by stage



Markov Decision Process (MDP)

- A MDP consists of
 - State S and actions A
 - Initial State s_0 Probability Distribution Π
 - Transition Model $\Phi(s'|s,a)$
 - ~~– Sensor Model $\theta(z|s)$~~
 - A reward function $R(s)$

Note: A typical MDP has no sensor model

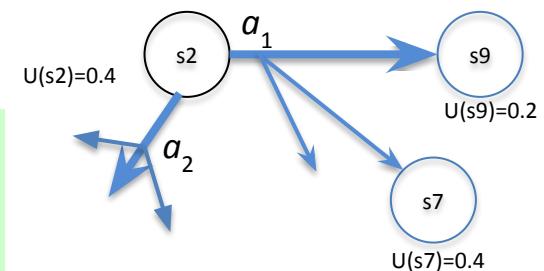
Maximum Expected Utility (MEU) and Rational Agents

- Every state has a utility value $U(s)$
- The expected utility of an action given the current evidence or observation e , is the average utility value of the outcomes, weighted by the probability that the outcome occurs:

$$EU(a | e) = \sum_{s'} P(result(a) = s' | a, e) U(s')$$

- The principle of maximum expected utility (MEU) is that a **rational agent** should choose the action that maximizes its expected utility:

$$action = \arg \max_a EU(a | e)$$



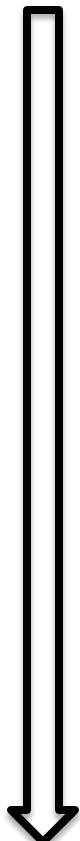
POMDP: Sensor Model, Belief States

- A Partially Observable MDP (POMDP) is defined as follows:
 - A set of states S (with an initial state s_0)
 - A set of Actions: $A(s)$ of actions in each state s
 - A transition model $P(s'|s,a)$, or $T(s,a,s')$
 - A reward function $R(s)$, or $R(s,a)$
 - A sensor model $P(e|s)$ □ □
 - A belief of what the current state is $b(s)$ □ □
- Belief States (where am I now? What is my current state?):
 - If $b(s)$ was the previous belief state, and the robot does action “ a ” and then perceives a new evidence “ e ”, then the new belief state:

$$b'(s') = \alpha P(e|s') \sum P(s'|s,a)b(s)$$

where α is a normalization constant making the belief states sum to 1

Goals, Rewards, Utilities, Policies



- Goals
 - Given to the agent from the problem statements
- Rewards
 - Given to the agent, designed based on the goals
- Utility values for states
 - Computed by the agent, based on the rewards
- Policies
 - Computed or learned by the agent
 - Used by the agent to select its actions
 - The better a policy, the more rewards it collects

Compute Utilities from Rewards over Time

- Utility Value = sum of all future rewards
 - Add the rewards as they are

$$U_h = ([s_0, s_1, s_2, \dots]) = R(s_0) + R(s_1) + R(s_2) + \dots$$

- Discount the far-away rewards in the future

$$U_h = ([s_0, s_1, s_2, \dots]) = R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \dots$$

- The expected future utility value $U^\pi(s)$ obtained by executing a policy π starting from s

$$U^\pi(s) = E \left[\sum_{t=0}^{\infty} \gamma^t R(s_t) \right]$$

The Optimal Policy

$$\pi_s^* = \arg \max_{\pi} U^\pi(s)$$

Compute Utilities (example)

-0.04	-0.04	-0.04	+1
-0.04		-0.04	-1
-0.04 (start)	-0.04	-0.04	-0.04

Rewards (given)



0.812	0.868	0.918	+1
0.762		0.660	-1
0.705	0.655	0.611	0.388

Utilities (computed)

Given the rewards in all nonterminal state are $R(s) = -0.04$,
Compute utilities using a future discount $\gamma = 1$
(we will see how these utilities are computed)

State Utility Value Iteration

(improving $U(s)$ every step)

- $U(s)$: the expected sum of maximum rewards achievable starting from a particular state s
- Bellman equations:
 - For n states, there are n equations must be solved *simultaneously*

$$U^*(s) = R(s) + \gamma \max_a \sum_{s'} T(s, a, s') U^*(s') \quad (17.5)$$

- Bellman iteration:
 - Converge to $U^*(s)$ step by step

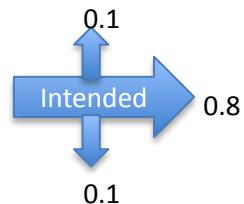
$$U_{i+1}(s) \leftarrow R(s) + \gamma \max_a \sum_{s'} T(s, a, s') U_i(s') \quad (17.6)$$

Bellman Iteration Example

Reward $R(s)$

-0.04	-0.04	-0.04	+1
-0.04		-0.04	-1
-0.04	-0.04	-0.04	-0.04

Transaction probability
 $T(s, a, s') = 0.8$ for intended,
 0.2 for sideways
 Discount Gamma $\gamma = 1.0$



The Initial Utility Values

0.0	0.0	0.0	0.0
0.0		0.0	0.0
0.0	0.0	0.0	0.0

$$U_{i+1}(s) \leftarrow R(s) + \gamma \max_a \sum_{s'} T(s, a, s') U_i(s')$$

0.812	0.868	0.918	+1
0.762		0.660	-1
0.705	0.655	0.611	0.388

The Final Utility Values

Bellman Iteration Example (1st iteration)

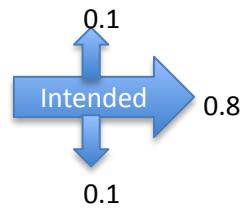
Reward $R(s)$

-0.04	-0.04	-0.04	+1
-0.04		-0.04	-1
-0.04	-0.04	-0.04	-0.04

Transaction probability

$T(s, a, s') = 0.8$ for intended,
0.2 for sideways

Discount Gamma $\gamma = 1.0$



The Initial Utilities U_0

0.0	0.0	0.0	0.0
0.0		0.0	0.0
0.0	0.0	0.0	0.0

$$U_1(s_1) = -0.04 + 1.0 \cdot \max(0.0 * 0.0)$$

$$U_1(s_4) = +1.0 + 1.0 \cdot \max(0.0 * 0.0)$$

$$U_1(s_7) = -1.0 + 1.0 \cdot \max(0.0 * 0.0)$$

-0.04	-0.04	-0.04	+1.0
-0.04		-0.04	-1.0
-0.04	-0.04	-0.04	-0.04

The Utility Values U_1 after 1st iteration

Bellman Iteration Example (2nd iteration)

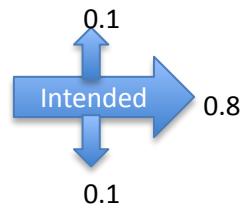
Reward $R(s)$

-0.04	-0.04	-0.04	+1
-0.04		-0.04	-1
-0.04	-0.04	-0.04	-0.04

Transaction probability

$T(s, a, s') = 0.8$ for intended,
0.2 for sideways

Discount Gamma $\gamma = 1.0$



U_1

-0.04	-0.04	-0.04	+1.0
-0.04		-0.04	-1.0
-0.04	-0.04	-0.04	-0.04



$$U_2(s_4) = +1.0 + 1.0 * \max(0, 0, 0)$$

$$U_2(s_7) = -1.0 + 1.0 * \max(0, 0, 0)$$

$$U_2(s_3) = -0.04 + 1.0 * \max(0.8 * 1 - 1 * 0.04 - 1 * 0.04, \dots)$$

$$U_2(s_6) = -0.04 + 1.0 * \max(-0.8 * 0.04 - 1 * 0.04 - 1 * 0.04, \dots)$$

U_2

?	?	0.752	+1.0
?		-0.08	-1.0
?	?	?	?



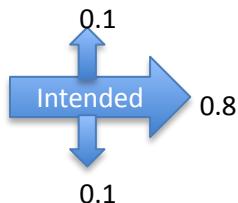
The Utility Values U_2 after 2nd iteration
Please fill in the values for “?”

Bellman Iteration Example

Reward $R(s)$

-0.04	-0.04	-0.04	+1
-0.04		-0.04	-1
-0.04	-0.04	-0.04	-0.04

Transaction probability
 $T(s, a, s') = 0.8$ for intended,
0.2 for sideways
Discount Gamma $\gamma = 1.0$



The Initial Utilities

0.0	0.0	0.0	0.0
0.0		0.0	0.0
0.0	0.0	0.0	0.0

$$U_{i+1}(s) \leftarrow R(s) + \gamma \max_a \sum_{s'} T(s, a, s') U_i(s')$$

0.812	0.868	0.918	+1
0.762		0.660	-1
0.705	0.655	0.611	0.388

Please verify these final Utility Values by yourself!!

Utility Value Iteration

(3 types of rewards)

- When rewards are given as $R(s)$
 - $U_{t+1} \leftarrow R(s) + \gamma \max_a \sum_{s'} T(s, a, s') U_t(s')$
- When rewards are given as $R(s, a)$
$$U_{t+1} = \max_a [R(s, a) + \gamma \sum_{s'} T(s, a, s') U_t(s')]$$
- When rewards are given as $R(s, a, s')$
 - $U_{t+1} \leftarrow \gamma \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + U_t(s')]$
 - $U_{t+1} \leftarrow \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma U_t(s')]$

Utility Value Iteration for POMDP

- In MDP, utility value iteration, when sensors are certain, we know the next state s'

$$U_{i+1}(s) \leftarrow R(s) + \gamma \max_a \sum_{s'} T(s, a, s') U_i(s')$$

- In POMDP, where sensors are uncertain, we must average over all possible evidences for the next state s' (see the last term below):

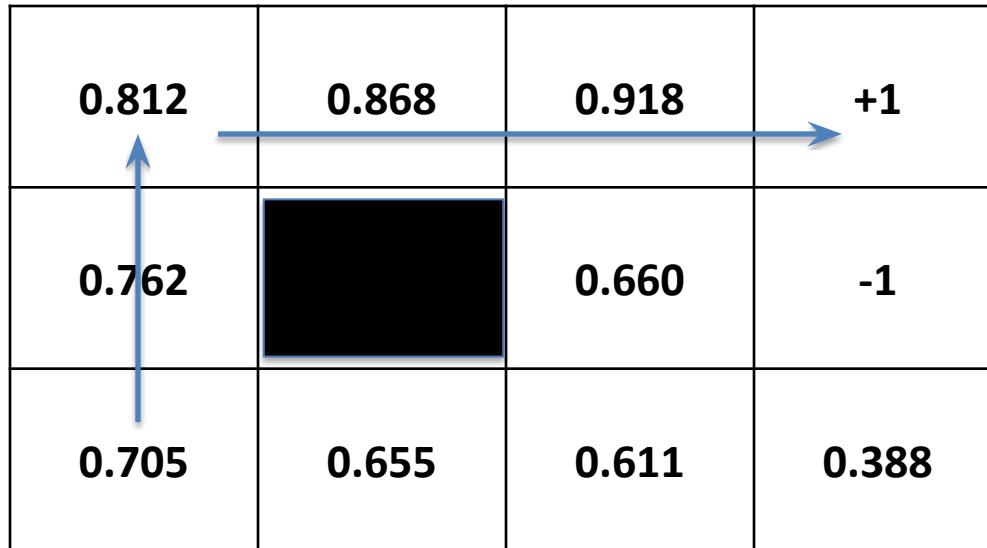
$$\alpha_p(s) = R(s) + \gamma \left(\sum_{s'} T(s, a, s') \sum_e P(e | s') \alpha_{p.e}(s') \right)$$

Policy and Optimal Policy

- A solution of (PO)MDP can be represented as
 - A Policy $\pi(s)=a$
 - Each execution of policy from s_0 may yield a different history or path due to the probabilistic nature of the transition model
 - An optimal policy $\pi^*(s)$ is a policy that yields the highest expected utility

Policy Based on Known Utility Values

0.812	0.868	0.918	+1
0.762		0.660	-1
0.705	0.655	0.611	0.388



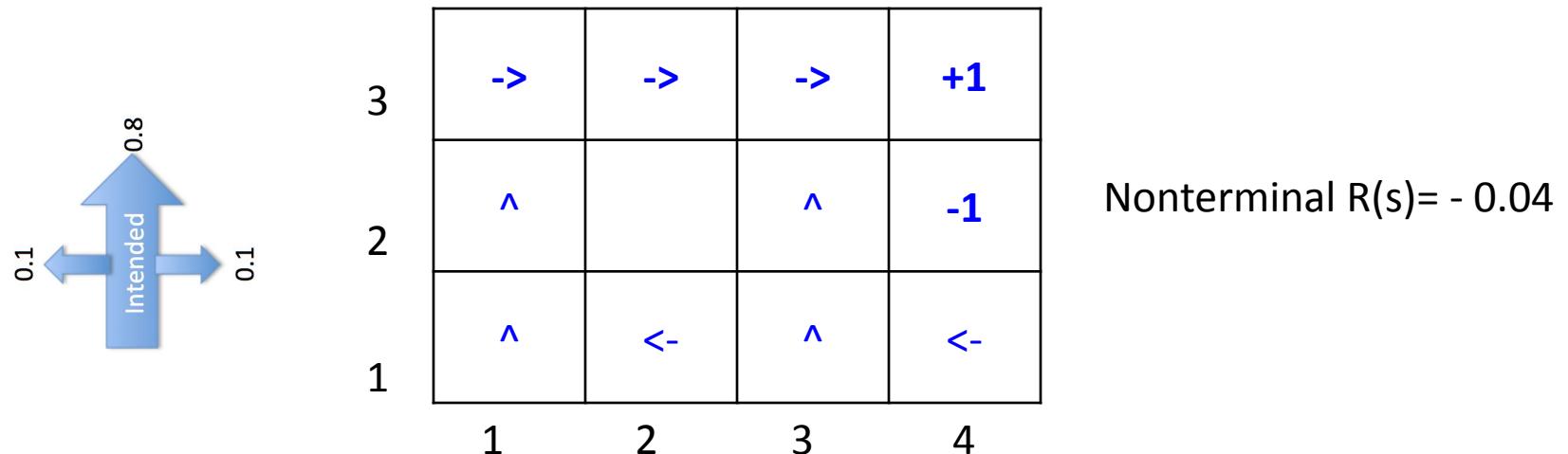
Suppose we know the utility values of the states as above,
What path would an optimal policy choose?

Rewards -> Utility Values -> Policies

Optimal Policy Examples

Depending on Reward distribution R(s)

If the **rewards** for the nonterminal states are evenly distributed (e.g., -0.04), then the path chosen will depend on the probabilities of the transition model



Caution: because the nondeterministic actions, from state (3,2) or (4,1), you may “accidentally” go to (4,2). So there is a “risk” in this policy.

Optimal Policy Examples

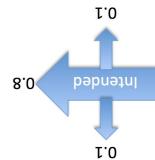
Depending on the reward distribution $R(s)$

Nonterminal rewards $R(s) = -0.04$

>	>	>	+1
^		^	-1
^	<	?	<

Why not go up?

Hint: Consider the nature of actions,
deterministic or not
Go up is “risky”



>	>	>	+1
^		> suicide	-1
^	>	>	^ suicide

$R(s) < -1.6284$
(life is painful, death is better)

>	>	> end nicely	+1
^		< no risk	-1
^	<	<	^ risky

$-0.0221 < R(s) < 0$
(life is good, minimize
risks, willing to end nicely)

>	>	>	+1
^		^ risky	-1
^	>	^	<

$-0.4278 < R(s) < -0.0850$
(life is OK, willing to risk)

◆	◆	< don't end	+1
◆		< no risk	-1
◆	◆	◆	∨ no risk

$R(s) > 0$
(life is rewarding,
I don't want to end it)

Optimal Policy

- You can compute the optimal policy once after all $U^*(s)$ are known

$$\pi_s^* = \operatorname{argmax}_a \sum T(s, a, s') U^*(s')$$

- Or, you can compute it incrementally every iteration when $U_i(s)$ is updated
 - This is called “Policy Iteration” (see the next slide)

Policy Iteration

(improving $\pi(s)$ every step)

- Start with a randomly chosen initial policy π_0
- Iterate until no change in utility values of the state:
- Policy evaluation: given a policy π_i , calculate the utility $U_i(s)$ of every state s using policy π_i by solving the system of equations:

$$U_{\pi}(s) = R(s) + \gamma \sum_{s'} T(s, \pi(s), s') U_{\pi}(s')$$

- Policy improvement: calculate the new policy π_{i+1} using one-step look-ahead based on the current $U_i(s)$:

$$\pi_{i+1}(s) = \operatorname{argmax}_a \sum_{s'} T(s, a, s') U^*(s')$$

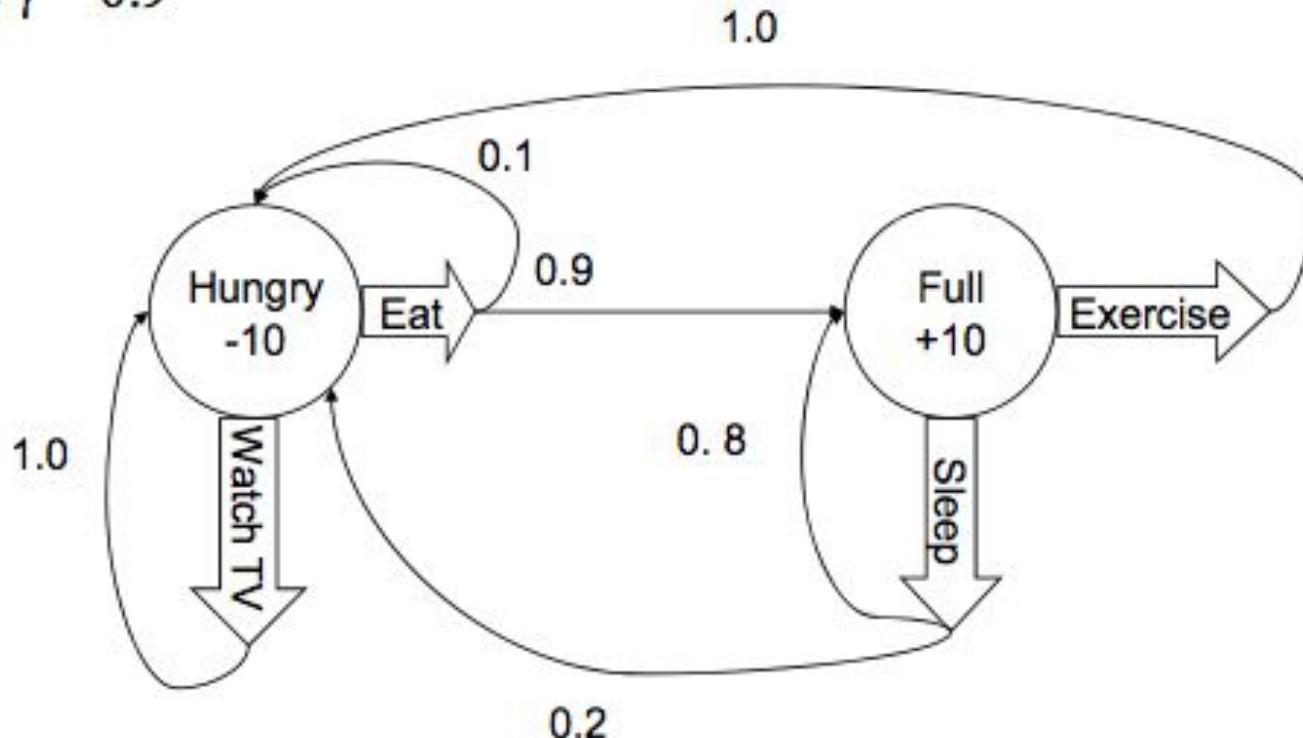
Policy Iteration Comments

- Each step of policy iteration is guaranteed to strictly improve the policy at some state when improvement of state utility value is possible
- Converge to the optimal policy
- Gives the exact value of the optimal policy

Policy Iteration Example

Do one iteration of policy iteration on the MDP below. Assume an initial policy of $\pi_1(\text{Hungry}) = \text{Eat}$ and $\pi_1(\text{Full}) = \text{Sleep}$.

Let $\gamma = 0.9$



Policy Iteration Example

Use initial policy for Hungry: $\pi_1(\text{Hungry}) = \text{Eat}$

$$U_1(\text{Hungry}) = -10 + (0.9)[(0.1)U_1(\text{Hungry}) + (0.9)U_1(\text{Full})]$$

$$\Rightarrow U_1(\text{Hungry}) = -10 + (0.09)U_1(\text{Hungry}) + (0.81)U_1(\text{Full})$$

$$\Rightarrow (0.91)U_1(\text{Hungry}) - (0.81)U_1(\text{Full}) = -10$$

Use initial policy for Full: $\pi_1(\text{Full}) = \text{Sleep.}$

$$U_1(\text{Full}) = 10 + (0.9)[(0.8)U_1(\text{Full}) + (0.2)U_1(\text{Hungry})]$$

$$\Rightarrow U_1(\text{Full}) = 10 + (0.72)U_1(\text{Full}) + (0.18)U_1(\text{Hungry})]$$

Policy Iteration Example

$$\begin{aligned} (0.91)U_1(\text{Hungry}) - (0.81)U_1(\text{Full}) &= -10 \dots \text{(Equation 1)} \\ (0.28)U_1(\text{Full}) - (0.18)U_1(\text{Hungry}) &= 10 \dots \text{(Equation 2)} \end{aligned} \quad \left. \begin{array}{l} \text{Solve for} \\ U_1(\text{Hungry}) \\ \text{and } U_1(\text{Full}) \end{array} \right\}$$

From Equation 1:

$$(0.91)U_1(\text{Hungry}) = -10 + (0.81)U_1(\text{Full})$$

$$\Rightarrow U_1(\text{Hungry}) = (-10/0.91) + (0.81/0.91)U_1(\text{Full})$$

$$\Rightarrow U_1(\text{Hungry}) = -10.9 + (0.89)U_1(\text{Full})$$

Policy Iteration Example

$$\begin{aligned} (0.91)U_1(\text{Hungry}) - (0.81)U_1(\text{Full}) &= -10 \dots \text{(Equation 1)} \\ (0.28)U_1(\text{Full}) - (0.18)U_1(\text{Hungry}) &= 10 \dots \text{(Equation 2)} \end{aligned} \quad \left. \begin{array}{l} \text{Solve for} \\ U_1(\text{Hungry}) \\ \text{and } U_1(\text{Full}) \end{array} \right\}$$

Substitute $U_1(\text{Hungry}) = -10.9 + (0.89)U_1(\text{Full})$ into Equation 2

$$(0.28)U_1(\text{Full}) - (0.18)[-10.9 + (0.89)U_1(\text{Full})] = 10$$

$$\Rightarrow (0.28)U_1(\text{Full}) + 1.96 - (0.16)U_1(\text{Full}) = 10$$

$$\Rightarrow (0.12)U_1(\text{Full}) = 8.04$$

$$\Rightarrow U_1(\text{Full}) = 67$$

$$\Rightarrow U_1(\text{Hungry}) = -10.9 + (0.89)(67) = -10.9 + 59.63 = 48.7$$

Policy Iteration Example

$$\pi_2(\text{Hungry})$$

$$\begin{aligned} &= \underset{\{\text{Eat, WatchTV}\}}{\operatorname{argmax}} \left\{ \begin{array}{l} T(\text{Hungry}, \text{Eat}, \text{Full})U_1(\text{Full}) + \\ T(\text{Hungry}, \text{Eat}, \text{Hungry})U_1(\text{Hungry}) \\ T(\text{Hungry}, \text{WatchTV}, \text{Hungry})U_1(\text{Hungry}) \end{array} \right\} \\ &= \underset{\{\text{Eat, WatchTV}\}}{\operatorname{argmax}} \left\{ \begin{array}{l} (0.9)U_1(\text{Full}) + (0.1)U_1(\text{Hungry}) \\ (1.0)U_1(\text{Hungry}) \end{array} \right\} \\ &= \underset{\{\text{Eat, WatchTV}\}}{\operatorname{argmax}} \left\{ \begin{array}{l} (0.9)(67) + (0.1)(48.7) \\ (1.0)(48.7) \end{array} \right\} \\ &= \underset{\{\text{Eat, WatchTV}\}}{\operatorname{argmax}} \left\{ \begin{array}{l} 65.2 \\ 48.7 \end{array} \right\} \\ &= \text{Eat} \end{aligned}$$

Policy Iteration Example

$\pi_2(\text{Full})$

$$\begin{aligned} &= \underset{\{\text{Exercise}, \text{Sleep}\}}{\operatorname{argmax}} \left\{ \begin{array}{ll} T(\text{Full}, \text{Exercise}, \text{Hungry})U_1(\text{Hungry}) & [\text{Exercise}] \\ T(\text{Full}, \text{Sleep}, \text{Full})U_1(\text{Full}) + \\ T(\text{Full}, \text{Sleep}, \text{Hungry})U_1(\text{Hungry}) & [\text{Sleep}] \end{array} \right\} \\ &= \underset{\{\text{Exercise}, \text{Sleep}\}}{\operatorname{argmax}} \left\{ \begin{array}{ll} (1.0)U_1(\text{Hungry}) & [\text{Exercise}] \\ (0.8)U_1(\text{Full}) + (0.2)U_1(\text{Hungry}) & [\text{Sleep}] \end{array} \right\} \\ &= \underset{\{\text{Exercise}, \text{Sleep}\}}{\operatorname{argmax}} \left\{ \begin{array}{ll} (1.0)(48.7) & [\text{Exercise}] \\ (0.8)(67) + (0.2)(48.7) & [\text{Sleep}] \end{array} \right\} \\ &= \underset{\{\text{Exercise}, \text{Sleep}\}}{\operatorname{argmax}} \left\{ \begin{array}{ll} 48.7 & [\text{Exercise}] \\ 63.34 & [\text{Sleep}] \end{array} \right\} \\ &= \text{Sleep} \end{aligned}$$

Policy Iteration Example

- The new policy π_2 after an iteration from π_1
 - $\pi_2(\text{Hungry}) \rightarrow \text{Eat}$
 - $\pi_2(\text{Full}) \rightarrow \text{Sleep}$

Summary for Uncertain Environments

- POMDP is very (the most) general model
 - Deal with uncertainty by
 - action models and sensor models
- Incorporate goals -> rewards -> utilities -> policy
 - Utilities and policies can be computed from rewards
 - Systematically (Bellman equations)
 - Iteratively (Bellman's iteration algorithm)
 - Solve a problem by following a good policy
 - One policy for one problem/goal
 - Different policies are needed for different goals

REINFORCEMENT LEARNING

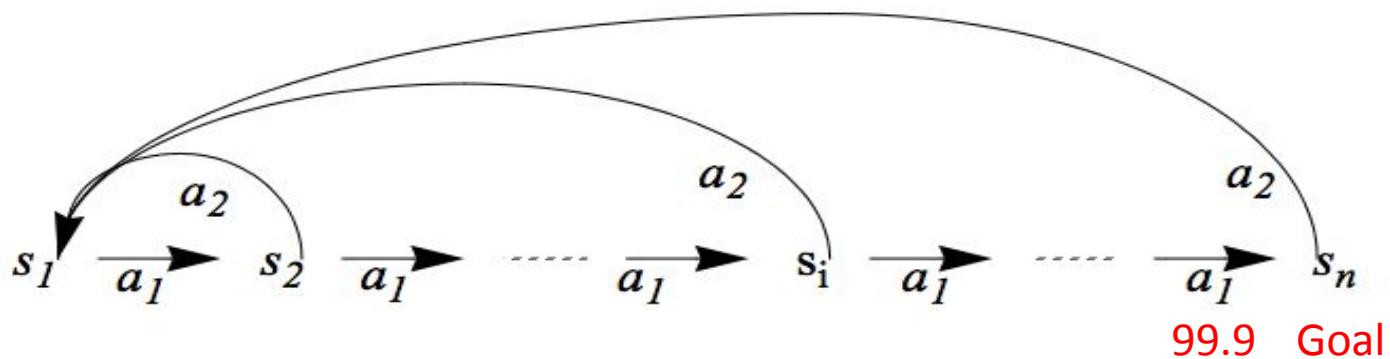
Markov Decision Process

- **Markov Decision Process**

- We learned that a Markov Decision Process (MDP) consists of
 - A set of **states** S (with an initial state s_0)
 - A set of **actions** A in each state,
 - A set of **percepts** that can be sensed
 - A **transition** model $P(s'|s,a)$, or $T(s,a,s')$
 - ~~A sensor model $\pi = P(z|s)$~~
 - **The current state** distribution
 - A **reward function** $R(s,a,s')$ // careful here
- The MDP's solution identifies the best action to take in each state
 - **Optimal policy** $\pi(s)=a$ obtained by value iteration or policy iteration (aka dynamic programming)

An Example for MDP

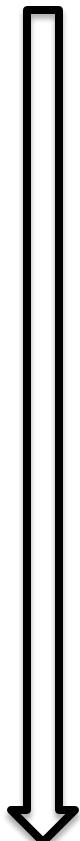
- Given
 - States: s_1, \dots, s_n , Actions: a_1, a_2
 - Transition Probability Distribution: (assume $s_{n+1} = s_1$)
 - $P(s_i, a_1, s_{i+1}) = 0.8, P(s_i, a_1, s_1) = 0.2, P(s_i, a_2, s_1) = 0.9, P(s_i, a_2, s_{i+1}) = 0.1.$
 - Reward: all $R(s_i, a_j, s_j) = 0.0$, except $R(s_{n-1}, a_1, s_n) = 99.9$
 - We define the goal state to be s_n
 - The Future discount factor: gamma $\gamma = 0.7$
- State utility values: $U(s_{n-1}), U(s_{n-2}), U(s_{n-3}), \dots$ and $U(s_2), U(s_1)$
- Optimal Policy: $\pi(s_i) = a_1$



Reinforcement Learning

- Given a Markov Decision Process (environment)
 - A set of states S ✓ (known)
 - A set of actions A in each state ✓ (known)
 - A set of percepts that can be sensed ✓ (known)
 - The current state ✓ (known)
 - Transitions $P(s'| s, a)$ ✗ (only known by the env)
 - Rewards $R(s, a, s')$ ✗ (only known by the env)
- Could the agent still learn the optimal policy?

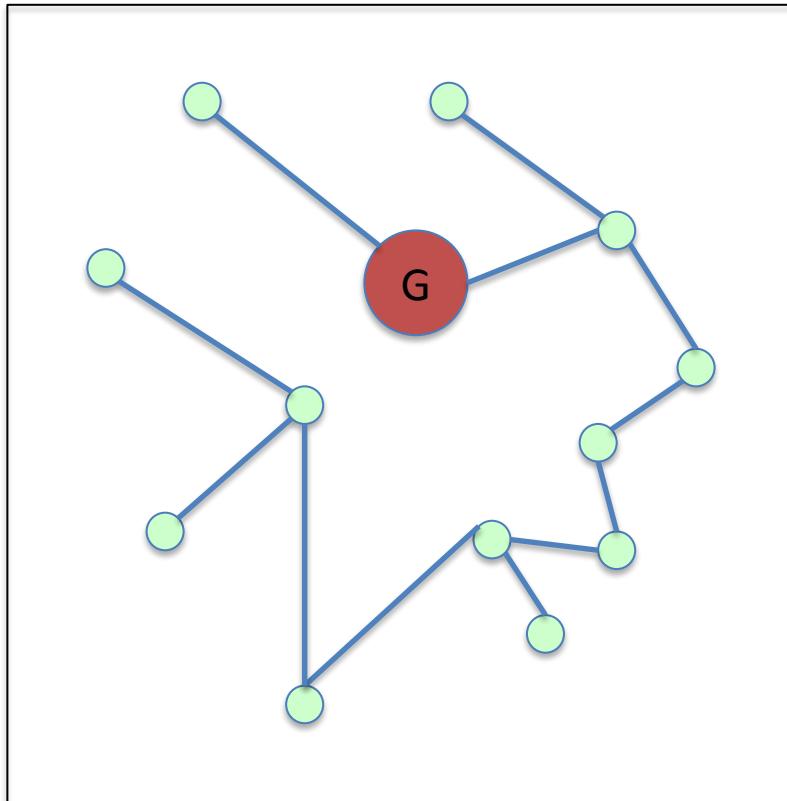
Goals, Rewards, Utilities, Policies



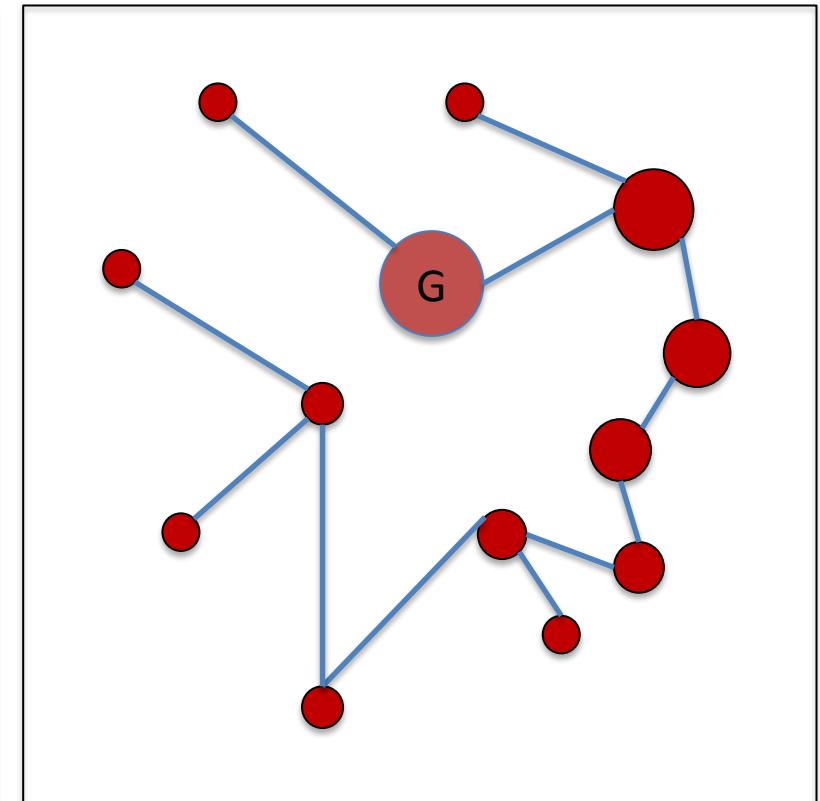
- Goals
 - Given to the agent from the problem statements
- Rewards
 - Given to the agent, designed based on the goals
- Utility values for states
 - Computed by the agent, based on the rewards
- Policies
 - Computed or learned by the agent
 - Used by the agent to select its actions
 - The better a policy, the more rewards it collects

Reinforcement Learning

Propagating the Delayed Rewards



Before RL



After RL

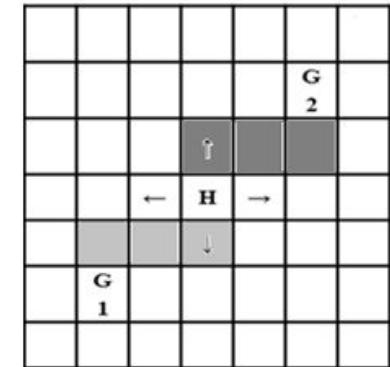
After RL, no matter where you are, you know which way to go to the Goal!
This is because all states now have utility values that will lead your agent to the goal.

Reinforcement Learning (Key Ideas)

- Can the agent still learn the optimal policy?
 - When it knows only the states S and actions A, but not the transition model $P(s,a,s')$ and/or reward function $R(s,a,s')$
- Try an action on a state of the environment, and get a “sample” that includes (s, a, s', r) and maybe U :
 - A state transition $(s,a) \rightarrow s'$
 - A reward received for the action $r = R(s, a, s')$
 - And maybe the utility value of the next state $U(s')$
- Objective: Try different actions in states to discover an optimal policy $\pi(s)=a$ and eventually tells the agent which action will lead to the most rewarding states

How to Explore the State Space

- Visit different states to discover a policy and improve it
 - Numerous strategies
 - A simple strategy is executing actions randomly
 - Initially there is no policy, but random actions eventually discover a policy
 - At every time step, act randomly with a probability (for exploration), otherwise, follow the current policy $\Pi(s)=a$
 - The random action may causes unnecessary action execution when the optimal policy is already discovered
 - One solution is to lower the probability of selecting a random action over time (i.e., reduce the “temperature”)

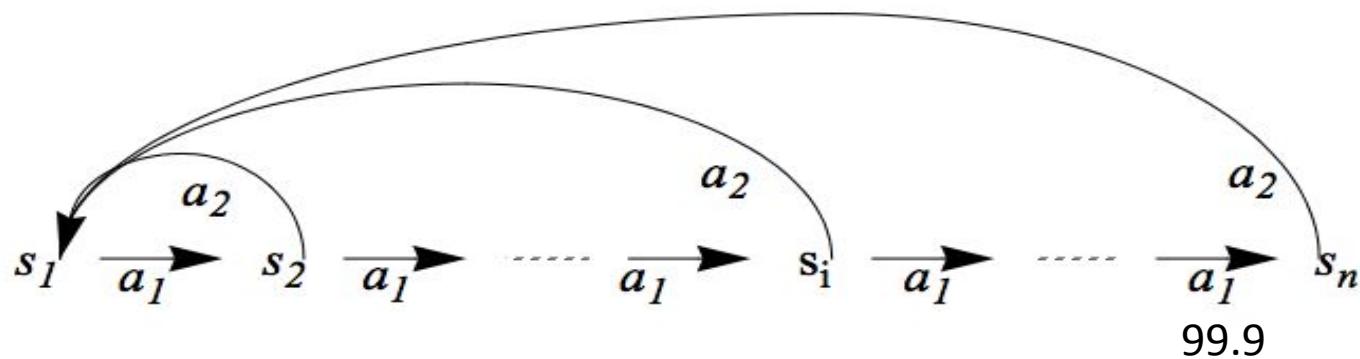


Reinforcement Learning

- Objective: Learn the optimal policy $\pi^*(s)=a$
- Two general classes of algorithms:
 - **Model-based RL**
 - First learn the transition model and the utility values of states, then learn the policy (e.g., policy iteration)
 - **Model-free RL**
 - Learn the policy without learning an explicit transition model, but by receiving “samples” from the environment
 - Three algorithms we will learn:
 - Monte Carlo
 - Temporal Difference
 - Q-Learning

An Example for Model-Based RL

- Given
 - States: s_1, \dots, s_n ,
 - Actions: a_1, a_2
 - Probabilistic transition model: (assume $s_{n+1} = s_1$)
 - $P(s_i, a_1, s_{i+1}) = 0.8, P(s_i, a_1, s_1) = 0.2, P(s_i, a_2, s_1) = 0.9, P(s_i, a_2, s_{i+1}) = 0.1.$
 - Reward: all $R(s_i, a_j, s_j) = 0.0$, except $R(s_{n-1}, a_1, s_n) = 99.9$
 - Future discount factor: $\gamma = 0.7$
- Try to learn: $U(s_{n-1}), U(s_{n-2}), U(s_{n-3}), \dots$ and $U(s_2), U(s_1)$



Model-Based RL

- Learn an approximate model based on random actions
 - From a state s , count outcomes of s' for each (s, a)
 - Normalize to give an estimate of $P(s, a, s')$
 - Learn state utility $U(s)$ from rewards $R(s, a, s')$ when encountered
 - Learn a policy (e.g., policy iteration) and solve the MDP

State Utility Value Iteration

(improving $U(s)$ every step)

- $U(s)$: the expected sum of maximum rewards achievable starting from a particular state s
- Bellman equations:
 - Many equations must be solved *simultaneously*
- Bellman iteration:
 - Converge to $U^*(s)$ step by step

$$U^*(s) = R(s) + \gamma \max_a \sum_{s'} T(s, a, s') U^*(s')$$

$$U_{i+1}(s) \leftarrow R(s) + \gamma \max_a \sum_{s'} T(s, a, s') U_i(s')$$

Policy Iteration

(improving $\pi(s)$ every step)

- Start with a randomly chosen initial policy π_0
- Iterate until no change in utilities:
- Policy evaluation: given a policy π_i , calculate the utility $U_i(s)$ of every state s using policy π_i by solving the system of equations:
$$U_\pi(s) = R(s) + \gamma \sum_{s'} T(s, \pi(s), s') U_\pi(s')$$
- Policy improvement: calculate the new policy π_{i+1} using one-step look-ahead based on $U_i(s)$:

$$\pi_{i+1}(s) = \operatorname{argmax}_a \sum_{s'} T(s, a, s') U^*(s')$$

State Utility Value Iteration (review)

(improving $U(s)$ every step)

- $U(s)$: the expected sum of maximum rewards achievable starting from a particular state s
- Bellman equations:
 - Many equations must be solved *simultaneously*

$$U^*(s) = R(s, a, s') + \gamma \max_a \sum_{s'} T(s, a, s') U^*(s')$$

- Bellman iteration:
 - Converge to $U^*(s)$ step by step

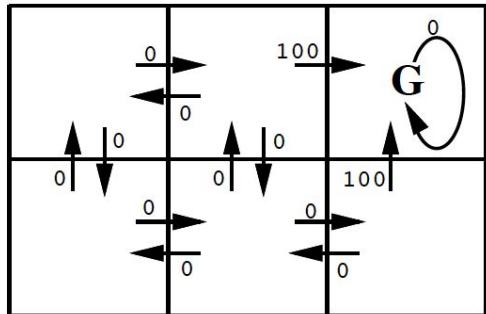
$$U_{i+1}(s) \leftarrow R(s, a) + \gamma \max_a \sum_{s'} T(s, a, s') U_i(s')$$

Utility Value Iteration (example)

Find your way to the goal

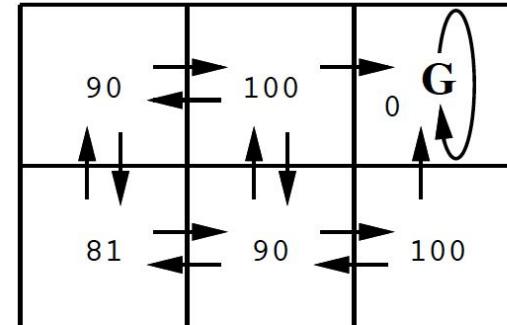
Use Bellman Iteration to compute state values from rewards

$$U_{i+1}(s) \leftarrow R(s, a) + \gamma \max_a \sum_{s'} T(s, a, s') U_i(s')$$



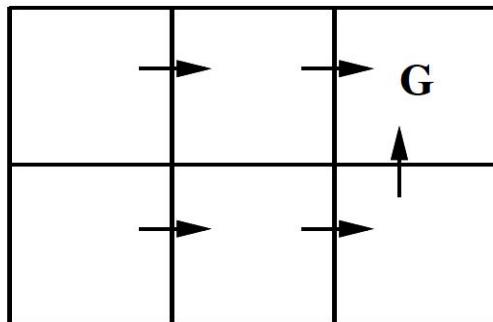
$$\gamma = 0.9$$

A blue arrow pointing to the right, indicating the transition to the next step in the value iteration process.



$r(s, a)$ (immediate reward) values

$V^*(s)$ values



One optimal policy

$$\begin{aligned}V^\pi(s_t) &= r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots \\&= \sum_{i=0}^{\infty} \gamma^i r_{t+i} \quad 0 \leq \gamma < 1\end{aligned}$$

Policy Iteration (review)

(improving $\pi(s)$ every step)

- Start with a randomly chosen initial policy π_0
- Iterate until no change in the utility values of the states:
- Policy evaluation: given a policy π_i , calculate the utility value $U_i(s)$ of every state s using policy π_i by solving the system of equations:

$$U_{\pi}(s) = R(s, \pi(s), s') + \gamma \sum_{s'} T(s, \pi(s), s') U_{\pi}(s')$$

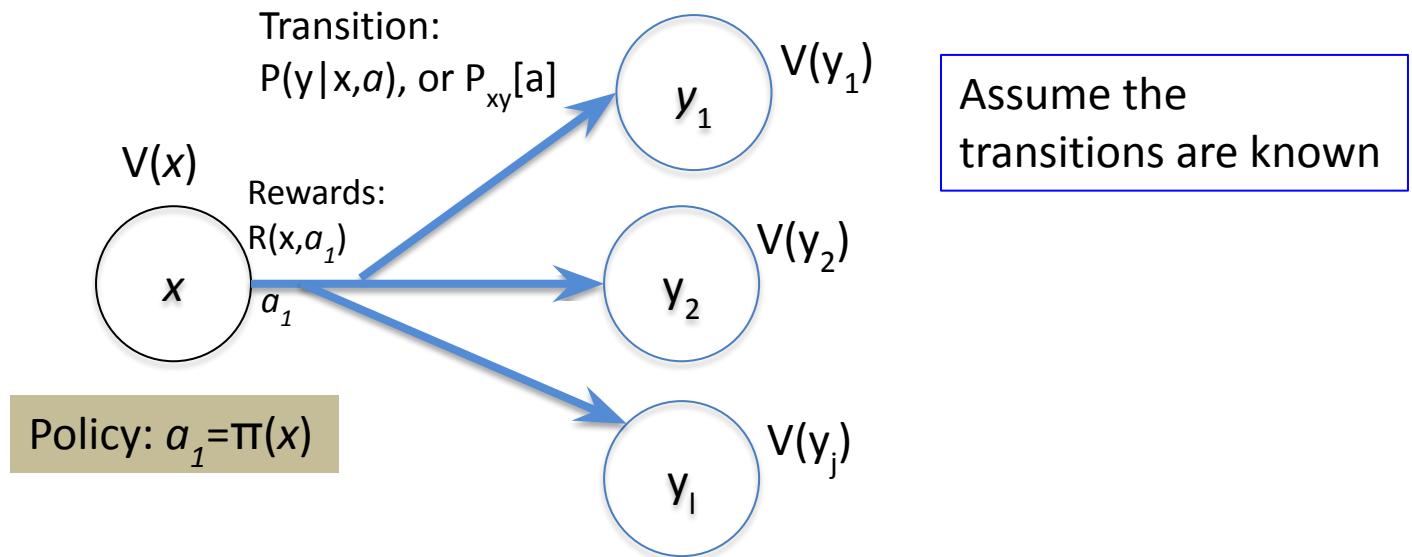
- Policy improvement: calculate the new policy π_{i+1} using one-step look-ahead based on $U_i(s)$:

$$\pi_{i+1}(s) = \operatorname{argmax}_a \sum_{s'} T(s, a, s') U^*(s')$$

Compute Utility from Rewards and Policy

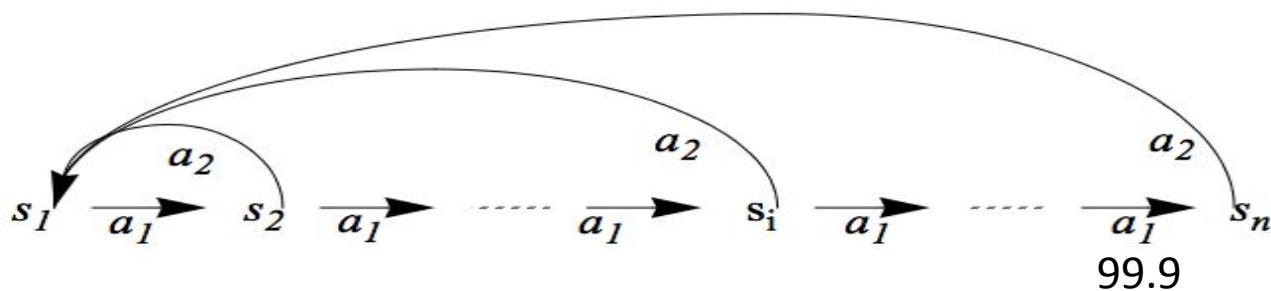
to determine an optimal policy, one that maximizes the total discounted expected reward. By *discounted reward*, we mean that rewards received m steps later are worth less than rewards received now by a factor of γ^m ($0 < \gamma < 1$). Under a policy π , the value of state x (again with respect to $R(x, a)$) is

$$V^\pi(x) \equiv R(x, \pi(x)) + \gamma \sum_y P_{xy}[\pi(x)] V^\pi(y)$$



An Example for Model-Based RL

- Given
 - States: s_1, \dots, s_n Actions: a_1, a_2
 - Probabilistic transition model: (assume $s_{n+1} = s_1$)
 - $P(s_i, a_1, s_{i+1}) = 0.8, P(s_i, a_1, s_1) = 0.2, P(s_i, a_2, s_1) = 0.9, P(s_i, a_2, s_{i+1}) = 0.1.$
 - Reward: all $R(s_i, a_j, s_j) = 0.0$, except $R(s_{n-1}, a_1, s_n) = 99.9$
 - Future discount factor: $\gamma = 0.7$, all Initial utility values are: $U(s_i) = 0$
- 1st iteration
 - $U(s_{n-1}) = 0.7 \max\{0.8(99.9 + 0) + 0.2(0 + 0), \dots, \dots\} = 55.944$; or. // see slide #49
 - $U(s_{n-1}) = \max\{0.8(99.9 + 0.7 * 0) + 0.2(0 + 0), \dots, \dots\} = 79.92;$
 - $U(s_{n-2}) = 0, U(s_{n-3}) = 0, \dots, U(s_1) = 0$
 - After many iterations: $U(s_1) = ?$ Is it $99.9 * (0.7 * 0.8)^{n-1}$?



$$U_{t+1}(s) \leftarrow \gamma \max_a \sum_{s'} P(s, a, s')[R(s, a, s') + U_t(s')]$$

$$U_{t+1}(s) \leftarrow \max_a \sum_{s'} P(s, a, s')[R(s, a, s') + \gamma U_t(s')]$$

Model-Based RL

- Learn an approximate model based on random actions
 - From a state s , count outcomes of s' for each (s, a)
 - Normalize to give an estimate of $P(s, a, s')$
 - Learn state utility $U(s)$ from rewards $R(s, a, s')$ when encountered
 - Solve the learned MDP and obtain a policy (e.g. policy iteration)

Pros:

If and when the goal changes, one can use the learned model $P(s, a, s')$ to recalculate $U(s)$ and compute a new policy offline

Cons:

Larger representation than model-free RL

Model-Free Reinforcement Learning

- Objective: Learn the optimal policy $\pi^*(s)=a$
- Model-based RL
 - First learn the transition model and the utility values of states, then learn the policy (e.g., policy iteration)
- **Model-free RL**
 - Learn the policy without learning an explicit model, but by receiving “samples” from the environment
 - Three algorithms we will learn:
 - Monte Carlo
 - Temporal Difference
 - Q-Learning

Monte Carlo RL (model free)

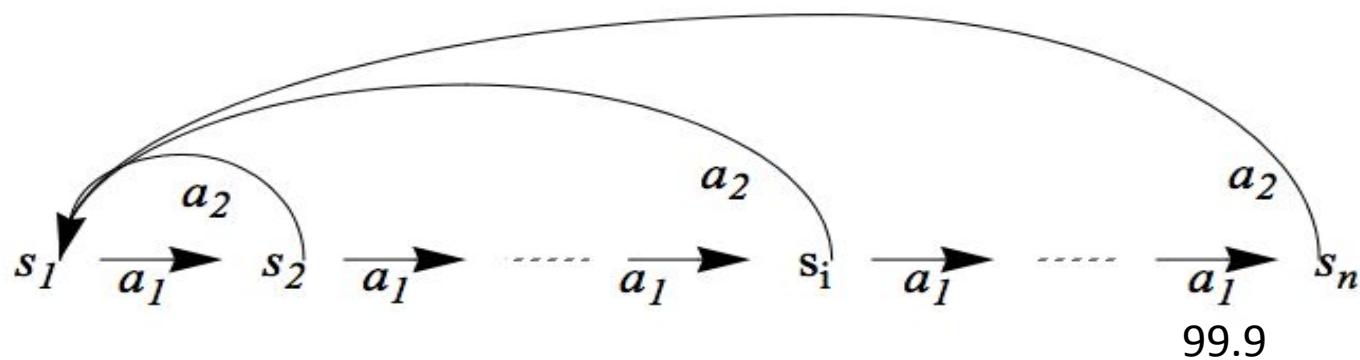
- Generate a fixed policy $\pi(s) = a$ based on $U(s)$
 - In each **episode**, the learner follows the policy starting at a random state
 - Average the sampled returns originating from each state
 - Generate a policy based on $U(s)$ as in “policy iteration”
- Cons:
 - The utility value of states $U(s)$ is given and fixed (not learned)
 - Works only in episodic problems
 - Takes a very long time to converge as learning is from complete sample returns
 - Wastes information as it figures out state values in isolation from other states

Temporal Difference RL (model free)

- Improve Monte Carlo, still using a fixed policy
 - Update $U(s)$ using each sample (s, a, s', r) received from the environment and each sample includes:
 1. A state transition $(s, a) \rightarrow s'$ done by the environment
 2. A reward received for the action $r = R(s, a, s')$
 3. And the value of the next state $U(s')$
 - That is, Sample = $r + \gamma U(s')$, where γ = future discount
- TD Algorithm:
 - $U(s) \leftarrow (1-\alpha)U(s) + \alpha * \text{Sample} = (1-\alpha)U(s) + \alpha[r + \gamma U(s')]$
 - Where α is the “learning rate” (how much you trust the sample)
 - You might decrease α over time when converges to the average
 - The parameter α also represents “forgetting long past values”
- Cons:
 - Only provides the utility values of states, not improve policy directly

An Example for TD-RL

- Given
 - States: s_1, \dots, s_n ,
 - Actions: a_1, a_2
 - Probabilistic transition model: (assume $s_{n+1} = s_1$)
 - $P(s_i, a_1, s_{i+1}) = 0.8$, $P(s_i, a_1, s_1) = 0.2$, $P(s_i, a_2, s_1) = 0.9$, $P(s_i, a_2, s_{i+1}) = 0.1$.
 - Reward: all $R(s_i, a_j, s_j) = 0.0$, except $R(s_{n-1}, a_1, s_n) = 99.9$
 - We define the goal state to be s_n
 - The learning rate $\alpha = 0.4$
 - The Future discount factor: $\gamma = 0.7$
- Use TD algorithm to learn: $U(s_{n-1}), U(s_{n-2}), U(s_{n-3}), \dots$ and $U(s_2), U(s_1)$



Q-Learning (Definition)

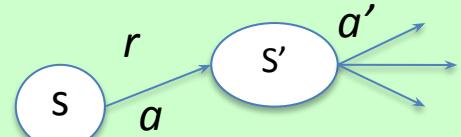
- The key idea is to use $Q(s,a)$ to represent the value of taking an action a in state s , rather than only the value of state $U(s)$:
 - Define:
$$Q(s,a) \equiv \sum_{s'} P(s,a,s')[R(s,a,s') + \gamma U(s')]$$
- Optimal $\pi(s) = \text{argmax } U^\pi(s) = \text{argmax}_a Q(s, a)$

$$Q_{k+1}(s,a) \leftarrow \sum_{s'} P(s,a,s')[R(s,a,s') + \gamma \max_{a'} Q_k(s',a')]$$

- The last term using Q to replace U

Q-Learning (Algorithm)

- Initially, let $Q(s, a)=0$, given α and γ
- Sample a transition and reward: (s,a,s',r)
 - Sample = $r + \gamma \max_{a'} Q(s',a')$
 - $Q(s,a) = (1-\alpha)Q(s,a) + \alpha * \text{Sample}$



$$Q_{t+1}(s,a) = (1-\alpha)Q_t(s,a) + \alpha[R(s,a,s') + \gamma \max_{a'} Q_t(s',a')]$$

Advantages:

1. No need for a fixed policy, can do off-policy learning, or directly learns a policy
2. The optimal policy $\pi(s)=a$ is to select the action a that has the best $Q(s, a)$
3. Provably convergent to the optimal policy when $t \rightarrow \infty$

The Q-Learning Algorithm

Initialize all $Q(s, a)$ arbitrarily

For all episodes

Initialize s

Repeat

 Choose a using policy derived from Q , e.g., ϵ -greedy

 Take action a , observe r and s'

 Update $Q(s, a)$:

$$Q(s, a) \leftarrow Q(s, a) + \eta(r + \gamma \max_{a'} Q(s', a') - Q(s, a))$$
$$s \leftarrow s'$$

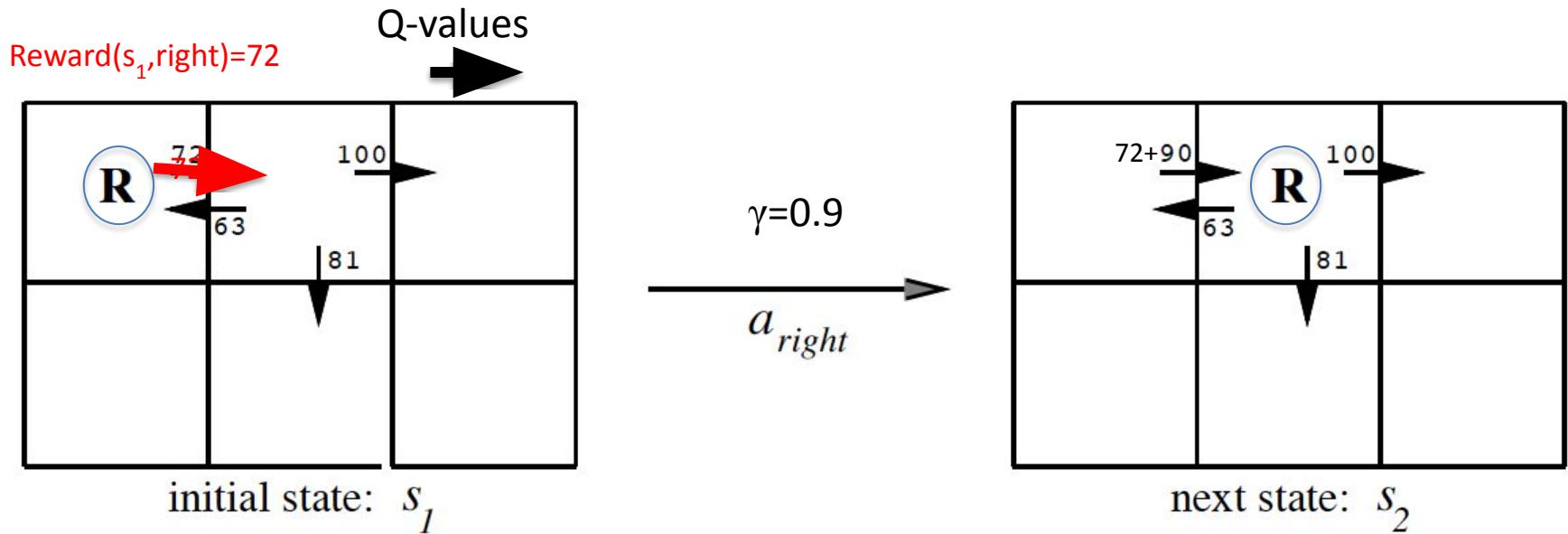
Until s is terminal state

Q-Learning (in One Formula)

$$Q(s_t, a_t) \leftarrow (1 - \alpha) \cdot \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \left(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\substack{\text{learned value} \\ \text{estimate of optimal future value}}} \right)$$

A blue arrow points from a box containing $r(s)$ or $r(s,a)$ up to the r_t term in the equation.

Q-Learning example from reward $r(s,a)$



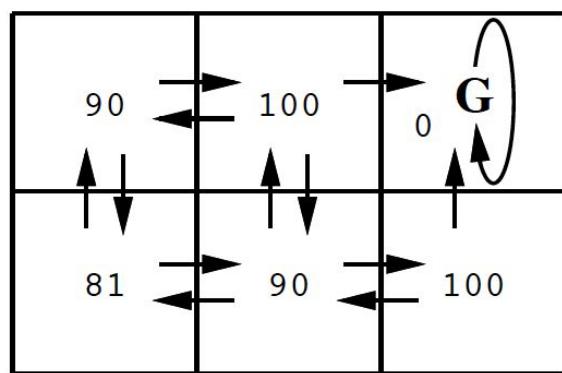
$$\hat{Q}(s_1, a_{right}) \leftarrow r + \gamma \max_{a'} \hat{Q}(s_2, a') \quad // \text{when } \alpha = 1$$

$$72 + 0.9 \max\{63, 81, 100\}$$

$$72 + 90$$

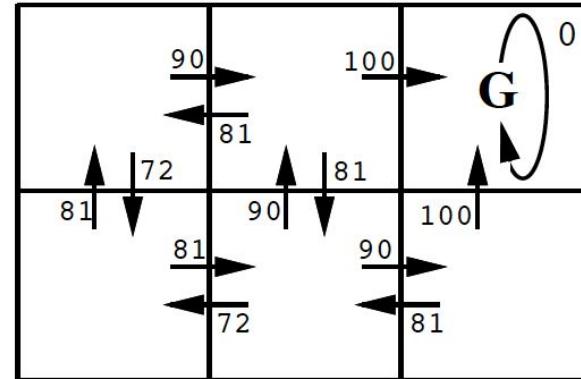
Back-propagate Q-values 1-step backwards

Q-Learning from state utility values



$V^*(s)$ values

Q-Learning

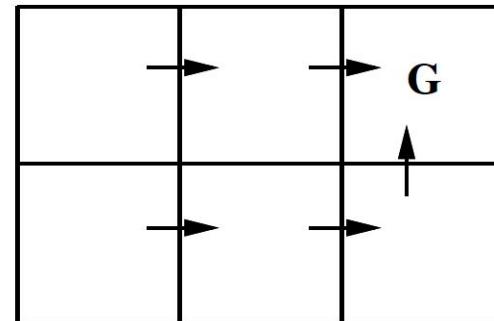


$Q(s, a)$ values

Notice these

$$\pi^*(s) = \underset{a}{\operatorname{argmax}} Q(s, a)$$

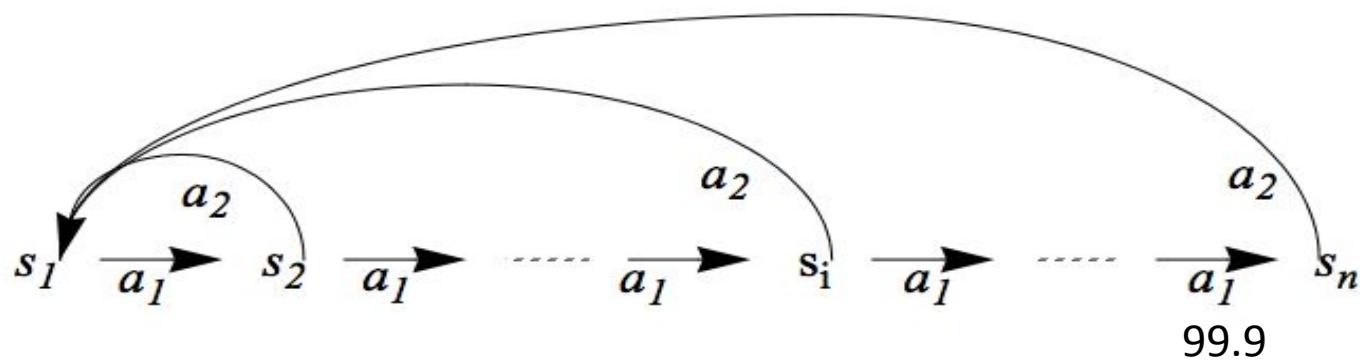
$$V^*(s) = \underset{a}{\max} Q(s, a)$$



One optimal policy

An Example for Q-Learning

- Given
 - States: s_1, \dots, s_n ,
 - Actions: a_1, a_2
 - Probabilistic transition model: (assume $s_{n+1} = s_1$)
 - $P(s_i, a_1, s_{i+1}) = 0.8, P(s_i, a_1, s_1) = 0.2, P(s_i, a_2, s_1) = 0.9, P(s_i, a_2, s_{i+1}) = 0.1$
 - Reward: all $R(s_i, a_j, s_j) = 0.0$, except $R(s_{n-1}, a_1, s_n) = 99.9$
 - We define the goal state to be s_n
 - The learning rate $\alpha = 0.4$
 - The Future discount factor: $\gamma = 0.7$
- Try to learn: $Q(s_{n-1}, a_1)$, $Q(s_{n-1}, a_2)$, $Q(s_{n-2}, a_1)$, and $Q(s_{n-2}, a_2) = ?$



An Example for Q-Learning

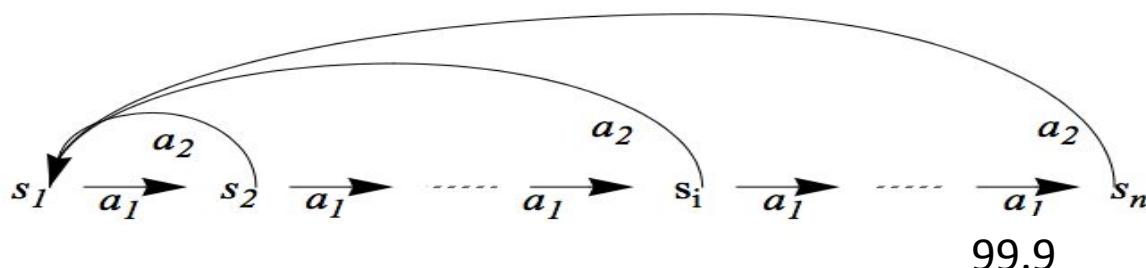
Initially: all $Q(s_i, a_j) = 0.0$

Assume you got two samples at the state s_{n-1} for action a_1

- $s_{n-1}, a_1, s_n, r=R(s_{n-1}, a_1, s_n)=99.9, \quad // P(s_{n-1}, a_1) \rightarrow s_n$
- $s_{n-1}, a_1, s_1, r=R(s_{n-1}, a_1, s_1)=0.0, \quad // P(s_{n-1}, a_1) \rightarrow s_1$

Updated Q value, if $P(s, a)$ are known:

- $Q(s_{n-1}, a_1) = 0.8 * [99.9 + 0.7 * 0.0] + 0.2[0.0 + 0.7 * 0.0] = 79.9$
- If $P(s, a)$ are unknown, assume s_{n-1} was visited 2 times so far:
 - sample₁=99.9+.7*0; $Q(s_{n-1}, a_1) = (1-0.4)*0.0 + 0.4*99.9 = 39.96$
 - sample₂=0+.7*0; $Q(s_{n-1}, a_1) = (1-0.4)*39.96 + 0.4*0.0 = 23.976$



Discussions of RL (1)

- Pros:
 - Easy to use in problems where the data can be mapped to states easily
 - Guaranteed to find the optimal policy given enough time even with suboptimal actions
- Cons:
 - States of the environment are not always known
 - Computation time is intractable for large or continuous state spaces
 - E.g. if each cell in a grid world is a state, then state space grows exponentially with number of rows and columns (states = map size = $m \times n$)
 - Cannot handle raw data, must use an approximation/reduction function
 - Designing approximation functions to disambiguate similar states requires human intelligence or an alternate learning technique
 - E.g. use the relative distance (states = $m \times n$) between two agents in a hunter-prey problem as opposed to their cell coordinates (states = $m \times n \times m \times n$)
 - Model-free RL cannot transfer the learned knowledge when the goal changes
 - Forgetting a learned policy is much more difficult (hysteresis), quicker to start from scratch

Discussions of RL (2)

Utility function is unknown at the beginning

- When agent visits each state, it receives a reward
 - Possibly negative

What function should it learn?

- $R(s)$: Utility-based agent
 - If it already knows the transition model
 - Then use MDP algorithm to solve for MEU actions
- $U(s,a)$: Q-learning agent
 - If it doesn't already know the transition model
 - Then pick action that has highest U in current state
- $\pi^*(s)$: reflex agent
 - Learn a policy directly, then pick the action that the policy says

Q Learning Summary

Modify Bellman equation to learn Q values of actions

- $U(s) = R(s) + \gamma \max_a \sum_{s_1} (P(s_1 | s, a) U(s_1))$
 - We don't know R or P
 - But when we perform a' in s , we move to s' and receive $R(s)$
 - We want $Q(s, a) = \text{Expected utility of performing } a \text{ in state } s$

Update Q after each step

- If we get a good reward now, then increase Q
- If we later get to a state with high Q , then increase Q here too
- $Q(s, a) \leftarrow (1-\alpha)Q(s, a) + \alpha(R(s) + \gamma \max_{a'} Q(s', a'))$
 - α is the learning rate, γ is the discount factor

Converges to correct values if α decays over time

- Similar to the “temperature” in simulated annealing

Q-Learning in Star War Environment

1 	2	3	4 
5		6	7 
8	9	10	11

Initial Q Values

$\alpha = 0.1$

$\gamma = 1$

1	U: 0.0 D: 0.0 L: 0.0 R: 0.0	2	U: 0.0 D: 0.0 L: 0.0 R: 0.0	3	U: 0.0 D: 0.0 L: 0.0 R: 0.0	4	U: 0.0 D: 0.0 L: 0.0 R: 0.0
5	U: 0.0 D: 0.0 L: 0.0 R: 0.0			6	U: 0.0 D: 0.0 L: 0.0 R: 0.0	7	U: 0.0 D: 0.0 L: 0.0 R: 0.0
8	U: 0.0 D: 0.0 L: 0.0 R: 0.0	9	U: 0.0 D: 0.0 L: 0.0 R: 0.0	10	U: 0.0 D: 0.0 L: 0.0 R: 0.0	11	U: 0.0 D: 0.0 L: 0.0 R: 0.0

Step 1

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha(R(s) + \gamma \max_{a'} Q(s',a'))$$

1	U: 0.0 D: 0.0 L: 0.0 R: 0.0	2	U: 0.0 D: 0.0 L: 0.0 R: 0.0	3	U: 0.0 D: 0.0 L: 0.0 R: 0.0	4	U: 0.0 D: 0.0 L: 0.0 R: 0.0
5	U: 0.0 D: 0.0 L: 0.0 R: 0.0			6	U: 0.0 D: 0.0 L: 0.0 R: 0.0	7	U: 0.0 D: 0.0 L: 0.0 R: 0.0
8	U: 0.0 D: 0.0 L: 0.0 R: 0.0	9	U: 0.0 D: 0.0 L: 0.0 R: 0.0	10	U: 0.0 D: 0.0 L: 0.0 R: 0.0	11	U: 0.0 D: 0.0 L: 0.0 R: 0.0

Step 1

$$Q(1, R) \leftarrow (1-\alpha)Q(1, R) + \alpha(R(1) + \gamma \max_{a'} Q(2, a'))$$

1	U: 0.0 D: 0.0 L: 0.0 R: 0.0	2	U: 0.0 D: 0.0 L: 0.0 R: 0.0	3	U: 0.0 D: 0.0 L: 0.0 R: 0.0	4	U: 0.0 D: 0.0 L: 0.0 R: 0.0
5	U: 0.0 D: 0.0 L: 0.0 R: 0.0			6	U: 0.0 D: 0.0 L: 0.0 R: 0.0	7	U: 0.0 D: 0.0 L: 0.0 R: 0.0
8	U: 0.0 D: 0.0 L: 0.0 R: 0.0	9	U: 0.0 D: 0.0 L: 0.0 R: 0.0	10	U: 0.0 D: 0.0 L: 0.0 R: 0.0	11	U: 0.0 D: 0.0 L: 0.0 R: 0.0

Step 1

$$Q(1, R) \leftarrow (1-\alpha)Q(1, R) + \alpha(-0.04 + \gamma \max_{a'} Q(2, a'))$$

1	U: 0.0 D: 0.0 L: 0.0 R: 0.0	2	U: 0.0 D: 0.0 L: 0.0 R: 0.0	3	U: 0.0 D: 0.0 L: 0.0 R: 0.0	4	U: 0.0 D: 0.0 L: 0.0 R: 0.0
5	U: 0.0 D: 0.0 L: 0.0 R: 0.0			6	U: 0.0 D: 0.0 L: 0.0 R: 0.0	7	U: 0.0 D: 0.0 L: 0.0 R: 0.0
8	U: 0.0 D: 0.0 L: 0.0 R: 0.0	9	U: 0.0 D: 0.0 L: 0.0 R: 0.0	10	U: 0.0 D: 0.0 L: 0.0 R: 0.0	11	U: 0.0 D: 0.0 L: 0.0 R: 0.0

Step 1

$$Q(1, R) \leftarrow (1 - \alpha)Q(1, R) + \alpha(-0.04 + \gamma \text{ 0.0})$$

1	U: 0.0	2	U: 0.0	3	U: 0.0	4	U: 0.0
	D: 0.0		D: 0.0		D: 0.0		D: 0.0
	L: 0.0		L: 0.0		L: 0.0		L: 0.0
	R: 0.0		R: 0.0		R: 0.0		R: 0.0
5	U: 0.0			6	U: 0.0	7	U: 0.0
	D: 0.0				D: 0.0		D: 0.0
	L: 0.0				L: 0.0		L: 0.0
	R: 0.0				R: 0.0		R: 0.0
8	U: 0.0	9	U: 0.0	10	U: 0.0	11	U: 0.0
	D: 0.0		D: 0.0		D: 0.0		D: 0.0
	L: 0.0		L: 0.0		L: 0.0		L: 0.0
	R: 0.0		R: 0.0		R: 0.0		R: 0.0

Step 1

$$Q(1, R) \leftarrow (1 - \alpha) 0.0 + \alpha (-0.04 + \gamma 0.0)$$

1	U: 0.0 D: 0.0 L: 0.0 R: 0.0	2	U: 0.0 D: 0.0 L: 0.0 R: 0.0	3	U: 0.0 D: 0.0 L: 0.0 R: 0.0	4	U: 0.0 D: 0.0 L: 0.0 R: 0.0
5	U: 0.0 D: 0.0 L: 0.0 R: 0.0			6	U: 0.0 D: 0.0 L: 0.0 R: 0.0	7	U: 0.0 D: 0.0 L: 0.0 R: 0.0
8	U: 0.0 D: 0.0 L: 0.0 R: 0.0	9	U: 0.0 D: 0.0 L: 0.0 R: 0.0	10	U: 0.0 D: 0.0 L: 0.0 R: 0.0	11	U: 0.0 D: 0.0 L: 0.0 R: 0.0

Step 1

$$Q(1,R) \leftarrow 0.1(-0.04+0.0)$$

1	U: 0.0 D: 0.0 L: 0.0 R: 0.0	2	U: 0.0 D: 0.0 L: 0.0 R: 0.0	3	U: 0.0 D: 0.0 L: 0.0 R: 0.0	4	U: 0.0 D: 0.0 L: 0.0 R: 0.0
5	U: 0.0 D: 0.0 L: 0.0 R: 0.0			6	U: 0.0 D: 0.0 L: 0.0 R: 0.0	7	U: 0.0 D: 0.0 L: 0.0 R: 0.0
8	U: 0.0 D: 0.0 L: 0.0 R: 0.0	9	U: 0.0 D: 0.0 L: 0.0 R: 0.0	10	U: 0.0 D: 0.0 L: 0.0 R: 0.0	11	U: 0.0 D: 0.0 L: 0.0 R: 0.0

Step 1

$$Q(1,R) \leftarrow -0.004$$

1	U: 0.0 D: 0.0 L: 0.0 R: -0.004	2	U: 0.0 D: 0.0 L: 0.0 R: 0.0	3	U: 0.0 D: 0.0 L: 0.0 R: 0.0	4	U: 0.0 D: 0.0 L: 0.0 R: 0.0
5	U: 0.0 D: 0.0 L: 0.0 R: 0.0			6	U: 0.0 D: 0.0 L: 0.0 R: 0.0	7	U: 0.0 D: 0.0 L: 0.0 R: 0.0
8	U: 0.0 D: 0.0 L: 0.0 R: 0.0	9	U: 0.0 D: 0.0 L: 0.0 R: 0.0	10	U: 0.0 D: 0.0 L: 0.0 R: 0.0	11	U: 0.0 D: 0.0 L: 0.0 R: 0.0

Step 2

$$Q(2, R) \leftarrow (1-\alpha)Q(2, R) + \alpha(R(2) + \gamma \max_{a'} Q(3, a'))$$

1	U: 0.0 D: 0.0 L: 0.0 R: -0.004	2	U: 0.0 D: 0.0 L: 0.0 R: -0.004	3	U: 0.0 D: 0.0 L: 0.0 R: 0.0	4	U: 0.0 D: 0.0 L: 0.0 R: 0.0
5	U: 0.0 D: 0.0 L: 0.0 R: 0.0			6	U: 0.0 D: 0.0 L: 0.0 R: 0.0	7	U: 0.0 D: 0.0 L: 0.0 R: 0.0
8	U: 0.0 D: 0.0 L: 0.0 R: 0.0	9	U: 0.0 D: 0.0 L: 0.0 R: 0.0	10	U: 0.0 D: 0.0 L: 0.0 R: 0.0	11	U: 0.0 D: 0.0 L: 0.0 R: 0.0

Step 3

$$Q(3, R) \leftarrow (1-\alpha)Q(3, R) + \alpha(R(3) + \gamma \max_{a'} Q(4, a'))$$

1 U: 0.0	2 U: 0.0	3 U: 0.0	4 U: 0.0
D: 0.0	D: 0.0	D: 0.0	D: 0.0
L: 0.0	L: 0.0	L: 0.0	L: 0.0
R: -0.004	R: -0.004	R: -0.004	R: 0.0
5 U: 0.0			6 U: 0.0
D: 0.0			D: 0.0
L: 0.0			L: 0.0
R: 0.0			R: 0.0
8 U: 0.0	9 U: 0.0	10 U: 0.0	11 U: 0.0
D: 0.0	D: 0.0	D: 0.0	D: 0.0
L: 0.0	L: 0.0	L: 0.0	L: 0.0
R: 0.0	R: 0.0	R: 0.0	R: 0.0

Step 4

$$Q(4,*) \leftarrow 1$$

1	U: 0.0 D: 0.0 L: 0.0 R: -0.004	2	U: 0.0 D: 0.0 L: 0.0 R: -0.004	3	U: 0.0 D: 0.0 L: 0.0 R: -0.004	4	
5	U: 0.0 D: 0.0 L: 0.0 R: 0.0			6	U: 0.0 D: 0.0 L: 0.0 R: 0.0	7	U: 0.0 D: 0.0 L: 0.0 R: 0.0
8	U: 0.0 D: 0.0 L: 0.0 R: 0.0	9	U: 0.0 D: 0.0 L: 0.0 R: 0.0	10	U: 0.0 D: 0.0 L: 0.0 R: 0.0	11	U: 0.0 D: 0.0 L: 0.0 R: 0.0

Step 5

$$Q(1, R) \leftarrow (1-\alpha)Q(1, R) + \alpha(R(s) + \gamma \max_{a'} Q(2, a'))$$

1	U: 0.0 D: 0.0 L: 0.0 R: -0.004	2	U: 0.0 D: 0.0 L: 0.0 R: -0.004	3	U: 0.0 D: 0.0 L: 0.0 R: -0.004	4	
							+1
5	U: 0.0 D: 0.0 L: 0.0 R: 0.0			6	U: 0.0 D: 0.0 L: 0.0 R: 0.0	7	U: 0.0 D: 0.0 L: 0.0 R: 0.0
8	U: 0.0 D: 0.0 L: 0.0 R: 0.0	9	U: 0.0 D: 0.0 L: 0.0 R: 0.0	10	U: 0.0 D: 0.0 L: 0.0 R: 0.0	11	U: 0.0 D: 0.0 L: 0.0 R: 0.0

Step 5

$$Q(1, R) \leftarrow (1 - \alpha)Q(1, R) + \alpha(R(s) + \gamma Q(0, 0))$$

1	U: 0.0 D: 0.0 L: 0.0 R: -0.004	2	U: 0.0 D: 0.0 L: 0.0 R: -0.004	3	U: 0.0 D: 0.0 L: 0.0 R: -0.004	4	
							+1
5	U: 0.0 D: 0.0 L: 0.0 R: 0.0			6	U: 0.0 D: 0.0 L: 0.0 R: 0.0	7	U: 0.0 D: 0.0 L: 0.0 R: 0.0
8	U: 0.0 D: 0.0 L: 0.0 R: 0.0	9	U: 0.0 D: 0.0 L: 0.0 R: 0.0	10	U: 0.0 D: 0.0 L: 0.0 R: 0.0	11	U: 0.0 D: 0.0 L: 0.0 R: 0.0

Step 5

$$Q(1,R) \leftarrow -0.0076$$

1 U: 0.0

2 U: 0.0

3 U: 0.0

4

D: 0.0

D: 0.0

D: 0.0

+1

L: 0.0

L: 0.0

L: 0.0

R: -0.0076

R: -0.004

R: -0.004

5 U: 0.0

6 U: 0.0

7 U: 0.0

D: 0.0

D: 0.0

D: 0.0

L: 0.0

L: 0.0

L: 0.0

R: 0.0

R: 0.0

R: 0.0

8 U: 0.0

9 U: 0.0

10 U: 0.0

11 U: 0.0

D: 0.0

D: 0.0

D: 0.0

D: 0.0

L: 0.0

L: 0.0

L: 0.0

L: 0.0

R: 0.0

R: 0.0

R: 0.0

R: 0.0

SARSA:

State-Action-Reward-State-Action

Modify Q learning to use chosen action, a' , not max

- $Q_{t+1}(s,a) \leftarrow (1-\alpha)Q_t(s,a) + \alpha [R(s,a,s') + \gamma \max_{a'} Q_t(s',a')]$
- $Q_{t+1}(s,a) \leftarrow (1-\alpha)Q_t(s,a) + \alpha [R(s,a,s') + \gamma Q_t(s',a')]$

On-policy, instead of off-policy

- SARSA learns based on real behavior, not optimal behavior
 - Good if real world provides obstacles to optimal behavior
- Q learning learns optimal behavior, beyond real behavior
 - Good if training phase has obstacles that won't persist

SARSA: State-Action-Reward-State-Action

Initialize all $Q(s, a)$ arbitrarily

For all episodes

Initalize s

Choose a using policy derived from Q , e.g., ϵ -greedy

Repeat

 Take action a , observe r and s'

 Choose a' using policy derived from Q , e.g., ϵ -greedy

 Update $Q(s, a)$:

$$Q(s, a) \leftarrow Q(s, a) + \eta(r + \gamma Q(s', a') - Q(s, a))$$

$s \leftarrow s'$, $a \leftarrow a'$

Until s is terminal state

Step 5: SARSA

$$Q(1, R) \leftarrow (1 - \alpha)Q(1, R) + \alpha(R(s) + \gamma Q(2, R))$$

1	U: 0.0 D: 0.0 L: 0.0 R: -0.004	2	U: 0.0 D: 0.0 L: 0.0 R: -0.004	3	U: 0.0 D: 0.0 L: 0.0 R: -0.004	4	
							+1
5	U: 0.0 D: 0.0 L: 0.0 R: 0.0			6	U: 0.0 D: 0.0 L: 0.0 R: 0.0	7	U: 0.0 D: 0.0 L: 0.0 R: 0.0
8	U: 0.0 D: 0.0 L: 0.0 R: 0.0	9	U: 0.0 D: 0.0 L: 0.0 R: 0.0	10	U: 0.0 D: 0.0 L: 0.0 R: 0.0	11	U: 0.0 D: 0.0 L: 0.0 R: 0.0

Step 5: SARSA

$Q(1, R) \leftarrow -0.008$ which is closer to -0.04 than -0.076

1	U: 0.0 D: 0.0 L: 0.0 R: -0.008	2	U: 0.0 D: 0.0 L: 0.0 R: -0.004	3	U: 0.0 D: 0.0 L: 0.0 R: -0.004	4	
							+1
5	U: 0.0 D: 0.0 L: 0.0 R: 0.0			6	U: 0.0 D: 0.0 L: 0.0 R: 0.0	7	U: 0.0 D: 0.0 L: 0.0 R: 0.0
8	U: 0.0 D: 0.0 L: 0.0 R: 0.0	9	U: 0.0 D: 0.0 L: 0.0 R: 0.0	10	U: 0.0 D: 0.0 L: 0.0 R: 0.0	11	U: 0.0 D: 0.0 L: 0.0 R: 0.0

Q Learning & SARSA

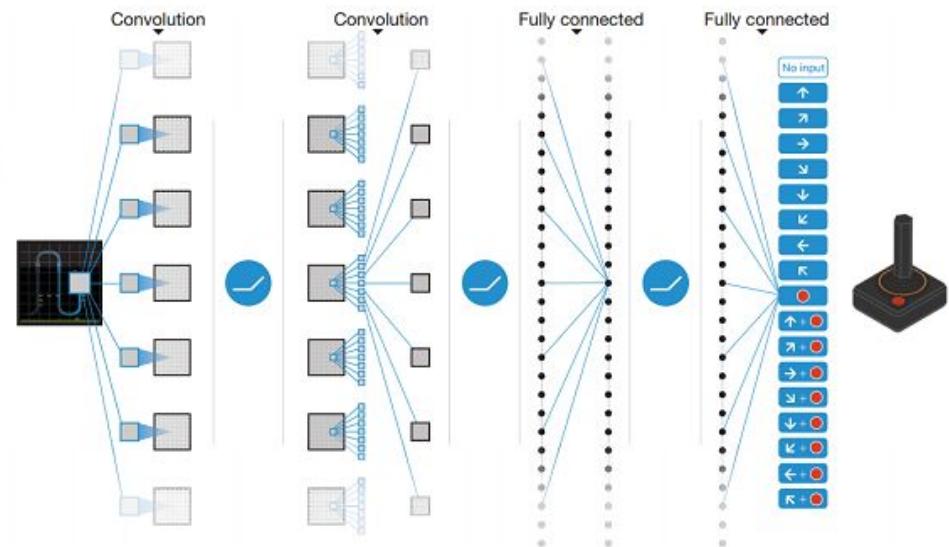
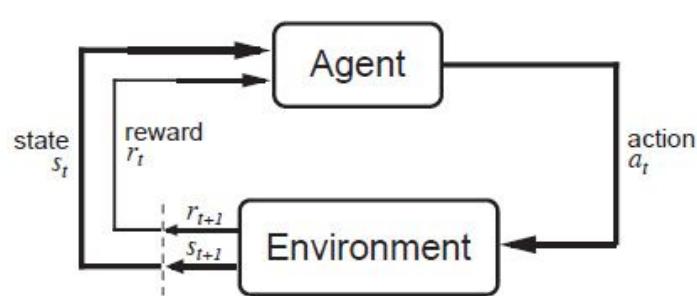
Converge to correct values

- Assuming agent tries all actions, in all states, many times
 - Otherwise, there won't be enough experience to learn Q
- And if α decays over time
 - Similar to simulated annealing

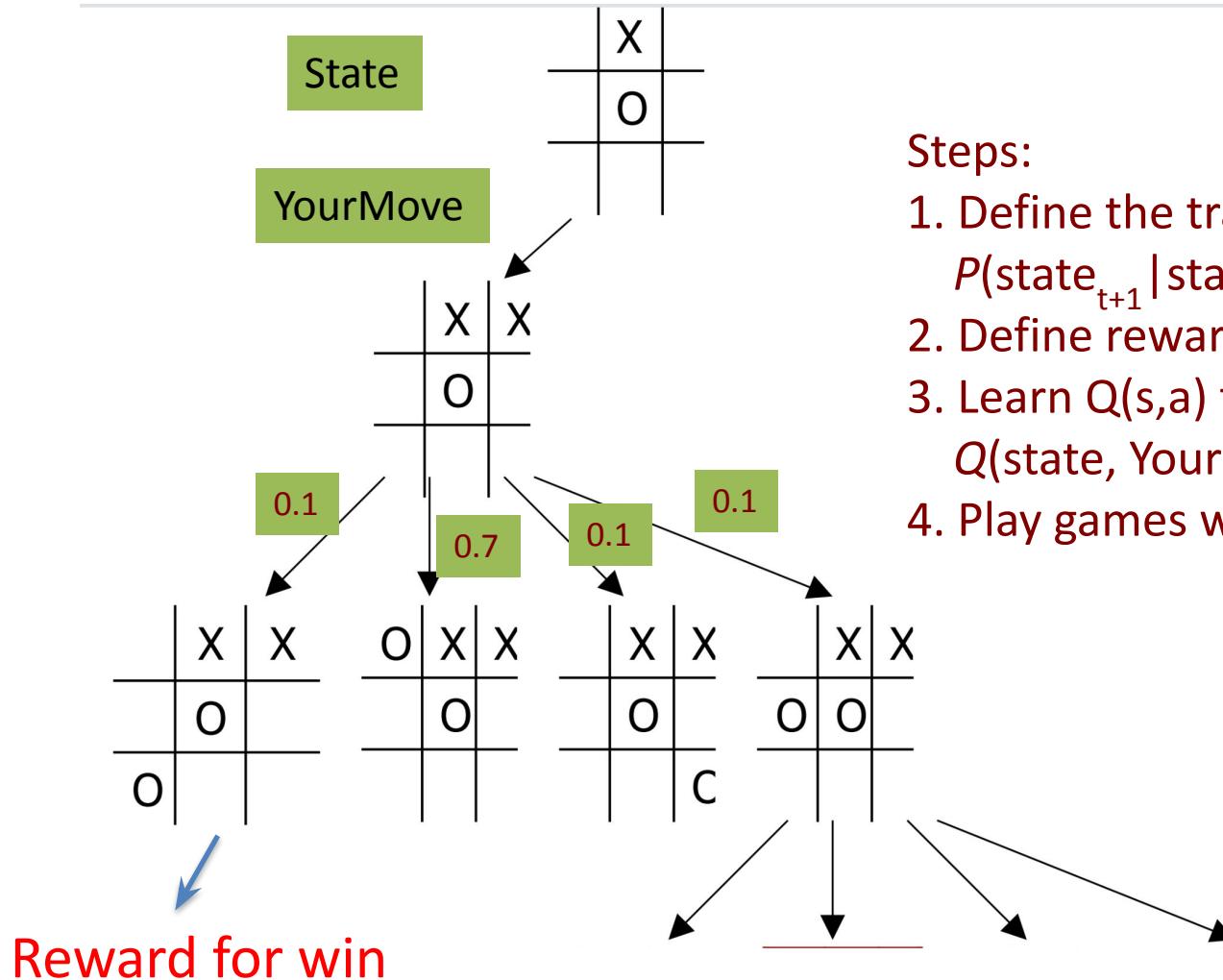
Avoids the complexity of solving an MDP

- MDP requires solving for the optimal policy before starting
- An RL agent can start right away and then learn as it goes
 - Although this might be wasteful if you already know P and R

Q-Learning for Game Playing



Q-Learning for Game Playing



Steps:

1. Define the transition model
 $P(\text{state}_{t+1} | \text{state}_t, \text{YourMove})$
2. Define rewards
3. Learn $Q(s,a)$ from rewards
 $Q(\text{state}, \text{YourMove})$
4. Play games with learned Q

A Key Question

- In all search and game playing problems, what is the key information that you wish to have for finding the optimal solution?
 - You wish to have but you may not always have
 - **answer: the true future value**
- How: Play a lot of games and
 - Learn the state utility values for winning the games
 - Learn the Q values for winning the games

Exploitation vs. Exploration

- RL algorithms do not specify how actions are chosen by the agent
- One strategy would be in state s to select the action a that maximizes $Q(s,a)$, thereby *exploiting* its current approximation Q .
 - Using this strategy the agent runs the risk that it will overcommit to actions that are found during early training to have high Q values, while failing to *explore* other actions that have even higher values.
 - It is common in Q learning to use a probabilistic approach to selecting actions.
 - Actions with higher Q values are assigned higher probabilities, but every action is assigned a nonzero probability. One way to assign such probabilities is where $P(a_i | s)$ is the probability T of selecting action a_i , given that the agent is in state s , and where $T > 0$ is a constant that determines how strongly the selection favors actions with high Q values.

$$P(a_i | s) = \frac{e^{\hat{Q}(s,a_i)/T}}{\sum_j e^{\hat{Q}(s,a_j)/T}}$$

Exploitation vs. Exploration

- Hence it is good to try new things now and then
 - If T is large, then do more **exploration**,
 - If T is small, then follow or **exploit** the current policy
- One can decrease T over time to first explore, and then converge and exploit
 - For example $T = c/k+d$ where k is iteration of the algorithm

$$P(a_i | s) = \frac{e^{\hat{Q}(s, a_i)/T}}{\sum_j e^{\hat{Q}(s, a_j)/T}}$$

- Decreasing T over time is also known as **simulated annealing**, which is inspired by annealing process in nature. T is also known as **Temperature**

CSCI 561

Foundation for Artificial Intelligence

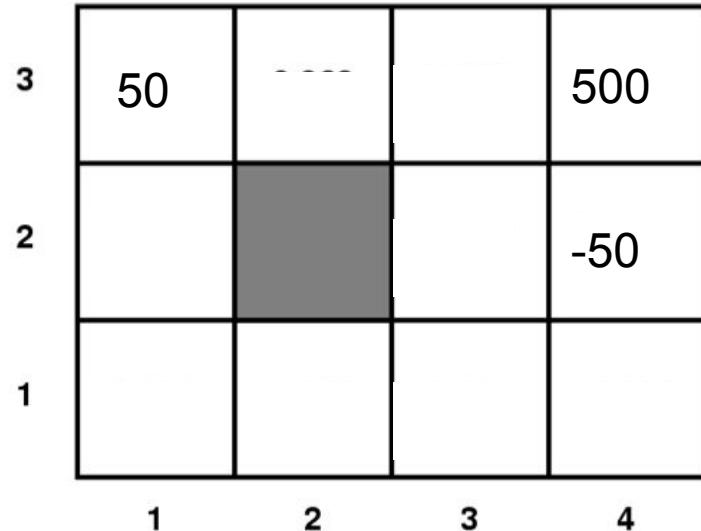
Discussion Section

(Week 4)

PROF WEI-MIN SHEN WMSHEN@USC.EDU

MDP

Given a Gridworld domain, where terminal states (1,3), (4,3), and (4,2) have rewards 50, 500, and -50 respectively, the set of possible actions are {N,E,S,W, or X for terminal states}, the agent moves deterministically, all V and Q values for non-terminal states have been initialized to 0.0, answer the questions below.



Circle the letter that corresponds to the best answer for the question.

What are the optimal utility values V for each state in the above grid if $\gamma = 0.5$, $c(a)=0$, $R(s)=0$ for the non-terminal states?

(Remember $V_{t+1}(s) = R(s) + \text{Max}_{a \in A} \{c(a) + \gamma \sum_{s' \in S} P(s'|a,s) V_t(s')\}$)

- a. $V_{(1,1)}=15.75, V_{(1,2)}=25, V_{(2,1)}=31.25, V_{(2,3)}=125, V_{(3,1)}=62.5, V_{(3,2)}=125, V_{(3,3)}=250, V_{(4,1)}=25$
- b. $V_{(1,1)}=12.5, V_{(1,2)}=25, V_{(2,1)}=31.25, V_{(2,3)}=125, V_{(3,1)}=62.5, V_{(3,2)}=125, V_{(3,3)}=250, V_{(4,1)}=31.25$
- c. $V_{(1,1)}=15.625, V_{(1,2)}=25, V_{(2,1)}=31.25, V_{(2,3)}=125, V_{(3,1)}=62.5, V_{(3,2)}=125, V_{(3,3)}=250, V_{(4,1)}=31.25$
- d. $V_{(1,1)}=12.5, V_{(1,2)}=25, V_{(2,1)}=25, V_{(2,3)}=25, V_{(3,1)}=50, V_{(3,2)}=100, V_{(3,3)}=250, V_{(4,1)}=25$
- e. None of the above

Q-Learning

What are the Q values of state (3,2) in the above grid if
 $\gamma = 0.5$, $c(a)=0$, $R(s)=-2$ for non-terminal states?

(Remember $Q_{t+1}(a,s) = R(s) + \gamma \sum_{s' \in S} P(s'|a,s) \max_{a' \in A} Q_t(a's')$)

- a. $Q_{((3,2),N)}=122$, $Q_{((3,2),E)}=-27$, $Q_{((3,2),S)}=59$
- b. $Q_{((3,2),N)}=122$, $Q_{((3,2),E)}=-27$, $Q_{((3,2),S)}=27.5$
- c. $Q_{((3,2),N)}=125$, $Q_{((3,2),E)}=-25$, $Q_{((3,2),S)}=62.5$
- d. $Q_{((3,2),N)}=120$, $Q_{((3,2),E)}=-27$, $Q_{((3,2),S)}=31.5$
- e. None of the above

The grid diagram shows a 4x4 grid with states labeled by row (1 to 3) and column (1 to 4). The top-right cell (3,4) contains 500, the bottom-right cell (4,4) contains -50, and the top-left cell (1,1) contains 50. The middle-right cell (3,2) is shaded gray. Row 1 has values 1, 2, 3, 4. Row 2 has values 50, blank, blank, 500. Row 3 has values blank, blank, -50, blank. Column 1 has values 1, blank, blank, blank. Column 2 has values blank, 2, blank, blank. Column 3 has values blank, blank, 3, blank. Column 4 has values blank, blank, blank, 4.

Q-Learning

What are the Q values of state (3,2) in the above grid if $\gamma = 0.5$, $c(a)=0$, $R(s)=-2$ for non terminal states?

(Remember $Q_{t+1}(a,s) = R(s) + c(a) + \gamma \sum_{s' \in S} P(s'|a,s) \max_{a' \in A} Q_t(a's')$)

- a. $Q_{((3,2),N)}=122$, $Q_{((3,2),E)}=-27$, $Q_{((3,2),S)}=59$
- b. $Q_{((3,2),N)}=122$, $Q_{((3,2),E)}=-27$, $Q_{((3,2),S)}=27.5$ ³
- c. $Q_{((3,2),N)}=125$, $Q_{((3,2),E)}=-25$, $Q_{((3,2),S)}=62.5$
- d. $Q_{((3,2),N)}=120$, $Q_{((3,2),E)}=-27$, $Q_{((3,2),S)}=31.5$ ²
- e. None of the above

50		E248	500
			-50
1	2	3	4

Q-Learning

What are the Q values of state (3,2) in the above grid if $\gamma = 0.5$, $c(a)=0$, $R(s)=-2$ for non terminal states?

(Remember $Q_{t+1}(a,s) = R(s) + c(a) + \gamma \sum_{s' \in S} P(s'|a,s) \max_{a' \in A} Q_t(a's')$)

- a. $Q_{((3,2),N)}=122$, $Q_{((3,2),E)}=-27$, $Q_{((3,2),S)}=59$
- b. $Q_{((3,2),N)}=122$, $Q_{((3,2),E)}=-27$, $Q_{((3,2),S)}=27.5$ ³
- c. $Q_{((3,2),N)}=125$, $Q_{((3,2),E)}=-25$, $Q_{((3,2),S)}=62.5$
- d. $Q_{((3,2),N)}=120$, $Q_{((3,2),E)}=-27$, $Q_{((3,2),S)}=31.5$ ²
- e. None of the above

50		E248	500
		N122	-50
1	2	3	4

Q-Learning

What are the Q values of state (3,2) in the above grid if $\gamma = 0.5$, $c(a)=0$, $R(s)=-2$ for non terminal states?

(Remember $Q_{t+1}(a,s) = R(s) + c(a) + \gamma \sum_{s' \in S} P(s'|a,s) \max_{a' \in A} Q_t(a's')$)

- a. $Q_{((3,2),N)}=122$, $Q_{((3,2),E)}=-27$, $Q_{((3,2),S)}=59$
- b. $Q_{((3,2),N)}=122$, $Q_{((3,2),E)}=-27$, $Q_{((3,2),S)}=27.5$ ³
- c. $Q_{((3,2),N)}=125$, $Q_{((3,2),E)}=-25$, $Q_{((3,2),S)}=62.5$
- d. $Q_{((3,2),N)}=120$, $Q_{((3,2),E)}=-27$, $Q_{((3,2),S)}=31.5$ ²
- e. None of the above

50		E248	500
		N122 S27.5	-50
		N59	

1 2 3 4

B

AlphaGo Documentary

This is a great [Link](#) documentary on the AlphaGo agent developed by Google Deepmind vs. the world champion Lee Sedol. If you have some time to spare definitely watch it, it's worth the time. You'll get a better understanding of the GO game and what it takes to beat the world champion in the hardest board game on earth.

Additionally, DeepMind also developed AlphaGo Zero which learned the game of GO without any human intervention and beat AlphaGo by 100 games to 0. For more info:

<https://www.deepmind.com/blog/alphago-zero-starting-from-scratch>

CSCI 561 - Foundation for Artificial Intelligence

Discussion Section (Week 5) Midterm-1 Review

PROF WEI-MIN SHEN WMSHEN@USC.EDU

Material covered by midterm 1

- **Covers everything studied in class up to and including CSP and RL (not include “logic”)**
- **Lectures vs book: what to know?**
 - If something is covered both in the book and the slides of lecture/discussion: use the slides.
 - If something is covered in the book only and was not covered at all in the lecture/discussions: you do not need to know it.
 - If something is covered in the book and in the slides of lecture/discussion but with additional details provided in the book: **you need to know both**, and use the slides for the overlapping parts.

Midterm 1 Instructions:

- Oct 2, 2022, 5:00PM, in SGM124 or DEN
- Maximum credits/points for this midterm: 100 points
- Credits/points for each question is indicated on the question
- Closed book
- No books or any other material are allowed
- Please practice in your sample exam-0 on DEN
- Please get your camera checked by a TA
- Please test your lockdown browser and make sure you know how to enter your answers
- No technical questions during the exam
- Be brief: a few words are often enough if they are precise and use the correct vocabulary studied in class
- Make sure your environment for the exam is quite/exclusive

Sample exam Questions

- 1. General AI**
- 2. Search Concepts**
- 3. Comparing Strategies**
- 4. Game Playing**
- 5. CSP**
- 6. Reinforcement Learning (RL)**

Note: The actual exam questions will be different or harder/easier than those distributed in the sample exam-1.

F The Turing test defines the conditions under which a machine can be said to be “intelligent”.

F A* is an admissible algorithm.

F DFS is faster than BFS.

T DFS has lower asymptotic space complexity than BFS.

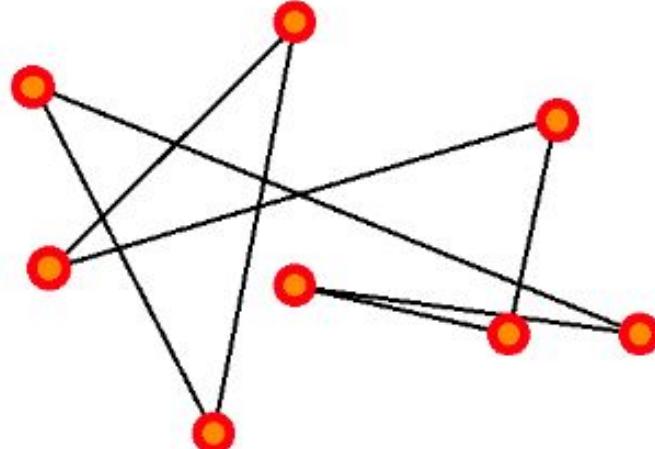
F When using the correct temperature decrease schedule, simulated annealing is guaranteed to find the global optimum in finite time.

- F** Alpha-beta pruning accelerates game playing at the cost of being an approximation to full minimax.
- F** Hill-climbing is an entirely deterministic algorithm.
- T** The exact evaluation function values do not affect minimax decision as long as the ordering of these values is maintained.
- F** A perfectly rational backgammon-playing agent never loses
- T** Hill climbing search is best used for problem domains with densely packed goals

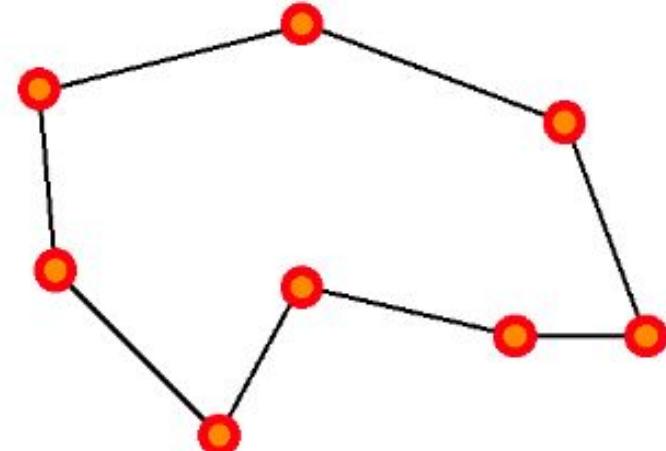
- A suitable representation for states: permutation of all cities in the tour
 $\langle A, B, C, D, E \rangle$
- The initial state of the problem: random permutation of all cities
- A good goal test to use in this problem: minimize the distance travelled
- Good operators to use for search: permute 2 cities
- Which search algorithm would be the most appropriate to use here if we want to minimize the distance of the tour found?

Local Search - GA/SA/hill climbing, etc...

Suboptimal solution (long path)



Optimal solution

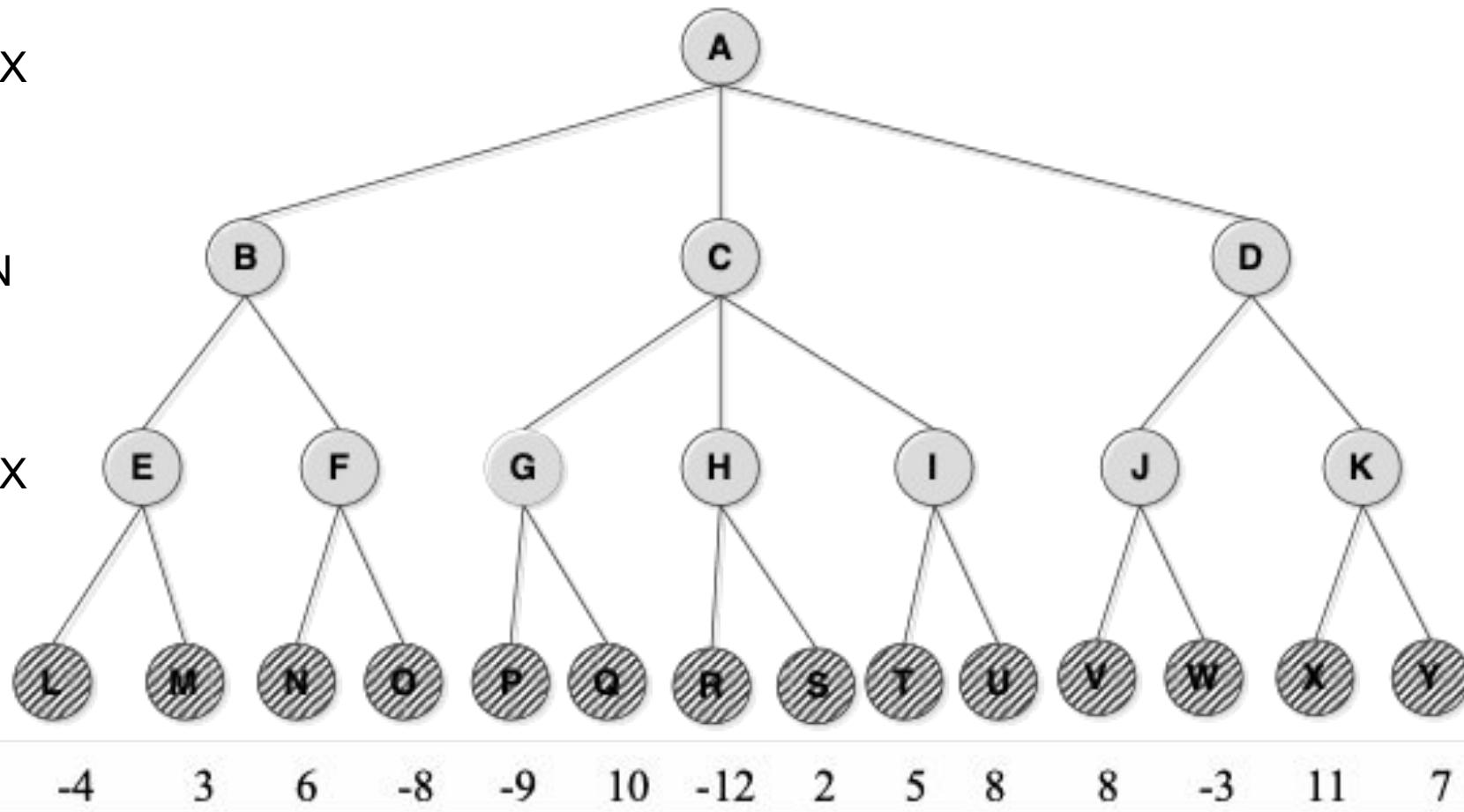


Minimax

MAX

MIN

MAX



MAX

$\alpha = -5$
 $\beta = +\infty$
 $v = 5$

MIN

$\alpha = -\infty$
 $\beta = 4$
 $v = 4$

$\alpha = 4$
 $\beta = 9$
 $v = 2$

α : at least max can get this much
 β : at most min will give this much
 Keep update $[\alpha, \beta]$, if v is in range,
 or else cut the branches

MAX

$\alpha = 4$
 $\beta = +\infty$
 $v = 4$

E

$\alpha = -\infty$
 $\beta = 4$
 $v = 6$

F

G

H

I

J

K

4

3

6

2

9

5

2

1

3

1

5

4

7

5

Max Node

for each a in ACTIONS(s) do

$v \leftarrow \text{MAX}(v, \text{MIN-VALUE}(\text{RESULT}(s, a), \alpha, \beta))$

if $v \geq \beta$ then return v

$\alpha \leftarrow \text{MAX}(\alpha, v)$

return v

Min Node

for each a in ACTIONS(s) do

$v \leftarrow \text{MIN}(v, \text{MAX-VALUE}(\text{RESULT}(s, a), \alpha, \beta))$

if $v \leq \alpha$ then return v

$\beta \leftarrow \text{MIN}(\beta, v)$

return v

The schedules of the customers are:

Company 1: Webflix: 8:00-9:00am

Company 2: Amazon: 8:30-9:30am

Company 3: Pied Piper: 9:00-10:00am

Company 4: Hooli: 9:00-10:00am

Company 5: Gulu: 9:30-10:30am

The profiles of your engineers are:

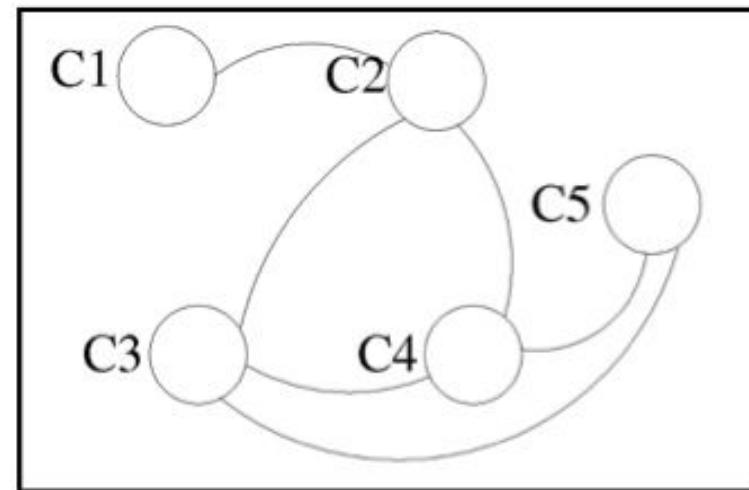
- 1) Albacore can maintain Pied Piper and Hooli.
- 2) Bosam can maintain all companies, but Webflix.
- 3) Coleslaw can maintain all companies.

- Using Company as variable, formulate this problem as a CSP problem with variables, domains, and constraints. Constraints should be specified formally and precisely, but may be implicit rather than explicit.
- Draw the constraint graph associated with your CSP.

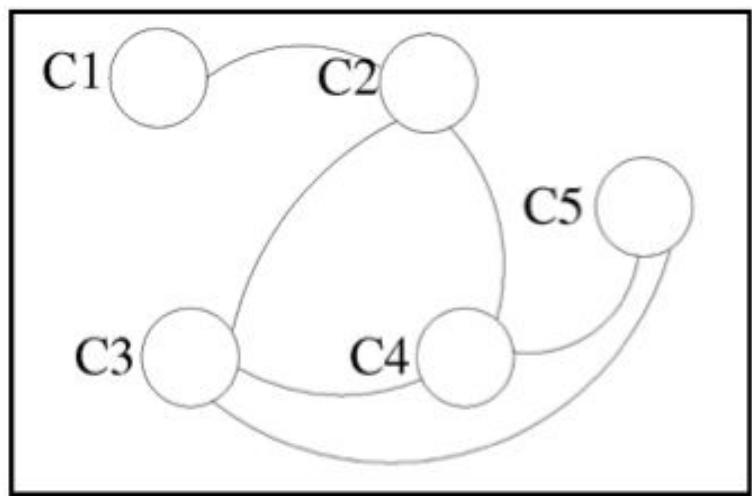
Variable	Domain
C1	C
C2	BC
C3	ABC
C4	ABC
C5	BC

Constraints:

$C1 \neq C2, C2 \neq C3, C3 \neq C4, C4 \neq C5, C2 \neq C4, C3 \neq C5.$



Variable	Domain
C1	C
C2	BC
C3	ABC
C4	ABC
C5	BC



Constraints:

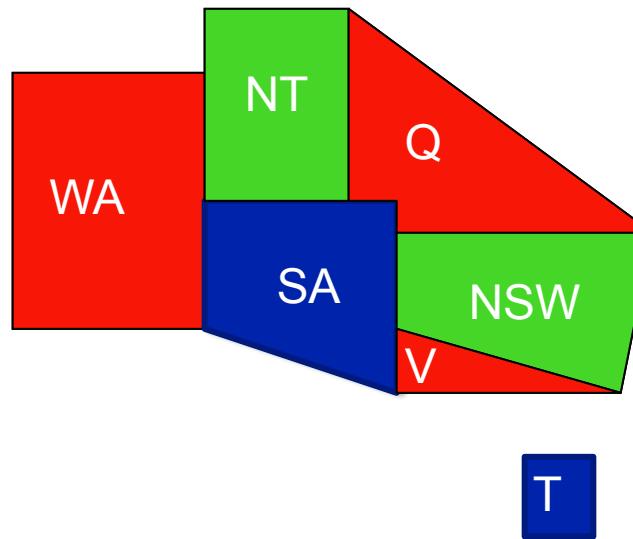
$C1 \neq C2, C2 \neq C3, C3 \neq C4, C4 \neq C5, C2 \neq C4, C3 \neq C5.$

- Show the domains of the variables after running arc-consistency on this initial graph (after having already enforced any unary constraints).
- Give one solution to this CSP.

C1 = C, C2 = B, C3 = C, C4 = A, C5 = B.

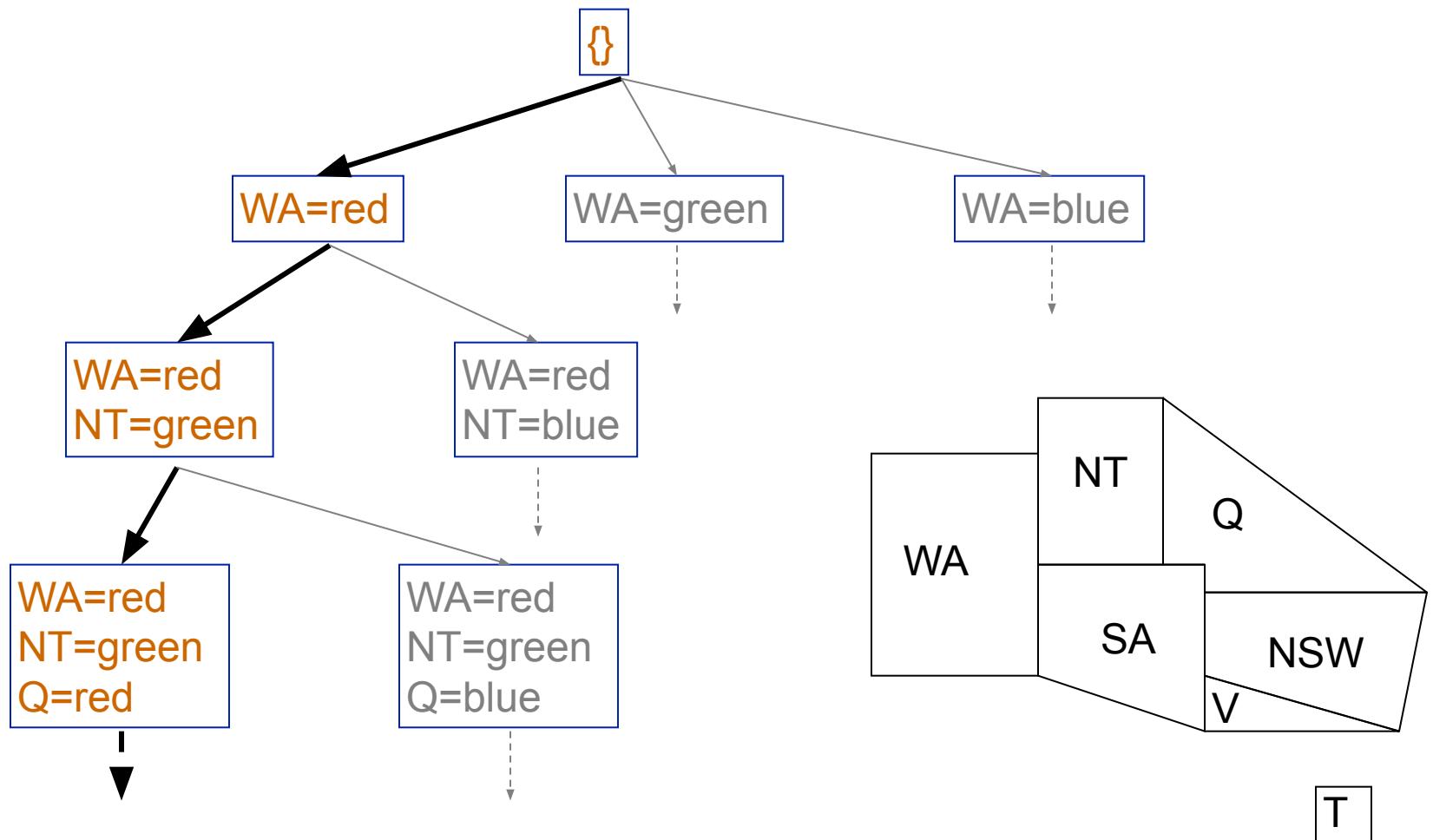
Variable	Domain
C1	C
C2	B
C3	AC
C4	AC
C5	BC

CSP Example: Map Coloring

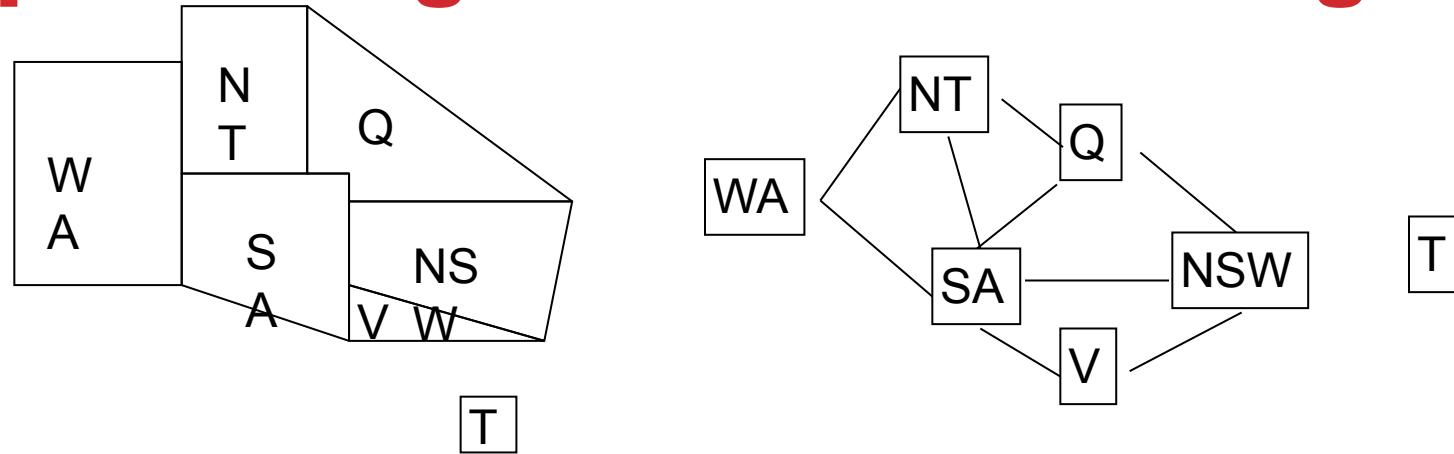


- 7 variables {WA, NT, SA, Q, NSW, V, T}
- Each variable has the same domain {red, green, blue}
- No two adjacent variables have the same value:
WA ≠ NT, WA ≠ SA, NT ≠ SA, NT ≠ Q, SA ≠ Q, SA ≠ NSW, SA ≠ V, Q ≠ NSW, NSW ≠ V

Backtracking Search: Map Coloring

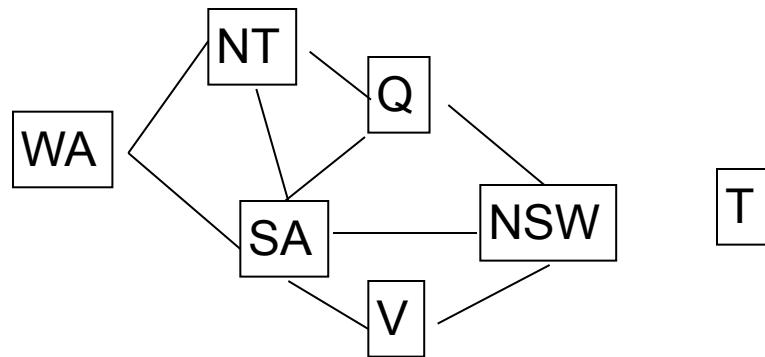


Map Coloring: Forward Checking



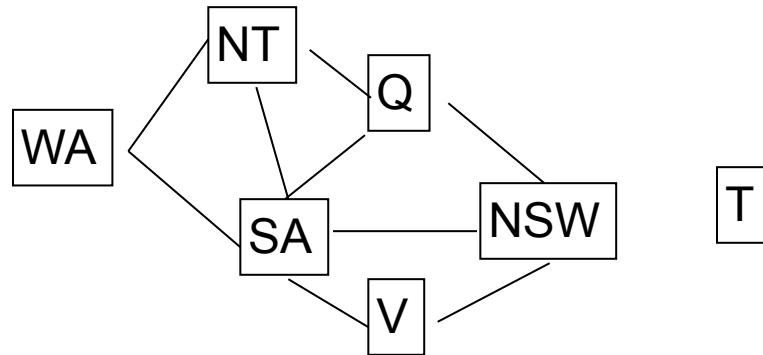
WA	NT	Q	NSW	V	SA	T
RGB	RGB	RGB	RGB	RGB	RGB	RGB
1: R	RGB	RGB	RGB	RGB	RGB	RGB

Map Coloring: Forward Checking



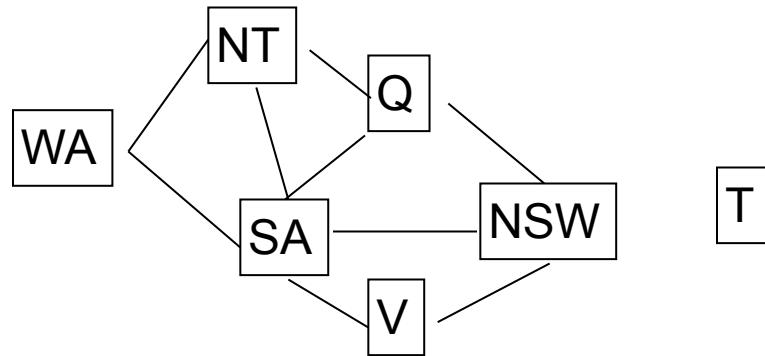
WA	NT	Q	NSW	V	SA	T
RGB	RGB	RGB	RGB	RGB	RGB	RGB
1: R	XRGB	RGB	RGB	RGB	XRGB	RGB

Map Coloring: FC



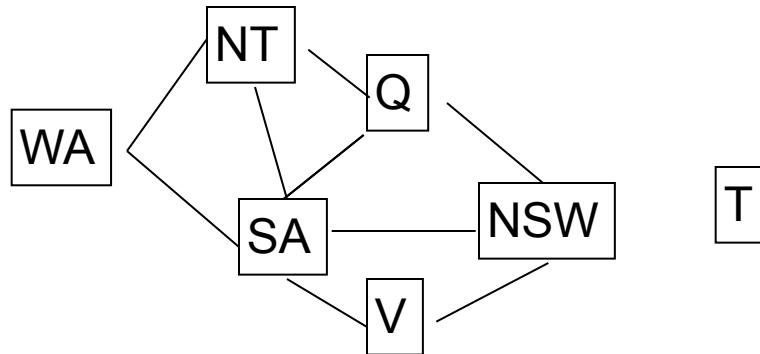
WA	NT	Q	NSW	V	SA	T
RGB	RGB	RGB	RGB	RGB	RGB	RGB
R	XRGB	RGB	RGB	RGB	XRGB	RGB
R	XRGB	2: G	RGB	RGB	XRGB	RGB

Map Coloring: FC



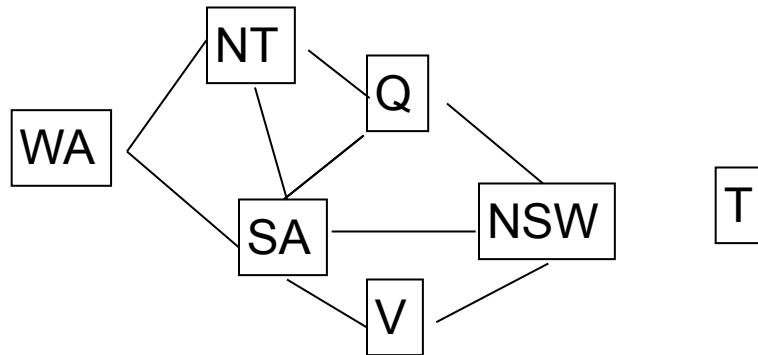
WA	NT	Q	NSW	V	SA	T
RGB	RGB	RGB	RGB	RGB	RGB	RGB
R	XRGB	RGB	RGB	RGB	XRGB	RGB
R	XRGB	2: G	XRGB	RGB	XRGB	RGB

Map Coloring: FC



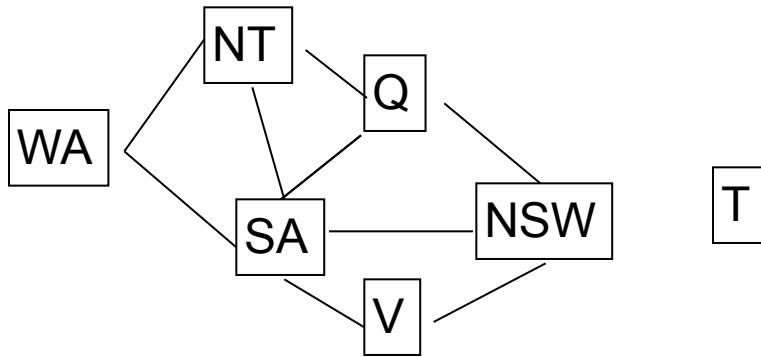
WA	NT	Q	NSW	V	SA	T
RGB	RGB	RGB	RGB	RGB	RGB	RGB
R	XRGB	RGB	RGB	RGB	XRGB	RGB
R	XRGB	G	XRGB	RGB	XRGB	RGB
R	XRGB	G	XRGB	3:B	XRGB	RGB

Map Coloring: FC



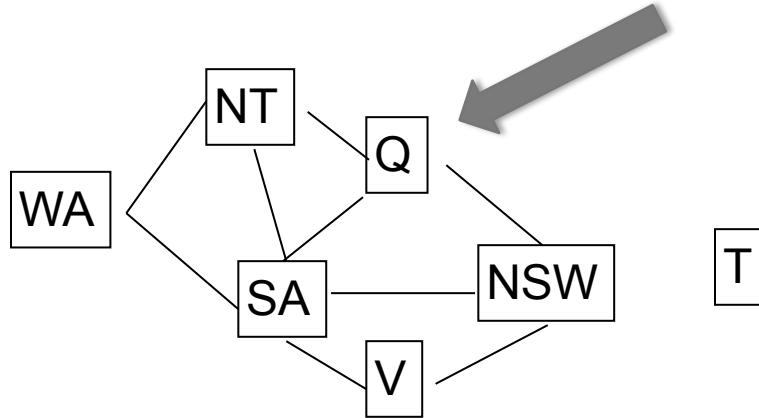
WA	NT	Q	NSW	V	SA	T
RGB	RGB	RGB	RGB	RGB	RGB	RGB
R	XRGB	RGB	RGB	RGB	XRGB	RGB
R	XRGB	G	XRGB	RGB	XRGB	RGB
R	XRGB	G	XRGB	3:B	XRGB	RGB

Other inconsistencies



WA	NT	Q	Impossible assignments that forward checking does not detect				
RGB	RGB	RGB	RGB	RGB	RGB	RGB	RGB
R	X RGB	RGB	RGB	RGB	RGB	X RGB	RGB
R	X RGB	X RGB	G	X RGB	RGB	X RGB	RGB
R	X RGB	X RGB	G	X RGB	3:B	X RGB	RGB

Map Coloring: Constraint Propagation

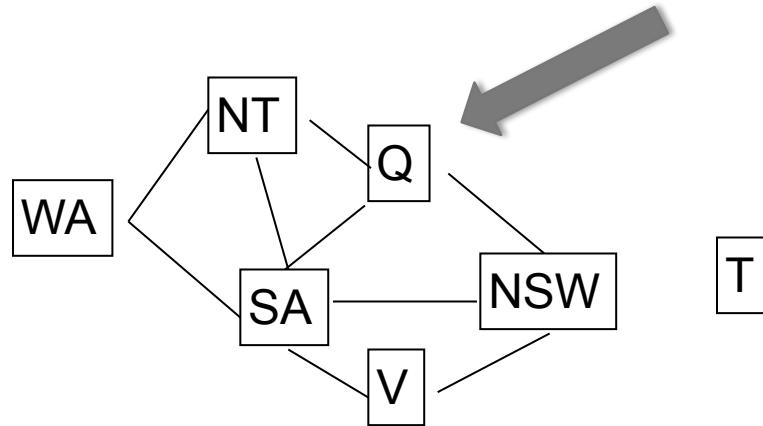


WA	NT	Q	NSW	V	SA	T
RGB	RGB	RGB	RGB	RGB	RGB	RGB
R	XRGB	RGB	RGB	RGB	XRGB	RGB
R	XRGB	2: G	RGB	RGB	XRGB	RGB



Go back to assigning
“GREEN” to Queensland

Map Coloring: CP

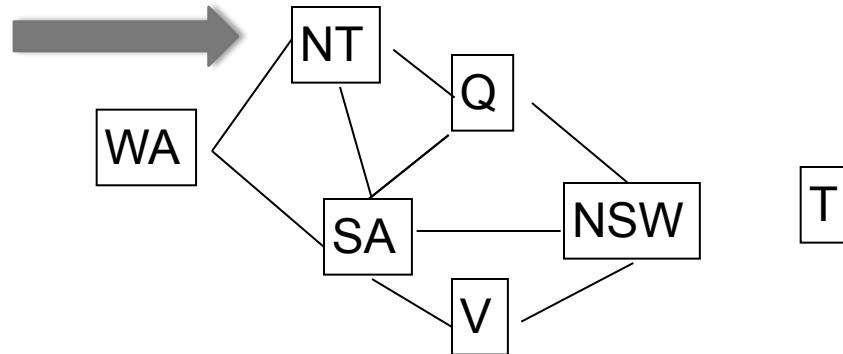


WA	NT	Q	NSW	V	SA	T
RGB	RGB	RGB	RGB	RGB	RGB	RGB
R	XRGB	RGB	RGB	RGB	XRGB	RGB
R	XRGB	2: G	XRGB	RGB	XRGB	RGB



Immediate propagation removes GREEN for NSW, SA & NT

Map Coloring: CP

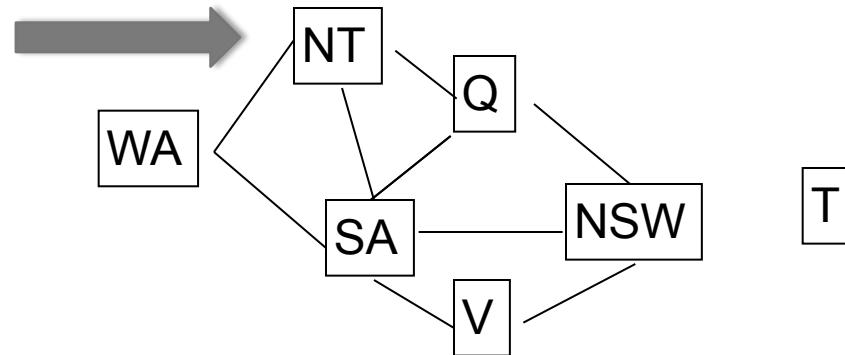


WA	NT	Q	NSW	V	SA	T
RGB	RGB	RGB	RGB	RGB	RGB	RGB
R	XRGB	RGB	RGB	RGB	XRGB	RGB
R	XRGB	G	XRGB	RGB	XRGB	RGB



Since possible values for NT changed, continue to check arc consistency from NT

Map Coloring: CP



WA	NT	Q	NSW	V	SA	T
RGB	RGB	RGB	RGB	RGB	RGB	RGB
R	XRGB	RGB	RGB	RGB	XRGB	RGB
R	XRGB	G	XRGB	RGB	XRGB	RGB

Constraint $NT \neq SA$

Constraint violation with SA
immediately detected

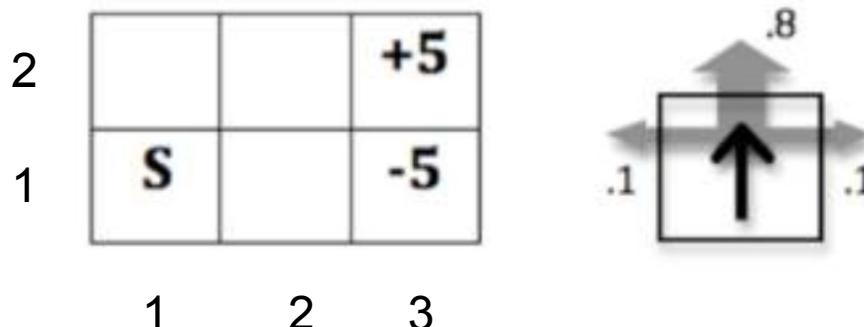
MDP and Q-Learning

The grid world MDP shown below operates like to the one we saw in class.

The states are grid squares, identified by their row and column number (row first).

The agent always starts in state (1,1), marked with the letter S. There are two terminal goal states, (2,3) with reward +5 and (1,3) with reward -5. Rewards are 0 in non-terminal states. (The reward for a state is received as the agent moves into the state.)

The transition function is such that the intended agent movement (North, South, West, or East) happens with probability .8. With probability .1 each, the agent ends up in one of the states perpendicular to the intended direction. If a collision with a wall happens, the agent stays in the same state.



MDP and Q-Learning

9.a. [4%] Draw the optimal policy for this grid. Draw it directly on the grid world above.

9.b. [6%] Suppose the agent knows the transition probabilities. Give the first two rounds of value iteration updates for each state, with a discount of 0.9. (Assume V_0 is 0 everywhere and compute V_i for times $i = 1, 2$).

State	(1,1)	(1,2)	(1,3)	(2,1)	(2,2)	(2,3)
V_0	0	0	0	0	0	0
V_1						
V_2						

$$U_{i+1}(s) \leftarrow R(s) + \gamma \max_a \sum_{s'} T(s, a, s') U_i(s')$$

MDP and Q-Learning

9.a. [4%] Draw the optimal policy for this grid. Draw it directly on the grid world above.

9.b. [6%] Suppose the agent knows the transition probabilities. Give the first two rounds of value iteration updates for each state, with a discount of 0.9. (Assume V_0 is 0 everywhere and compute V_i for times $i = 1, 2$).

State	(1,1)	(1,2)	(1,3)	(2,1)	(2,2)	(2,3)
v_0	0	0	0	0	0	0
v_1	0	0	-5	0	0	+5
v_2						

$$U_{i+1}(s) \leftarrow R(s) + \gamma \max_a \sum_{s'} T(s, a, s') U_i(s')$$

MDP and Q-Learning

9.a. [4%] Draw the optimal policy for this grid. Draw it directly on the grid world above.

9.b. [6%] Suppose the agent knows the transition probabilities. Give the first two rounds of value iteration updates for each state, with a discount of 0.9. (Assume V_0 is 0 everywhere and compute V_i for times $i = 1, 2$).

State	(1,1)	(1,2)	(1,3)	(2,1)	(2,2)	(2,3)
v_0	0	0	0	0	0	0
v_1	0	0	-5	0	0	+5
v_2	0	0	-5	0		+5

$$U_{i+1}(s) \leftarrow R(s) + \gamma \max_a \sum_{s'} T(s, a, s') U_i(s')$$

$0 + 0.9 * \max\{$
 left $0.8*0+0.1*0+0.1*0,$
 up $0.1*0+0.8*0+0.1*5,$
 down $0.1*0+0.8*0+0.1*5,$
 right $0.1*0+0.1*0+0.8*5\}$
 $= 0 + 0.9 * \{4.0\} = 3.6$