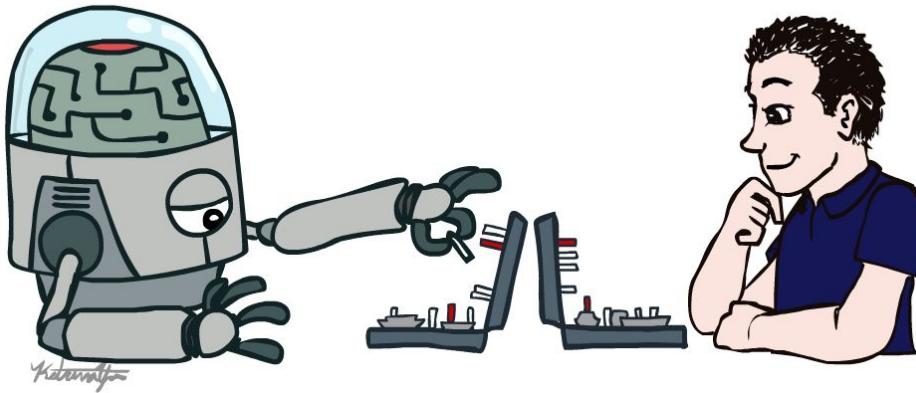
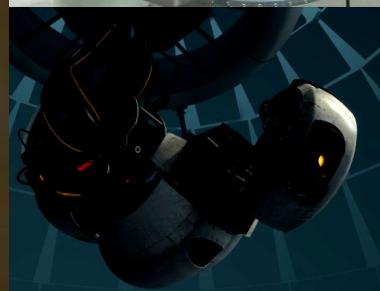
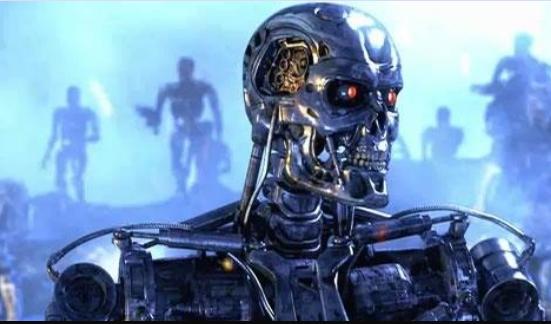


Artificial Intelligence



Sci-Fi AI?





TUG
CAUTION
MAY CONTAIN
CHEMOTHERAPY DRUG

CAUTION
MAY CONTAIN
CHEMOTHERAPY DRUG



Today

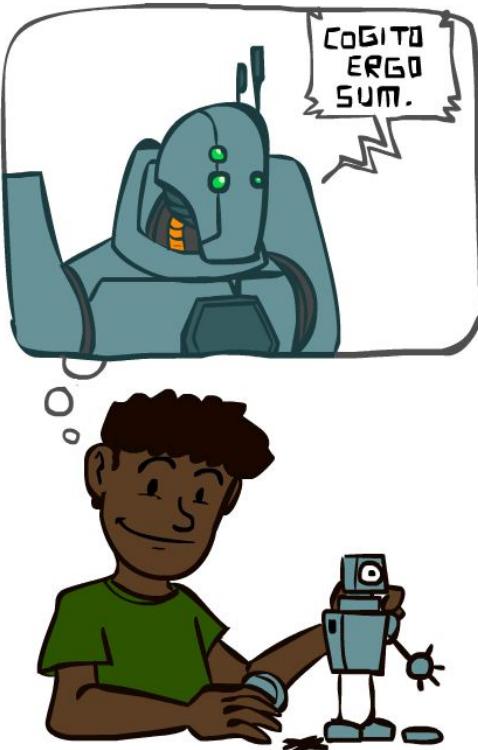
- What is this course?
- What is artificial intelligence?
- History of AI
- What can AI do?



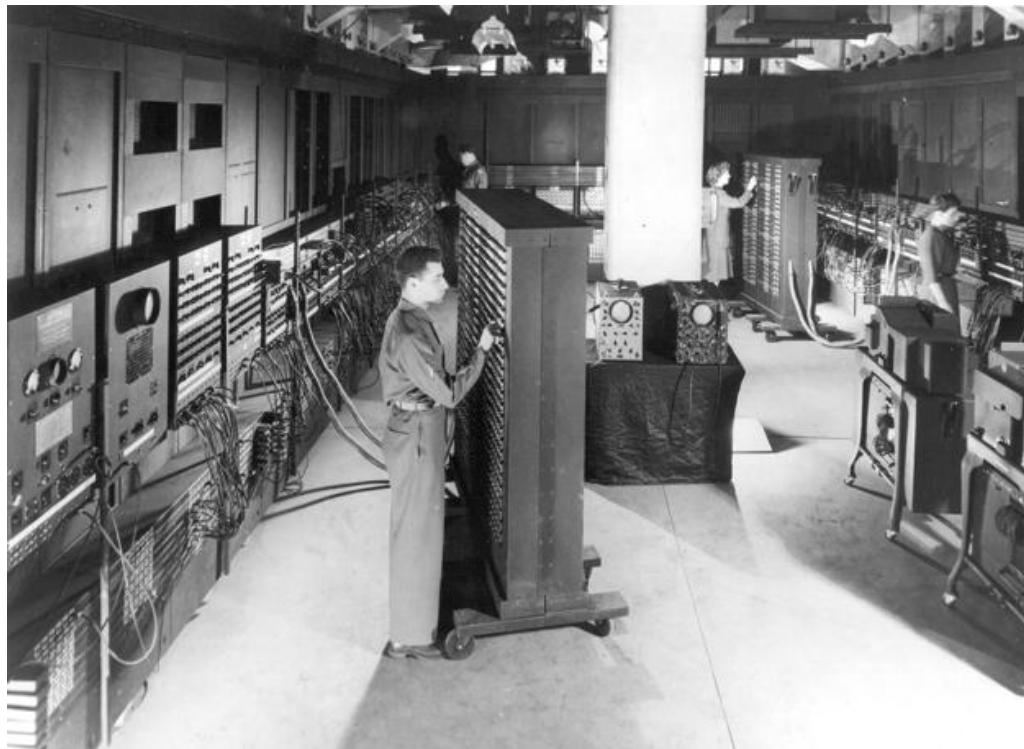
About Course

- *Text Book:*
 - *Artificial Intelligence: A Modern Approach. Third Edition.* – Stuart Russell and Peter Norvig (**AIMA**)
- *References:*
 - *Building Problem Solvers* – K.D.Forbus and J.D.Kleer
 - *Knowledge Representation and Reasoning*– R. Brachman & H. Levesque
 - *Artificial Intelligence. Third Edition* – Patrick Winston

A (Short) History of AI



ENIAC (1940s)



Early AI: 1940-1950

- 1943- McCulloch and Walter Pitts- Neural Network

Turing Test-1950s

1950: Turing asks the question....



I propose to consider the question:
“Can machines think?”

--Alan Turing, 1950

Early AI: 1950—70

- 1950s: Early AI programs, including Samuel's checkers program, Newell & Simon's Logic Theorist, Gelernter's Geometry Engine
- 1956: Dartmouth meeting: “Artificial Intelligence” adopted as a separate field
 - 4 people: John McCarthy, Marvin Minsky, Claude Shannon and Nathaniel Rochester.
 - The Dartmouth Summer Research Project on Artificial Intelligence was a 1956 summer workshop widely considered to be the founding event of artificial intelligence as a field.
- 1958: LISP by John McCarthy. Became a dominant AI language.

Early AI: 1950—70

- 1964: First Chat bot- Eliza-psychotherapist
- 1965: Robinson's complete algorithm for logical reasoning
- 1966: Shakey- the general purpose mobile robot at Stanford research institute

“Look ma, no hands!” era- John McCarthy

—

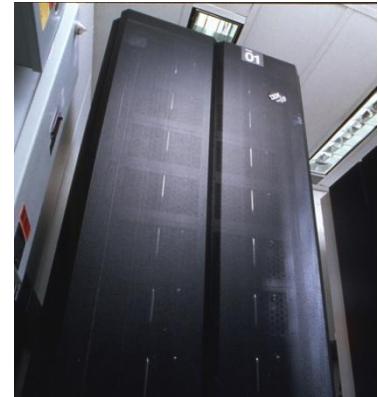
—

Knowledge-based approaches: 1970—90

- 1969—79: Early development of knowledge-based systems
 - Failure of neural network
 - Poor speech understanding
 - Failed machine translation (Russian to English)
 - Caused cancellation of govt. funds
- 1980—88: Expert systems industry booms
 - Decline of LISP
- 1988—93: Expert systems industry busts: “AI Winter”
- 1988- Backpropagation

Statistical approaches + subfield expertise: 1990—2012

- 1996: EQP (an algorithm) proves Robbins algebra are Booleans
- 1997: Deep blue wins chess vs Gary Kasparov
 - “I could feel human level intelligence across room- Gary Kasparov”
 - “Deep Blue hasn't proven anything.”
 - “If it works it is not AI”
 - But now can a human defeat the machine in a series of game??????
- Early 2000s: Resurgence of probability, focus on uncertainty
- 2005: DARPA- Driverless car (first working demonstration)
- Agents and learning systems... “AI Spring”?



Present AI: 2010s to present

- 2011: IBM Watson won the Jeopardy vs Ken Jennings and Brad Rutter
 - “I, for one, welcome our new computer overlords,” - Ken Jennings
- Big data, big compute, neural networks
 - Some re-unification of subfields
 - AI used in many industries



2016:Alpha Go



Lee Se-dol while playing with the AI

Computer Vision



"man in black shirt is playing guitar."



"construction worker in orange safety vest is working on road."



"two young girls are playing with lego toy."



"boy is doing backflip on wakeboard."



"girl in pink dress is jumping in air."



"black and white dog jumps over bar."



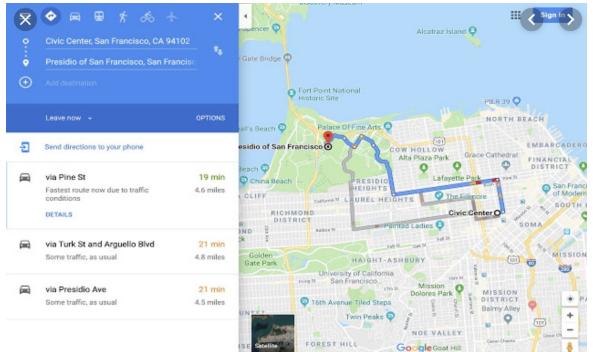
"young girl in pink shirt is swinging on swing."



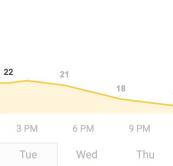
"man in blue wetsuit is surfing on wave."

Karpathy & Fei-Fei, 2015; Donahue et al., 2015; Xu et al, 2015; many more

Tools for Predictions & Decisions

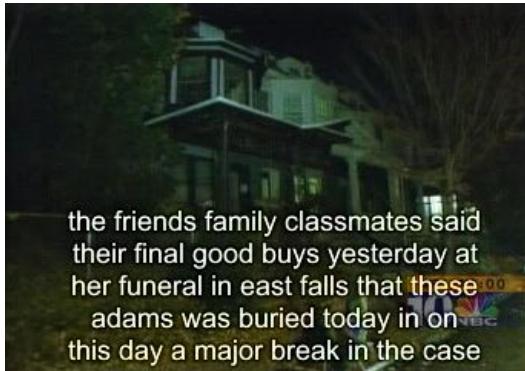


Berkeley, CA 94709
Tuesday 2:00 PM
Mostly Sunny



Natural Language

- Speech technologies (e.g. Siri)
 - Automatic speech recognition (ASR)
 - Text-to-speech synthesis (TTS)
 - Dialog systems
- Language processing technologies
 - Question answering
 - Machine translation



"Il est impossible aux journalistes de rentrer dans les régions tibétaines"

Bruno Philip, correspondant du "Monde" en Chine, estime que les journalistes de l'AFP qui ont été expulsés de la province tibétaine du Qinghai "n'étaient pas dans l'ilégalité".



Les faits Le dalaï-lama dénonce l'"enfer" imposé au Tibet depuis sa fuite, en 1959
Vidéo Anniversaire de la rébellion

"It is impossible for journalists to enter Tibetan areas"

Philip Bruno, correspondent for "World" in China, said that journalists of the AFP who have been deported from the Tibetan province of Qinghai "were not illegal."

Fact The Dalai Lama denounces the "hell" imposed since he fled Tibet in 1959
Video Anniversary of the Tibetan rebellion: China on guard

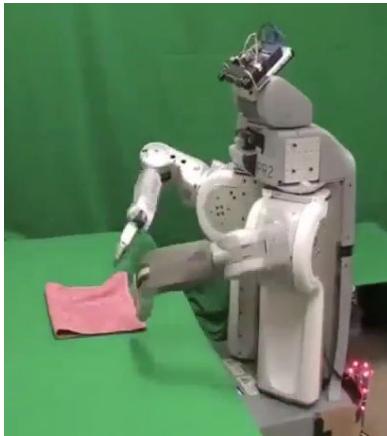


- Web search
- Text classification, spam filtering, etc...

<https://play.aidungeon.io/>

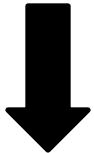
Robotics

- Robotics
 - Part mech. eng.
 - Part AI
 - Reality much harder than simulations!
- Technologies
 - Vehicles
 - Rescue
 - Help in the home
 - Lots of automation...
- In this class:
 - We ignore mechanical aspects
 - Methods for planning
 - Methods for control



Images from UC Berkeley, Boston Dynamics, RoboCup, Google

-
- If it works its not AI

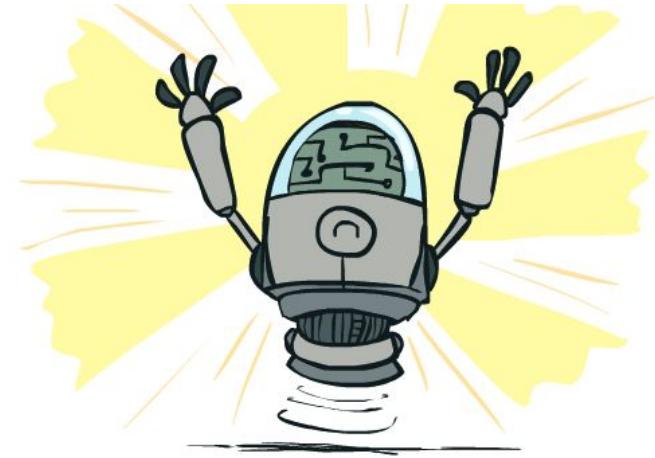


- It's all AI

What Can AI Do?

Quiz: Which of the following can be done at present?

- Play a decent game of Jeopardy?
- Win against any human at chess?
- Win against the best humans at Go?
- Play a decent game of tennis?
- Grab a particular cup and put it on a shelf?
- Unload any dishwasher in any home?
- Drive safely along the highway?
- Drive safely along Telegraph Avenue?
- Buy a week's worth of groceries on the web?
- Buy a week's worth of groceries at Berkeley Bowl?
- Discover and prove a new mathematical theorem?
- Perform a surgical operation?
- Translate spoken Chinese into spoken English in real time?
- Write an intentionally funny story?



How so much increase???

- DATA
- Computational Power
- Algorithms

Artificial Intelligence

- AI: What is the nature of intelligent thought?
- What is intelligence????
 - *Dictionary meaning: capacity for learning, reasoning, understanding and similar for of **mental activity***
- Ability to perceive and act in the world
- Reasoning: Proving theorems, Medical Diagnosis
- Planning: Take decisions
- Learning and Adaptation: Recommend Movies, learn traffic Patterns
- Understanding: Text, speech, visual scene

-
- Are human intelligent??????
 - Are human *always* intelligent???????????
 - Can non-human behavior be intelligent??????

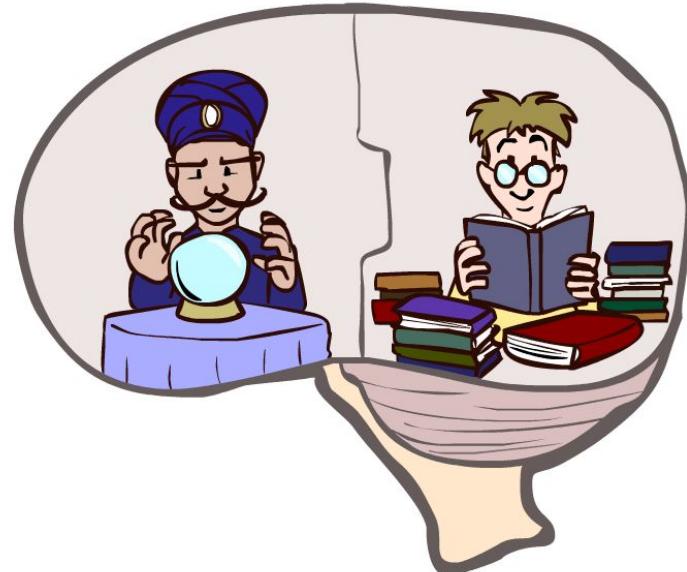
What is AI?

The science of making machines that:

- Acting humanly
 - Turning Test
 - Do you want machine to make human like errors????
- Thinking humanly
 - Cognitive modeling
 - (very hard to understand how human think)
 - Intelligent tutor
 - Elderly healthcare bot
- Thinking Rationally
 - Laws of thought (Logic and reasoning, inferences and conclusion)
 - Purposeful thinking?
- Acting Rationally
 - Rational behavior: Doing right thing
 - What is rationality??

What About the Brain?

- Brains (human minds) are very good at making rational decisions, but not perfect
- Brains aren't as modular as software, so hard to reverse engineer!
- "Brains are to intelligence as wings are to flight"
- Lessons learned from the brain: memory (data) and simulation (computation) are key to decision making



Rational Agents

- An agents should strive **to do right thing**, based on what it can perceive and the actions it can perform.
- The right action is the one that will cause the agent to be the most successful
- Performance measure: An objective criterion for success of an agent's behavior
- For example: performance measure of a vacuum cleaner agent could be the amount of dirt cleaned up, time taken, electricity consumed, etc.

Ideal Rational Agents

- “For each possible percept sequence, does whatever action is **expected to maximize its performance measure** on the basis of **evidence perceived** so far and **built in knowledge**”

- RATIONALITY vs OMNISCIENCE???
- Acting in order to obtain information

Rational Decisions

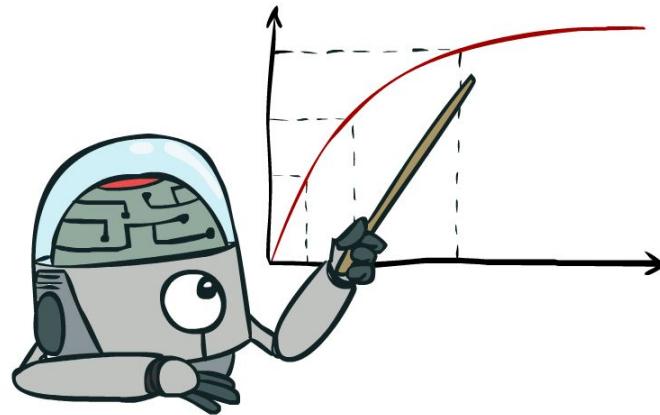
We'll use the term **rational** in a very specific, technical way:

- Rational: maximally achieving pre-defined goals
- Rationality only concerns what decisions are made
(not the thought process behind them)
- Goals are expressed in terms of the **utility** of outcomes
- Being rational means **maximizing your expected utility**

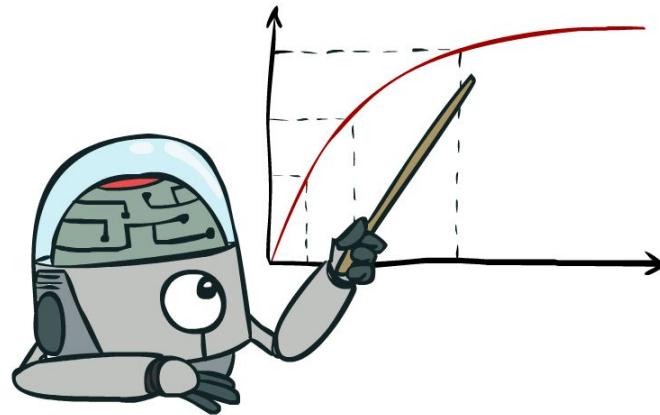
A better title for this course would be:

Computational Rationality

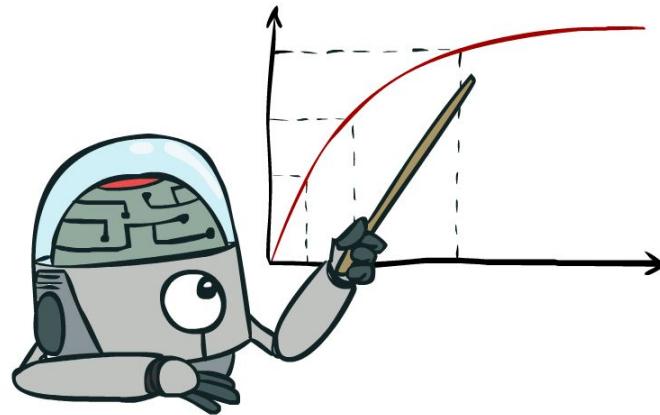
Maximize Your Expected Utility



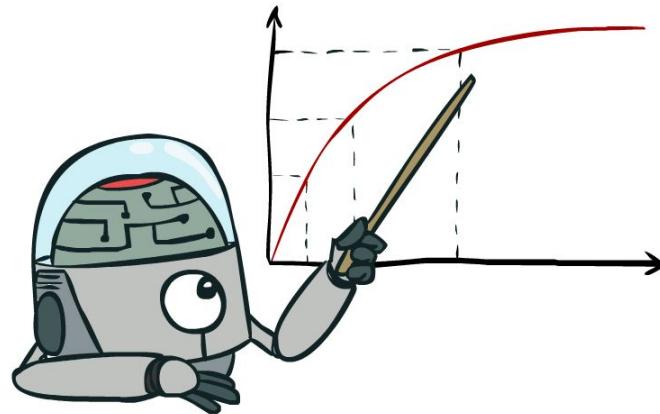
Maximize Your Expected Utility



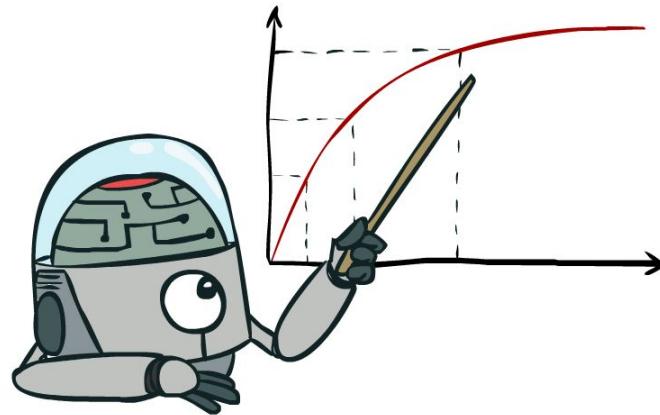
Maximize Your Expected Utility



Maximize Your Expected Utility

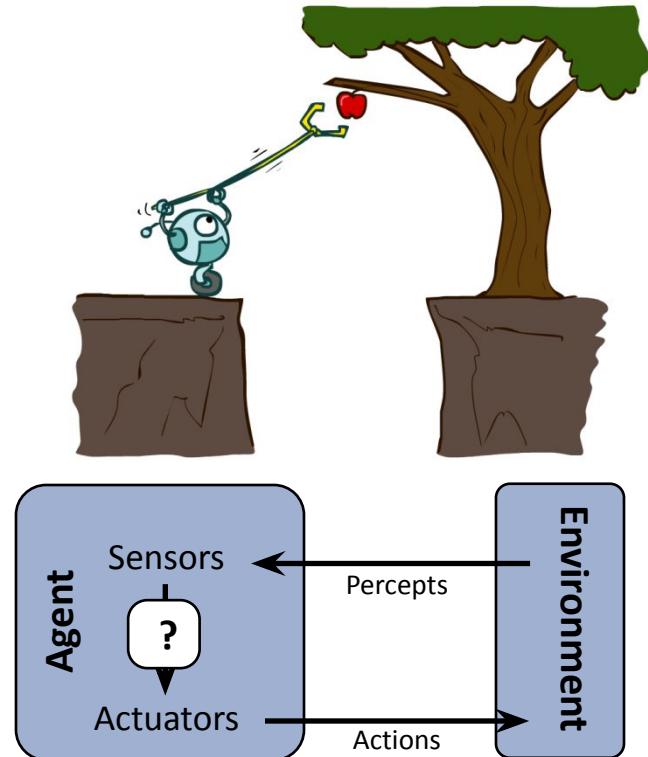


Maximize Your Expected Utility



Designing Rational Agents

- An **agent** is an entity that *perceives* and *acts*.
- A **rational agent** selects actions that maximize its (expected) **utility**.
- Characteristics of the **percepts**, **environment**, and **action space** dictate techniques for selecting rational actions
- **This course is about:**
 - General AI techniques for a variety of problem types
 - Learning to recognize when and how a new problem can be solved with an existing technique



INTELLIGENT AGENTS

CHAPTER 2

Reminders

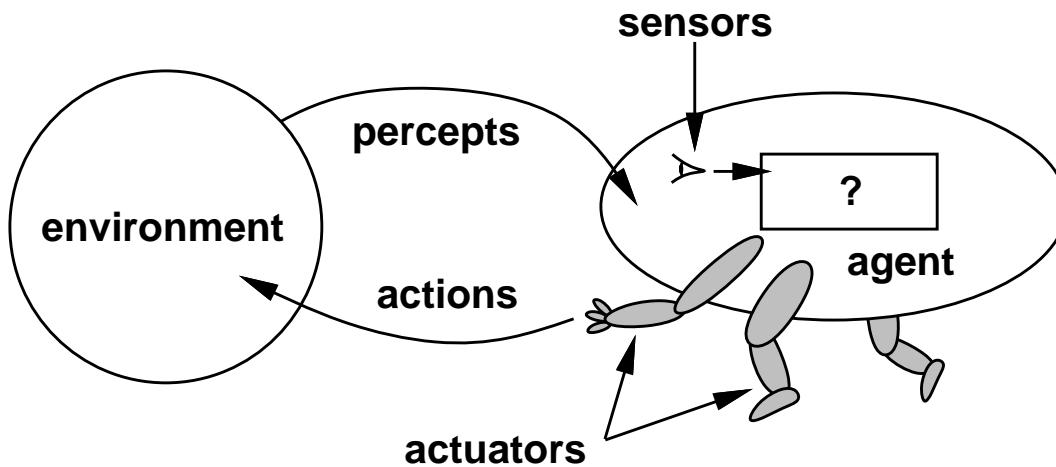
Assignment 0 (lisp refresher) due 1/28

Lisp/emacs/AIMA tutorial: 11-1 today and Monday, 271 Soda

Outline

- ◊ Agents and environments
- ◊ Rationality
- ◊ PEAS (Performance measure, Environment, Actuators, Sensors)
- ◊ Environment types
- ◊ Agent types

Agents and environments



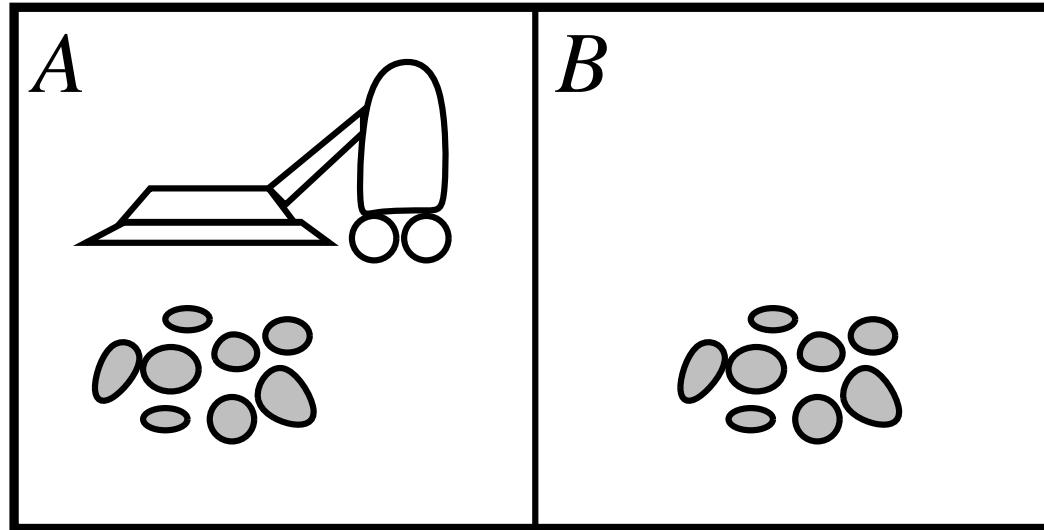
Agents include humans, robots, softbots, thermostats, etc.

The agent function maps from percept histories to actions:

$$f : \mathcal{P}^* \rightarrow \mathcal{A}$$

The agent program runs on the physical architecture to produce f

Vacuum-cleaner world



Percepts: location and contents, e.g., [*A*, *Dirty*]

Actions: *Left*, *Right*, *Suck*, *NoOp*

A vacuum-cleaner agent

Percept sequence	Action
$[A, Clean]$	<i>Right</i>
$[A, Dirty]$	<i>Suck</i>
$[B, Clean]$	<i>Left</i>
$[B, Dirty]$	<i>Suck</i>
$[A, Clean], [A, Clean]$	<i>Right</i>
$[A, Clean], [A, Dirty]$	<i>Suck</i>
:	:

```
function REFLEX-VACUUM-AGENT( [location,status] ) returns an action
    if status = Dirty then return Suck
    else if location = A then return Right
    else if location = B then return Left
```

What is the **right** function?
Can it be implemented in a small agent program?

Rationality

Fixed **performance measure** evaluates the **environment sequence**

- one point per square cleaned up in time T ?
- one point per clean square per time step, minus one per move?
- penalize for $> k$ dirty squares?

A **rational agent** chooses whichever action maximizes the **expected value** of the performance measure **given the percept sequence to date**

Rational \neq omniscient

- percepts may not supply all relevant information

Rational \neq clairvoyant

- action outcomes may not be as expected

Hence, rational \neq successful

Rational \Rightarrow exploration, learning, autonomy

PEAS

To design a rational agent, we must specify the **task environment**

Consider, e.g., the task of designing an automated taxi:

Performance measure??

Environment??

Actuators??

Sensors??

PEAS

To design a rational agent, we must specify the **task environment**

Consider, e.g., the task of designing an automated taxi:

Performance measure?? safety, destination, profits, legality, comfort, . . .

Environment?? US streets/freeways, traffic, pedestrians, weather, . . .

Actuators?? steering, accelerator, brake, horn, speaker/display, . . .

Sensors?? video, accelerometers, gauges, engine sensors, keyboard, GPS, . . .

Internet shopping agent

Performance measure??

Environment??

Actuators??

Sensors??

Internet shopping agent

Performance measure?? price, quality, appropriateness, efficiency

Environment?? current and future WWW sites, vendors, shippers

Actuators?? display to user, follow URL, fill in form

Sensors?? HTML pages (text, graphics, scripts)

Environment types

	Solitaire	Backgammon	Internet shopping	Taxi
<u>Observable??</u> <u>Deterministic??</u> <u>Episodic??</u> <u>Static??</u> <u>Discrete??</u> <u>Single-agent??</u>				

Environment types

	Solitaire	Backgammon	Internet shopping	Taxi
<u>Observable??</u>	Yes	Yes	No	No
<u>Deterministic??</u>				
<u>Episodic??</u>				
<u>Static??</u>				
<u>Discrete??</u>				
<u>Single-agent??</u>				

Environment types

	Solitaire	Backgammon	Internet shopping	Taxi
<u>Observable??</u>	Yes	Yes	No	No
<u>Deterministic??</u>	Yes	No	Partly	No
<u>Episodic??</u>				
<u>Static??</u>				
<u>Discrete??</u>				
<u>Single-agent??</u>				

Environment types

	Solitaire	Backgammon	Internet shopping	Taxi
<u>Observable??</u>	Yes	Yes	No	No
<u>Deterministic??</u>	Yes	No	Partly	No
<u>Episodic??</u>	No	No	No	No
<u>Static??</u>				
<u>Discrete??</u>				
<u>Single-agent??</u>				

Environment types

	Solitaire	Backgammon	Internet shopping	Taxi
<u>Observable??</u>	Yes	Yes	No	No
<u>Deterministic??</u>	Yes	No	Partly	No
<u>Episodic??</u>	No	No	No	No
<u>Static??</u>	Yes	Semi	Semi	No
<u>Discrete??</u>				
<u>Single-agent??</u>				

Environment types

	Solitaire	Backgammon	Internet shopping	Taxi
<u>Observable??</u>	Yes	Yes	No	No
<u>Deterministic??</u>	Yes	No	Partly	No
<u>Episodic??</u>	No	No	No	No
<u>Static??</u>	Yes	Semi	Semi	No
<u>Discrete??</u>	Yes	Yes	Yes	No
<u>Single-agent??</u>				

Environment types

	Solitaire	Backgammon	Internet shopping	Taxi
<u>Observable??</u>	Yes	Yes	No	No
<u>Deterministic??</u>	Yes	No	Partly	No
<u>Episodic??</u>	No	No	No	No
<u>Static??</u>	Yes	Semi	Semi	No
<u>Discrete??</u>	Yes	Yes	Yes	No
<u>Single-agent??</u>	Yes	No	Yes (except auctions)	No

The environment type largely determines the agent design

The real world is (of course) partially observable, stochastic, sequential, dynamic, continuous, multi-agent

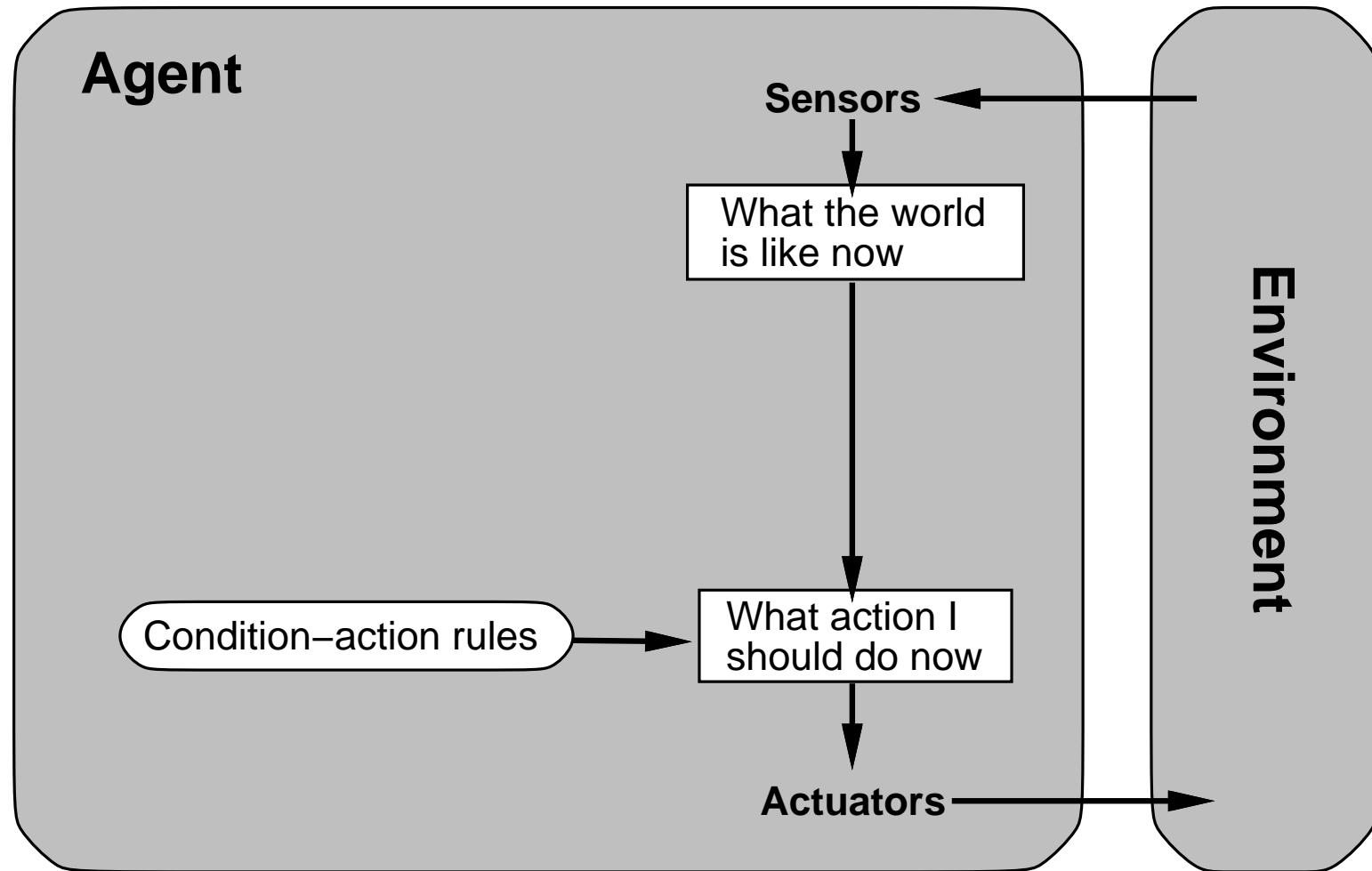
Agent types

Four basic types in order of increasing generality:

- simple reflex agents
- reflex agents with state
- goal-based agents
- utility-based agents

All these can be turned into learning agents

Simple reflex agents



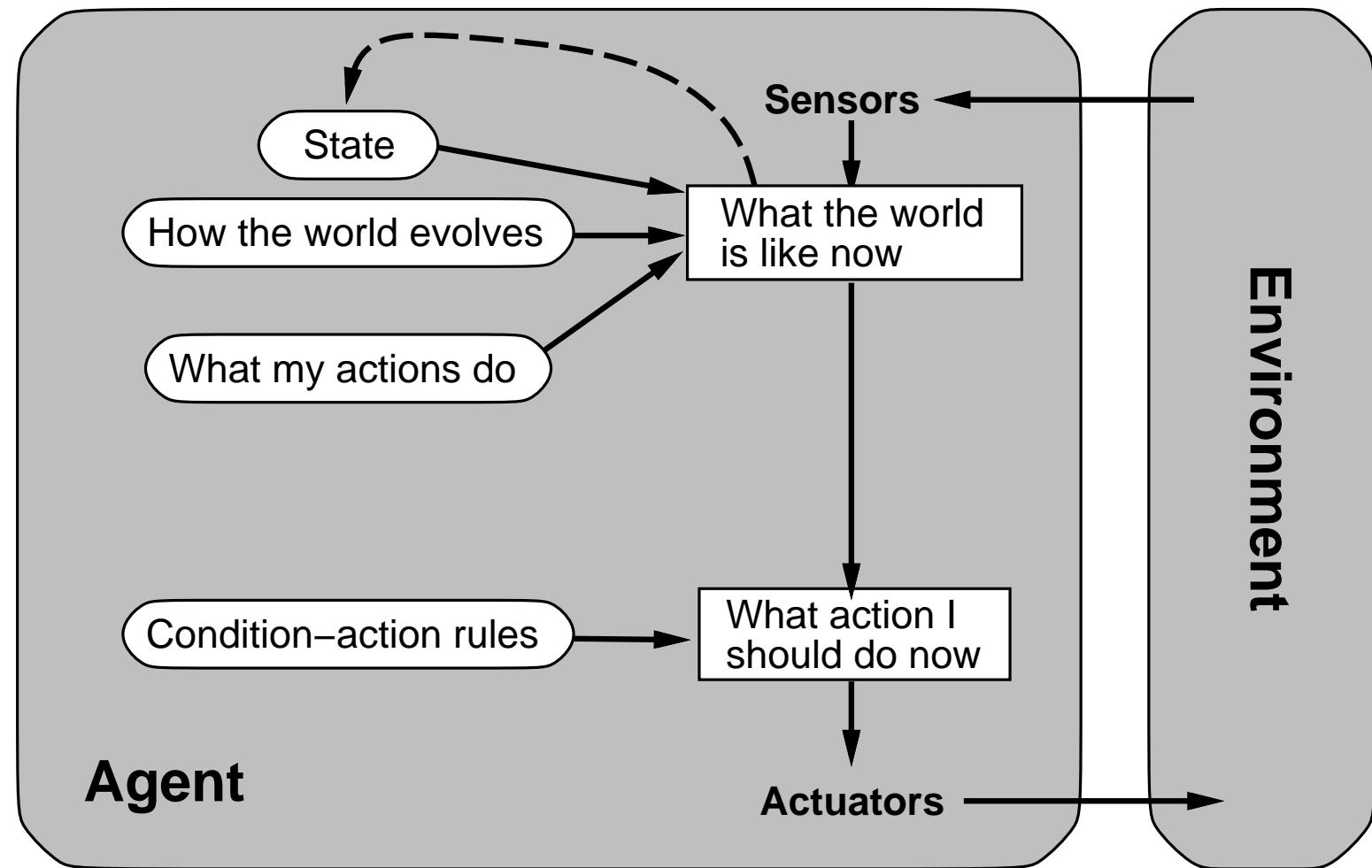
Example

```
function REFLEX-VACUUM-AGENT( [location,status]) returns an action
  if status = Dirty then return Suck
  else if location = A then return Right
  else if location = B then return Left
```

```
(setq joe (make-agent :name 'joe :body (make-agent-body)
                      :program (make-reflex-vacuum-agent-program))

(defun make-reflex-vacuum-agent-program ()
  #'(lambda (percept)
      (let ((location (first percept)) (status (second percept)))
        (cond ((eq status 'dirty) 'Suck)
              ((eq location 'A) 'Right)
              ((eq location 'B) 'Left))))
```

Reflex agents with state

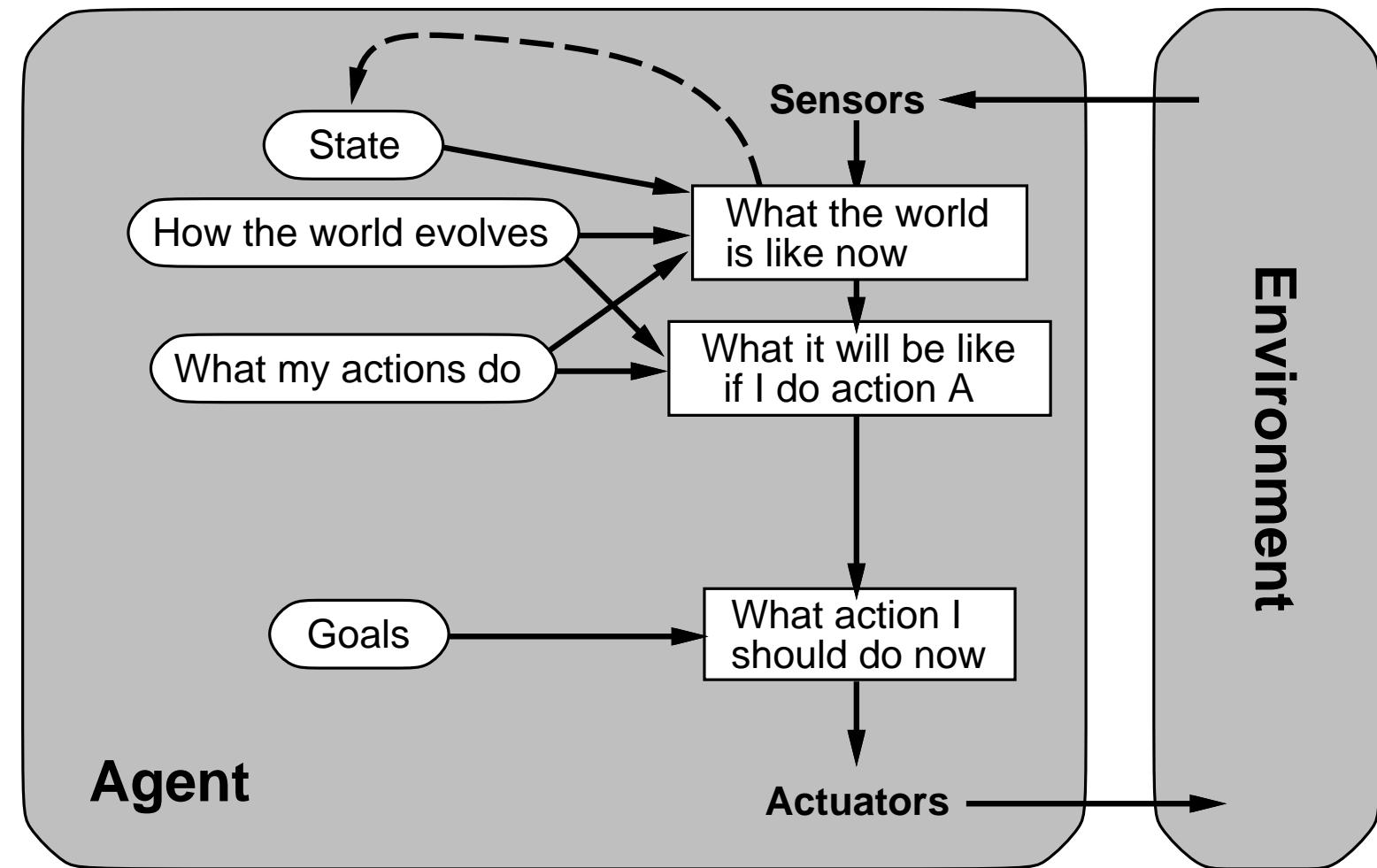


Example

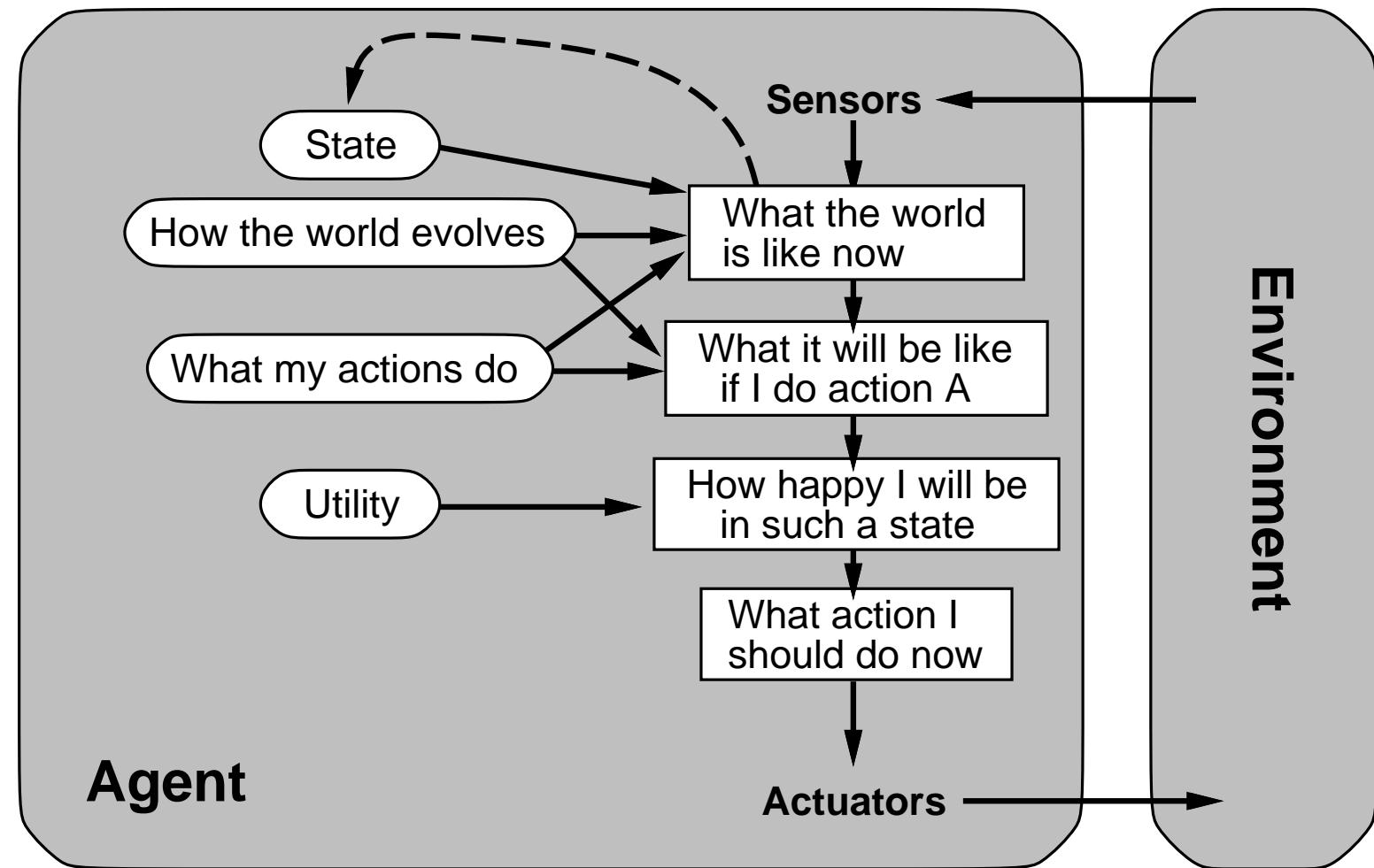
```
function REFLEX-VACUUM-AGENT( [location,status]) returns an action
static: last_A, last_B, numbers, initially ∞
    if status = Dirty then ...
```

```
(defun make-reflex-vacuum-agent-with-state-program ()
  (let ((last-A infinity) (last-B infinity))
    #'(lambda (percept)
        (let ((location (first percept)) (status (second percept)))
          (incf last-A) (incf last-B)
          (cond
            ((eq status 'dirty)
             (if (eq location 'A) (setq last-A 0) (setq last-B 0))
             'Suck)
            ((eq location 'A) (if (> last-B 3) 'Right 'NoOp))
            ((eq location 'B) (if (> last-A 3) 'Left 'NoOp)))))))
```

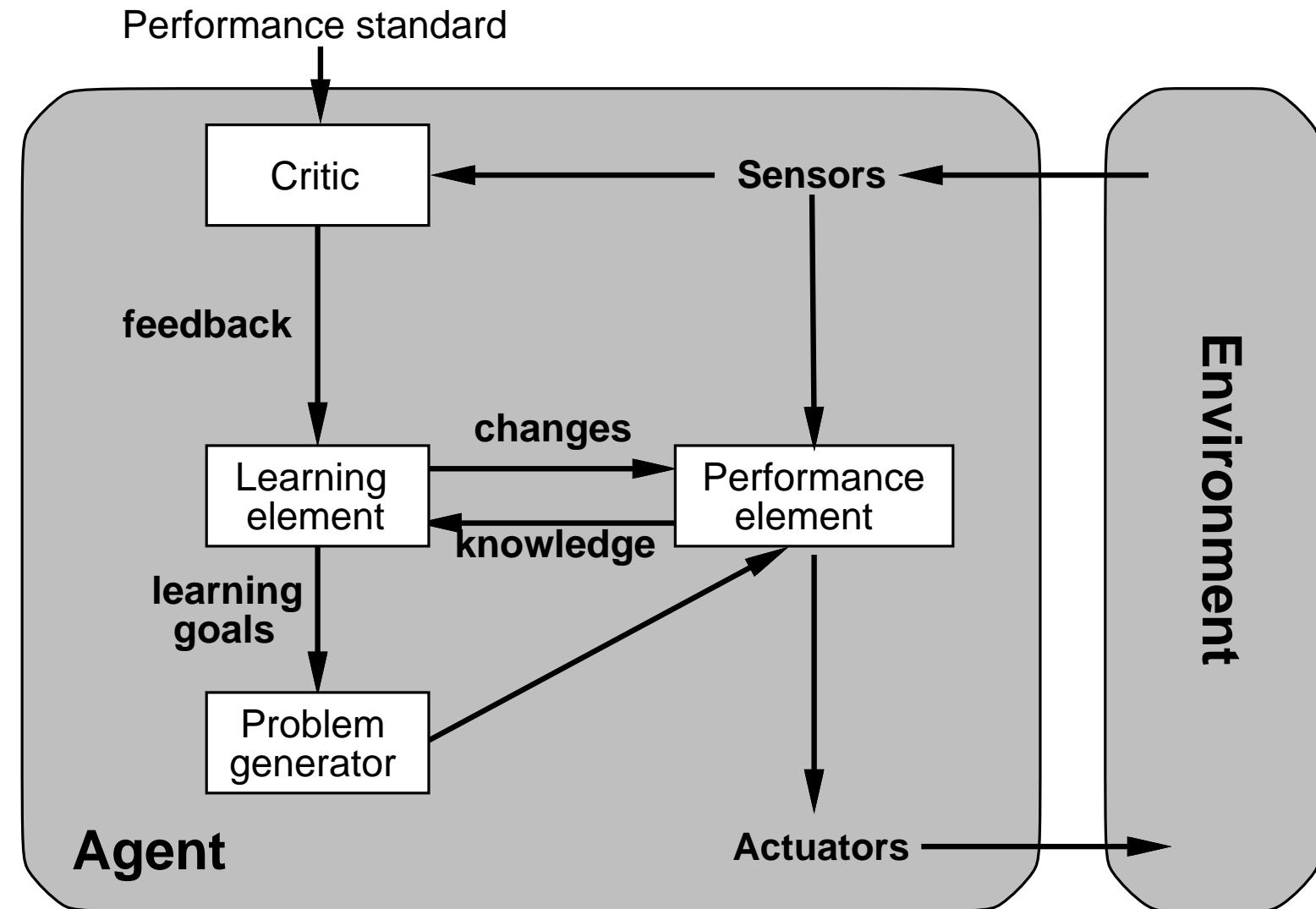
Goal-based agents



Utility-based agents



Learning agents



Summary

Agents interact with environments through **actuators** and **sensors**

The agent function describes what the agent does in all circumstances

The performance measure evaluates the environment sequence

A perfectly rational agent maximizes expected performance

Agent programs implement (some) agent functions

PEAS descriptions define task environments

Environments are categorized along several dimensions:

observable? deterministic? episodic? static? discrete? single-agent?

Several basic agent architectures exist:

reflex, reflex with state, goal-based, utility-based

Problem solving and search

AI

Problem Solving Agents

- When the correct action to take is not immediately obvious, an agent may need to plan ahead to consider a **sequence of actions** that form a ***path to a goal state***.
- Such agents is called a problem-solving agent, and the computational process it undertakes is called **search**.
- Problem Solving Phases (four-phase problem-solving process):
 - **Goal formulation:** Goals organize behaviour by limiting the objectives and hence the actions to be considered.
 - **Problem formulation:** The agent devises a description of the states and actions necessary to reach the goal.
 - **Search:** Before taking any action in the real world, the agent simulates sequences of actions in its model, searching until it finds a sequence of actions that reaches the goal.
 - Such a sequence is called a solution.
 - **Execution:** The agent can now execute the actions in the solution, one at a time.

Example: Romania

On holiday in Romania; currently in Arad.
Flight leaves tomorrow from Bucharest

Formulate goal:

be in Bucharest

Formulate problem:

states: various cities

actions: drive between cities

Find solution:

sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest

- Credits: AIMA by Peter N.

Example: Romania

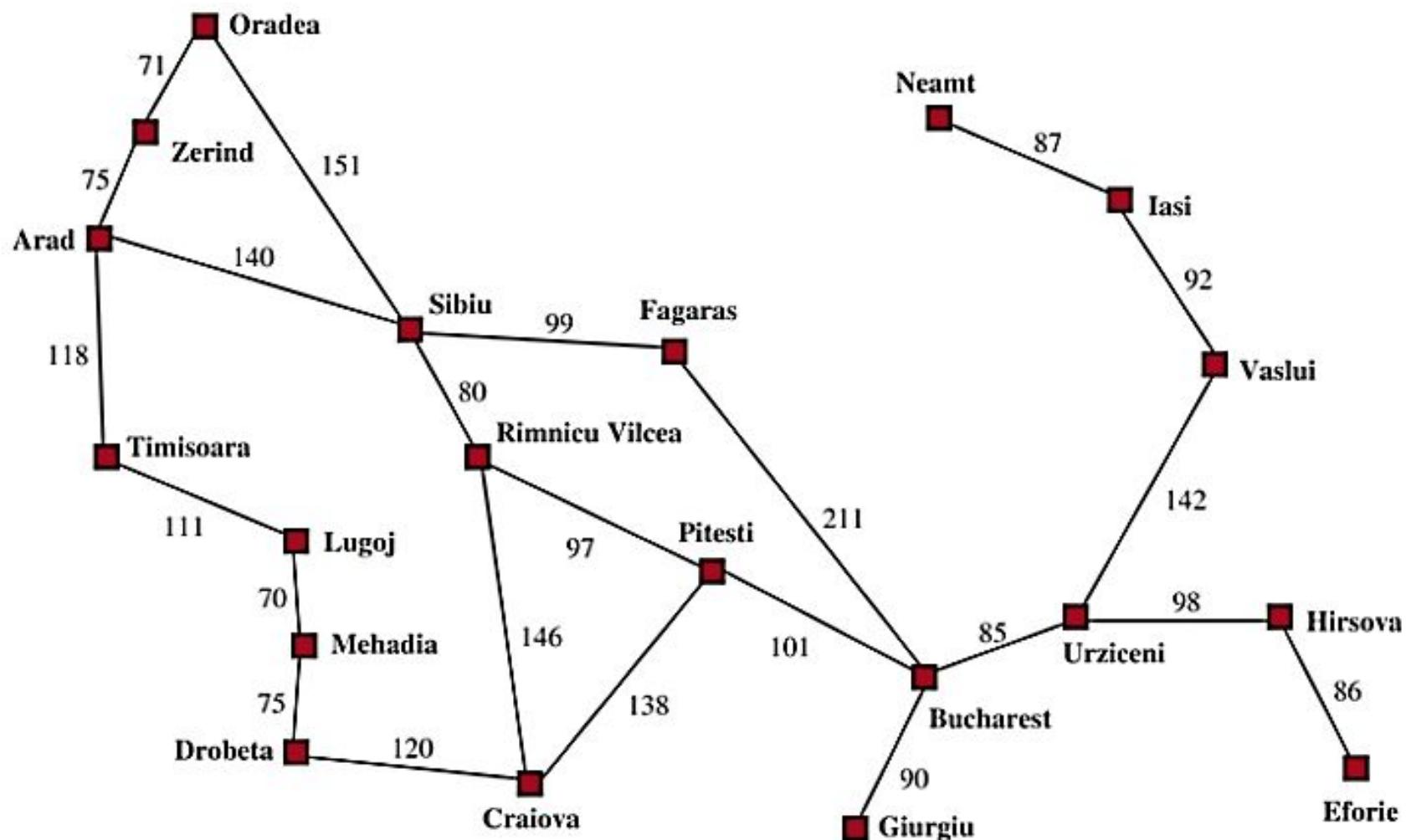


Figure 3.1 A simplified road map of part of Romania, with road distances in miles.

- Credits: AIMA by Peter N.

Search problems and solutions

- A search problem can be defined formally as follows:
 - A set of possible **states** that the environment can be in. We call this the **state space**.
 - The **initial state** that the agent starts in. For example: Arad.
 - A set of **one or more goal states**.
 - The **actions** available to the agent.
 - » ACTIONS (Arad) = {ToSibiu, ToTimisoara, ToZerind }
 - A **transition model**, which describes what each action does.
 - An **action cost function**, denoted by ACTION-COST(s, a, s'').

Search problems and solutions

- A sequence of actions forms a **path**, and a **solution** is a path from the initial state to a goal state.
- An **optimal solution** has the lowest path cost among all solutions.
- The state space can be represented as a **graph** in which the vertices are states and the directed edges between them are actions.

Formulating problems

- Our formulation of the problem of getting to goal is a **model**—an abstract mathematical description.
- The process of removing detail from a representation is called **abstraction**.
- A good problem formulation has the right level of detail.

*“The choice of a **good abstraction** involves **removing as much detail as possible** while **retaining validity** and **ensuring that the abstract actions are easy to carry out.**”*

Problem types

Deterministic, fully observable \Rightarrow single-state problem

Agent knows exactly which state it will be in; solution is a sequence

Non-observable \Rightarrow conformant problem

Agent may have no idea where it is; solution (if any) is a sequence

Nondeterministic and/or partially observable \Rightarrow contingency problem

percepts provide new information about current state

solution is a contingent plan or a policy

often interleave search, execution

Unknown state space \Rightarrow exploration problem ("online")

Example: vacuum world

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

[*Right, Suck*]

Conformant, start in {1, 2, 3, 4, 5, 6, 7, 8}

e.g., *Right* goes to {2, 4, 6, 8}. Solution??

[*Right, Suck, Left, Suck*]

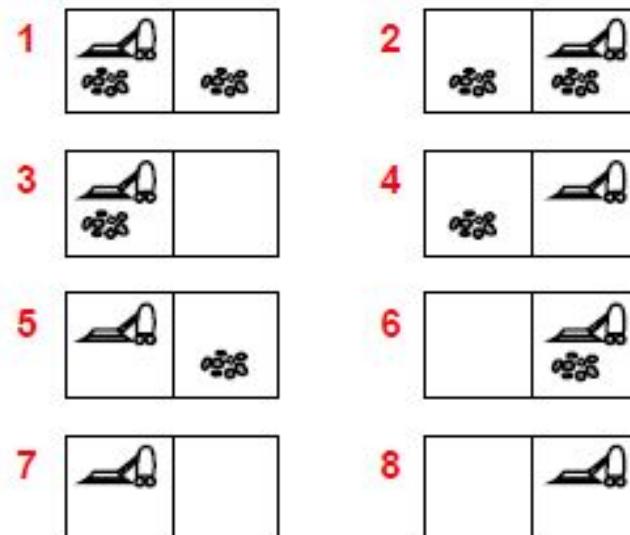
Contingency, start in #5

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

Local sensing: dirt, location only.

Solution??

[*Right, if dirt then Suck*]



Single-state problem formulation

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

initial state e.g., "at Arad"

successor function $S(x)$ = set of action-state pairs

e.g., $S(Arad) = \{(Arad \rightarrow Zerind, Zerind), \dots\}$

goal test, can be

explicit, e.g., x = "at Bucharest"

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

path cost (additive)

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

$c(x, a, y)$ is the step cost, assumed to be ≥ 0

A **solution** is a sequence of actions

leading from the initial state to a goal state

Selecting a state space

Real world is absurdly complex

⇒ state space must be **abstracted** for problem solving

(Abstract) state = set of real states

(Abstract) action = complex combination of real actions

e.g., "Arad → Zerind" represents a complex set
of possible routes, detours, rest stops, etc.

For guaranteed realizability, **any** real state "in Arad"

must get to **some** real state "in Zerind"

(Abstract) solution =

set of real paths that are solutions in the real world

Each abstract action should be "easier" than the original problem!

Example: Vacuum world state-space graph

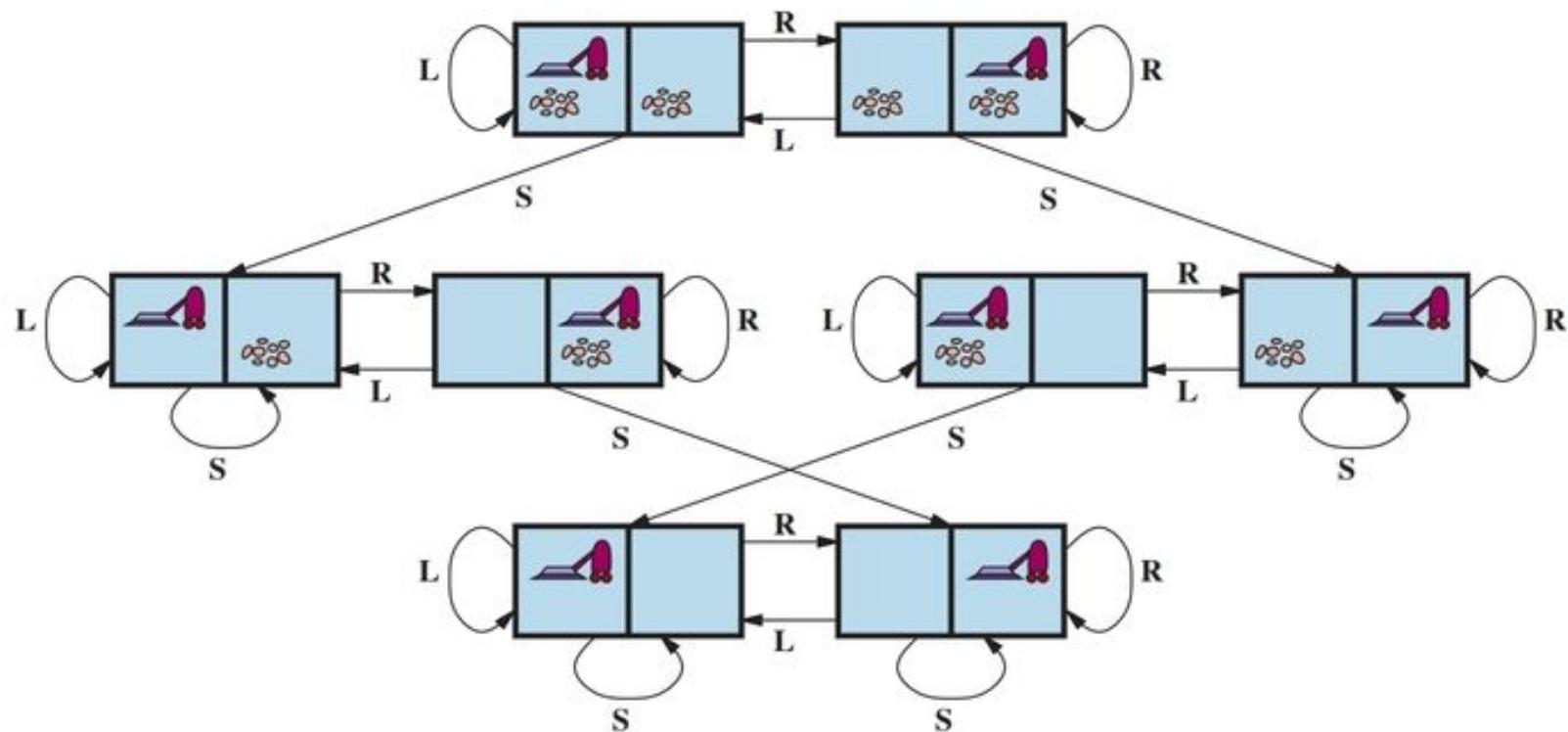
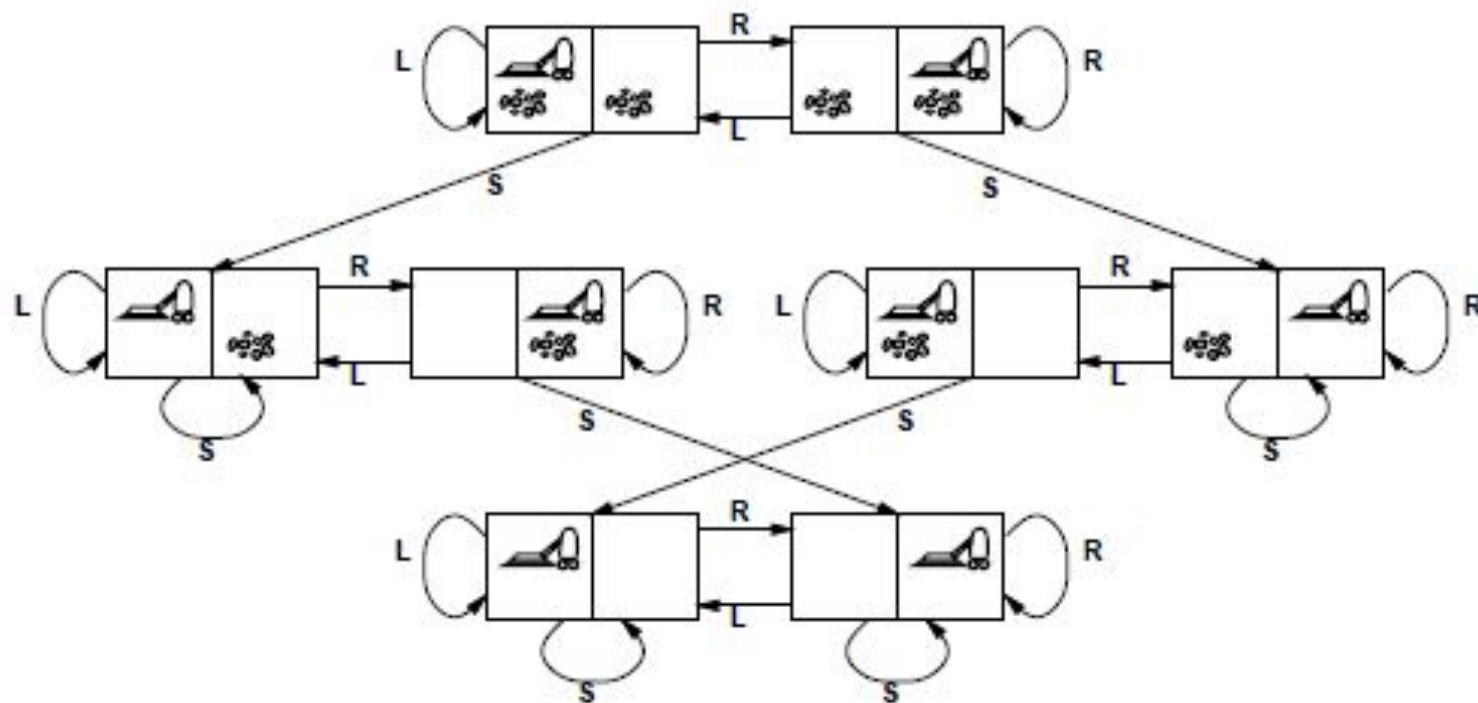


Figure 3.2 The state-space graph for the two-cell vacuum world. There are 8 states and three actions for each state: L = Left, R = Right, S = Suck.

Example: vacuum world state space graph



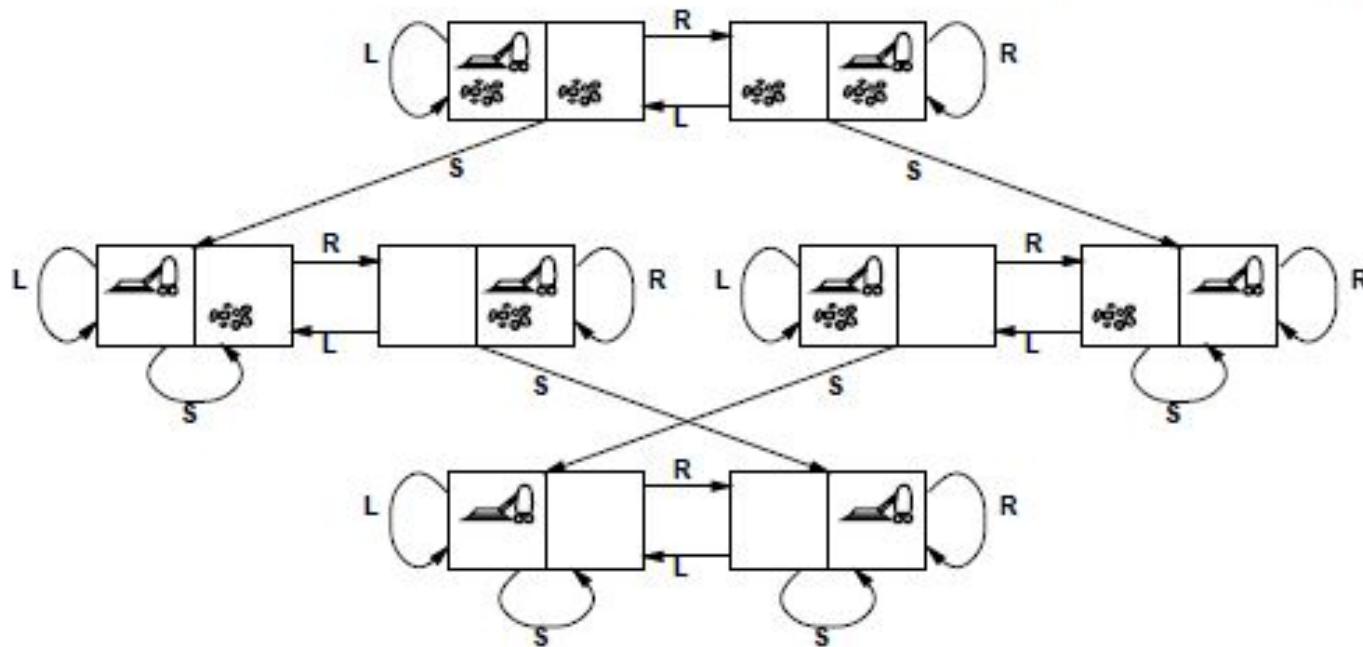
states??

actions??

goal test??

path cost??

Example: vacuum world state space graph



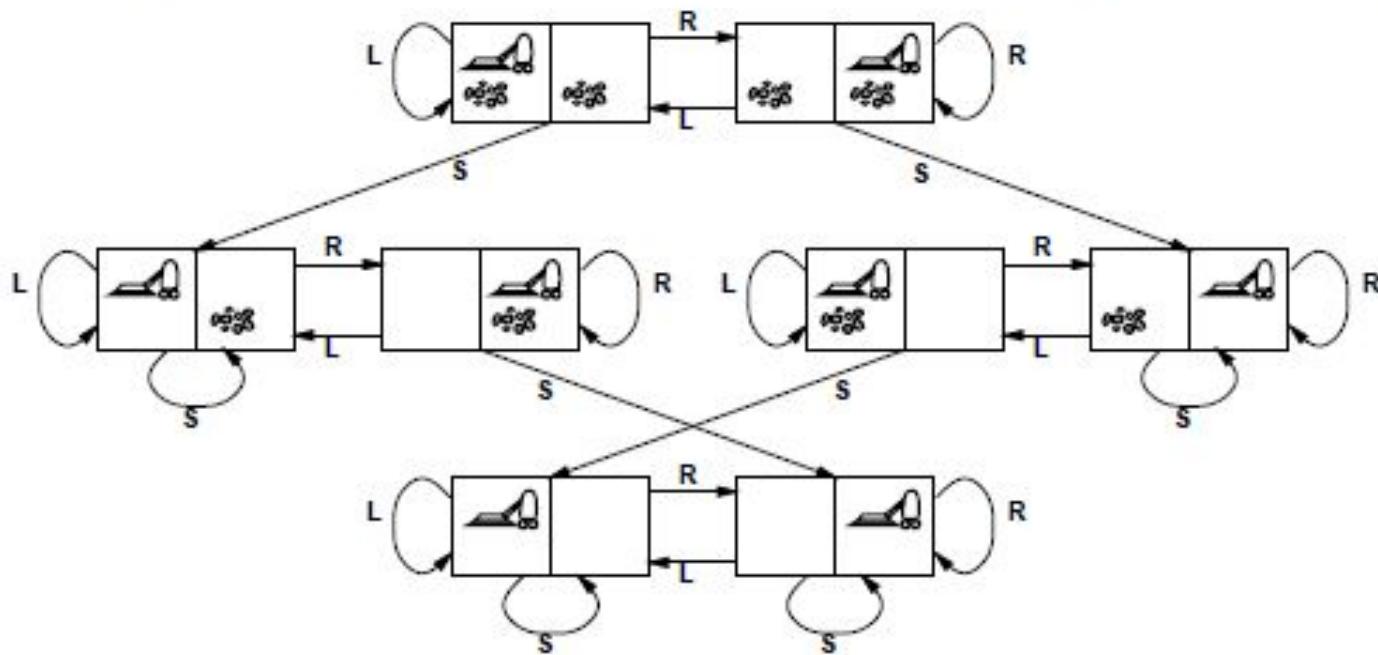
states??: integer dirt and robot locations (ignore dirt amounts etc.)

actions??

goal test??

path cost??

Example: vacuum world state space graph



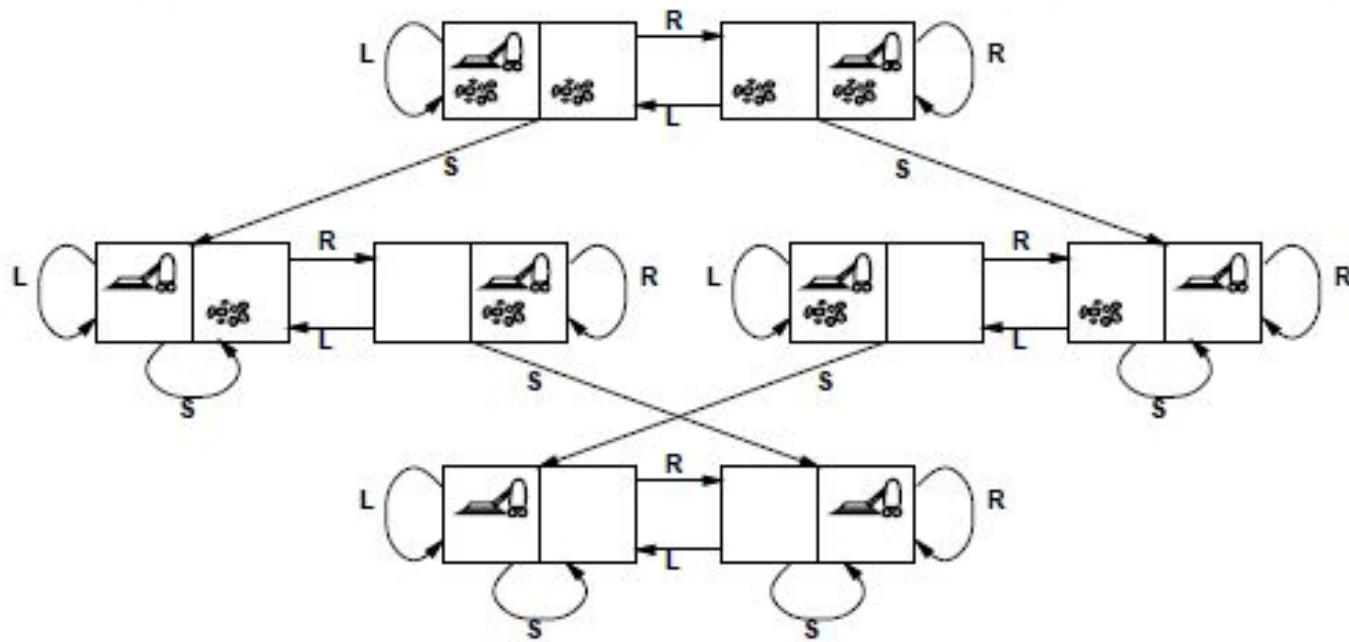
states??: integer dirt and robot locations (ignore dirt amounts etc.)

actions??: *Left, Right, Suck, NoOp*

goal test??

path cost??

Example: vacuum world state space graph



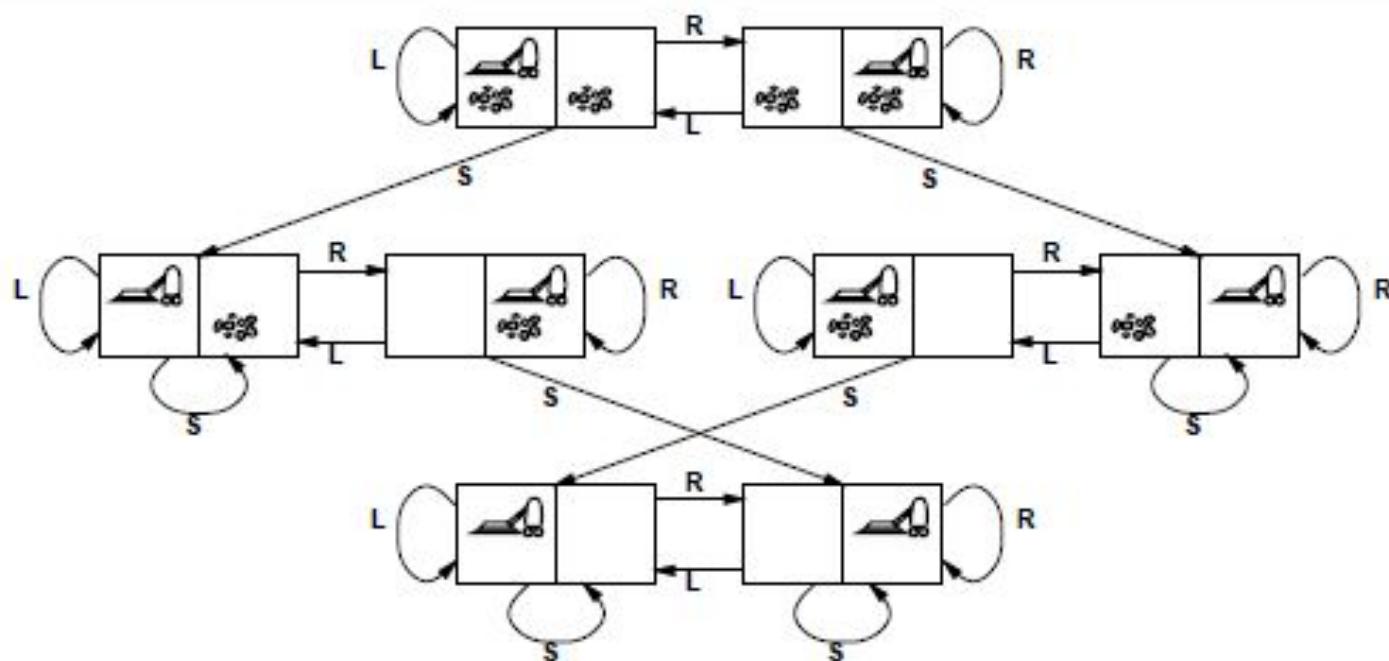
states??: integer dirt and robot locations (ignore dirt amounts etc.)

actions??: *Left*, *Right*, *Suck*, *NoOp*

goal test??: no dirt

path cost??

Example: vacuum world state space graph



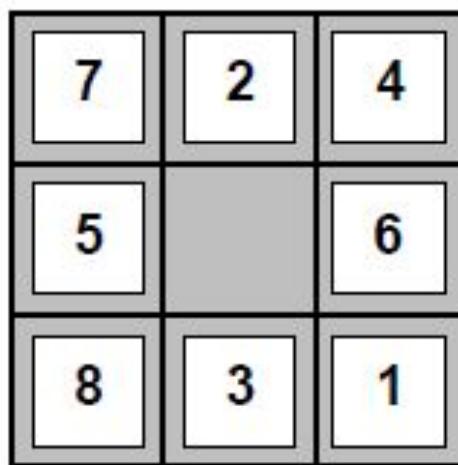
states??: integer dirt and robot locations (ignore dirt amounts etc.)

actions??: *Left*, *Right*, *Suck*, *NoOp*

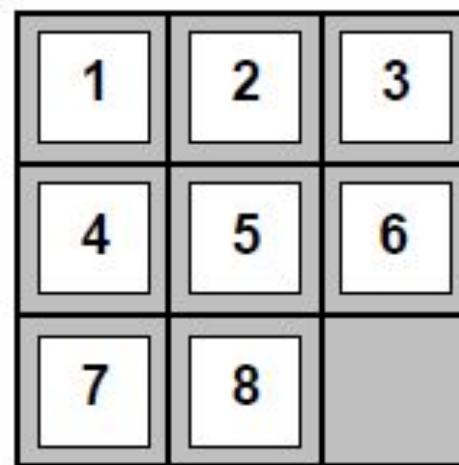
goal test??: no dirt

path cost??: 1 per action (0 for *NoOp*)

Example: The 8-puzzle



Start State



Goal State

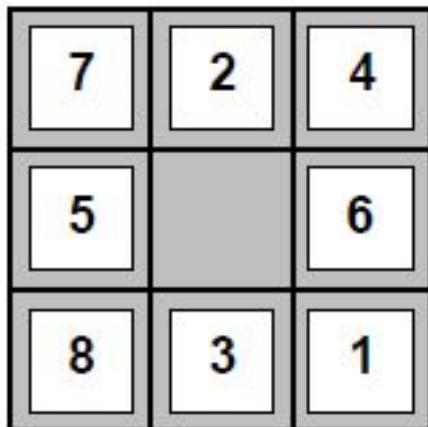
states??

actions??

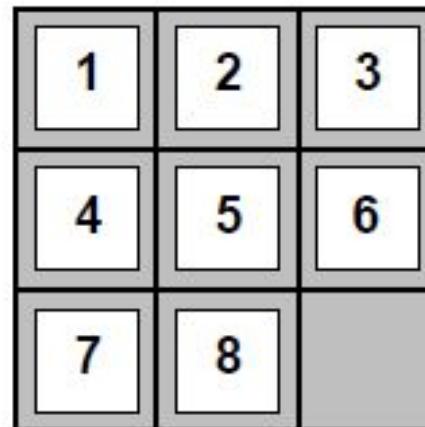
goal test??

path cost??

Example: The 8-puzzle



Start State



Goal State

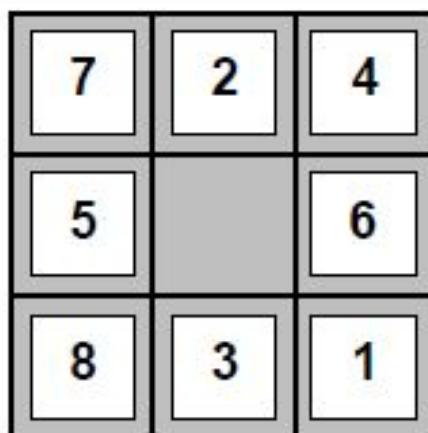
states??: integer locations of tiles (ignore intermediate positions)

actions??

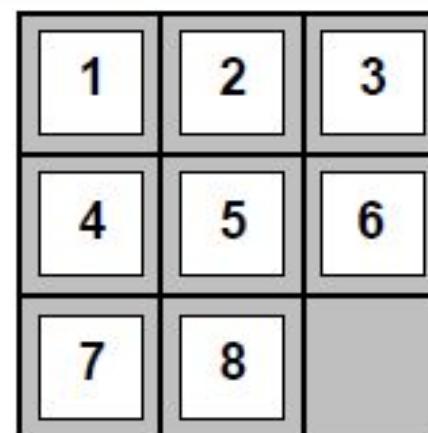
goal test??

path cost??

Example: The 8-puzzle



Start State



Goal State

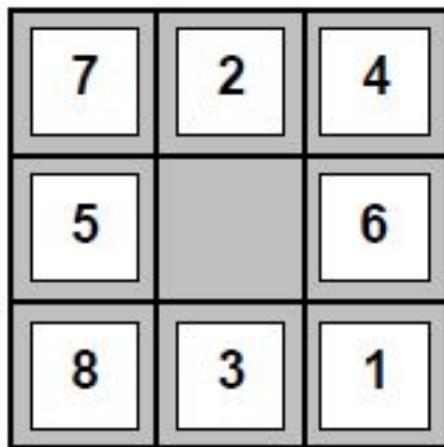
states??: integer locations of tiles (ignore intermediate positions)

actions??: move blank left, right, up, down (ignore unjamming etc.)

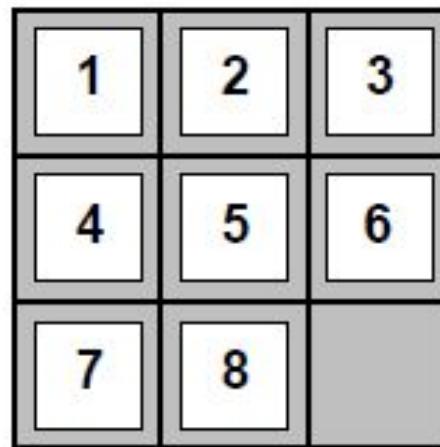
goal test??

path cost??

Example: The 8-puzzle



Start State



Goal State

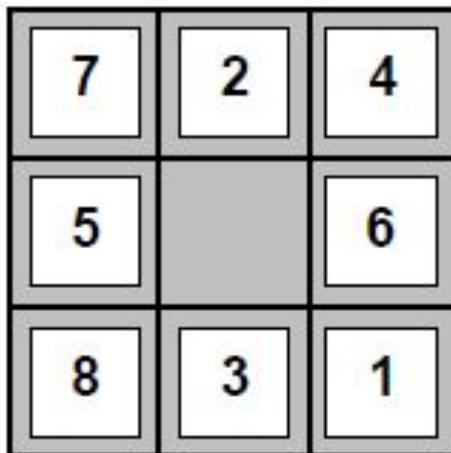
states??: integer locations of tiles (ignore intermediate positions)

actions??: move blank left, right, up, down (ignore unjamming etc.)

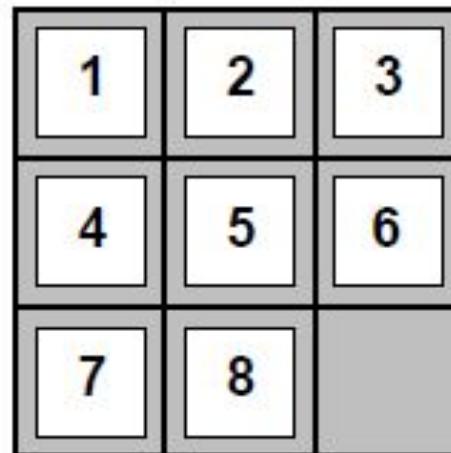
goal test??: = goal state (given)

path cost??

Example: The 8-puzzle



Start State



Goal State

states??: integer locations of tiles (ignore intermediate positions)

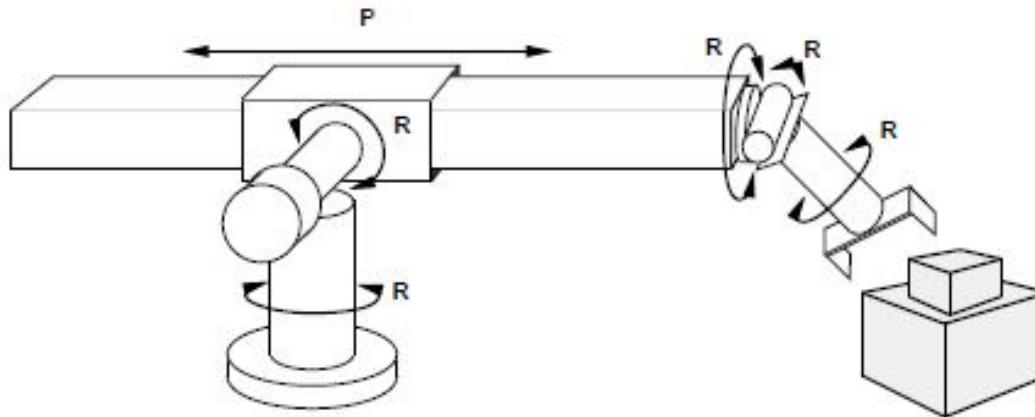
actions??: move blank left, right, up, down (ignore unjamming etc.)

goal test??: = goal state (given)

path cost??: 1 per move

[Note: optimal solution of n -Puzzle family is NP-hard]

Example: robotic assembly



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

actions??: continuous motions of robot joints

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

path cost??: time to execute

PROBLEM SOLVING AND SEARCH

CHAPTER 3

Reminders

Assignment 0 due 5pm today

Assignment 1 posted, due 2/9

Section 105 will move to 9-10am starting next week

Outline

- ◊ Problem-solving agents
- ◊ Problem types
- ◊ Problem formulation
- ◊ Example problems
- ◊ Basic search algorithms

Problem-solving agents

Restricted form of general agent:

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

  state  $\leftarrow$  UPDATE-STATE(state, percept)
  if seq is empty then
    goal  $\leftarrow$  FORMULATE-GOAL(state)
    problem  $\leftarrow$  FORMULATE-PROBLEM(state, goal)
    seq  $\leftarrow$  SEARCH(problem)
  action  $\leftarrow$  RECOMMENDATION(seq, state)
  seq  $\leftarrow$  REMAINDER(seq, state)
  return action
```

Note: this is **offline** problem solving; solution executed “eyes closed.”

Online problem solving involves acting without complete knowledge.

Example: Romania

On holiday in Romania; currently in Arad.

Flight leaves tomorrow from Bucharest

Formulate goal:

be in Bucharest

Formulate problem:

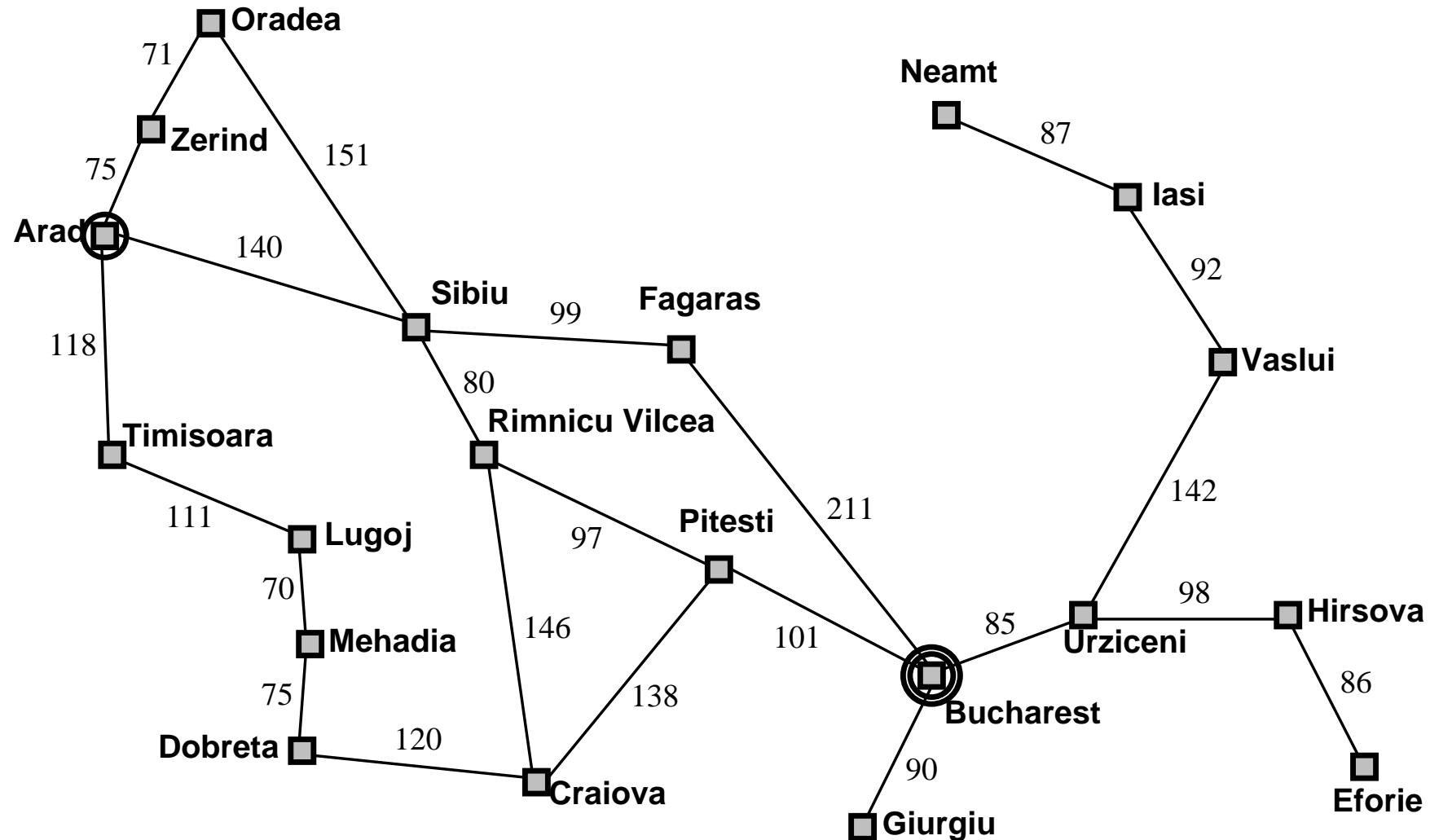
states: various cities

actions: drive between cities

Find solution:

sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest

Example: Romania



Problem types

Deterministic, fully observable \Rightarrow single-state problem

Agent knows exactly which state it will be in; solution is a sequence

Non-observable \Rightarrow conformant problem

Agent may have no idea where it is; solution (if any) is a sequence

Nondeterministic and/or partially observable \Rightarrow contingency problem

percepts provide new information about current state

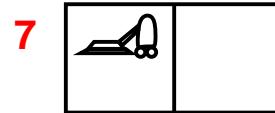
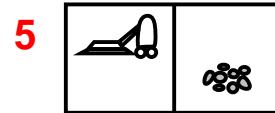
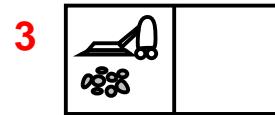
solution is a contingent plan or a policy

often interleave search, execution

Unknown state space \Rightarrow exploration problem (“online”)

Example: vacuum world

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



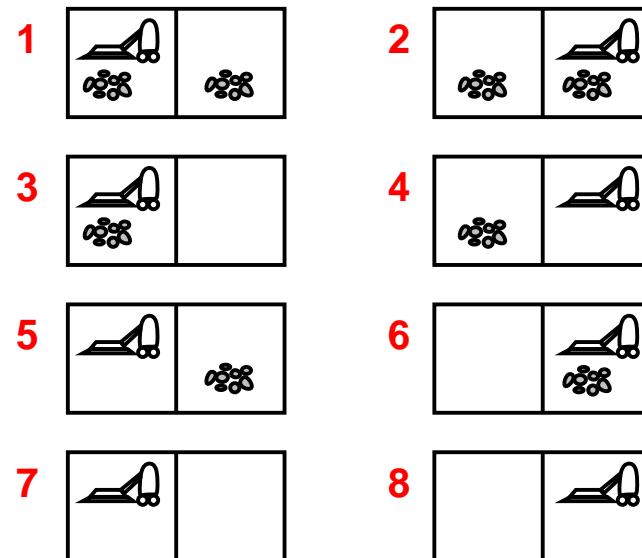
Example: vacuum world

Single-state, start in #5. [Solution??](#)

[*Right*, *Suck*]

Conformant, start in $\{1, 2, 3, 4, 5, 6, 7, 8\}$

e.g., *Right* goes to $\{2, 4, 6, 8\}$. [Solution??](#)



Example: vacuum world

Single-state, start in #5. [Solution??](#)

[*Right, Suck*]

Conformant, start in $\{1, 2, 3, 4, 5, 6, 7, 8\}$

e.g., *Right* goes to $\{2, 4, 6, 8\}$. [Solution??](#)

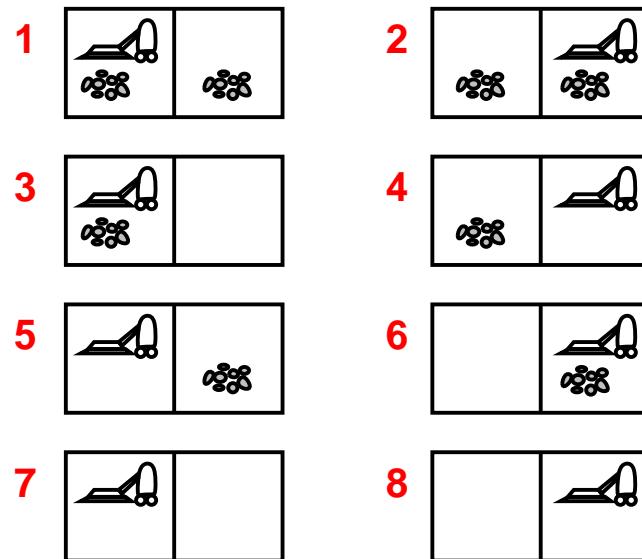
[*Right, Suck, Left, Suck*]

Contingency, start in #5

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

Local sensing: dirt, location only.

[Solution??](#)



Example: vacuum world

Single-state, start in #5. [Solution??](#)

[*Right, Suck*]

Conformant, start in {1, 2, 3, 4, 5, 6, 7, 8}

e.g., *Right* goes to {2, 4, 6, 8}. [Solution??](#)

[*Right, Suck, Left, Suck*]

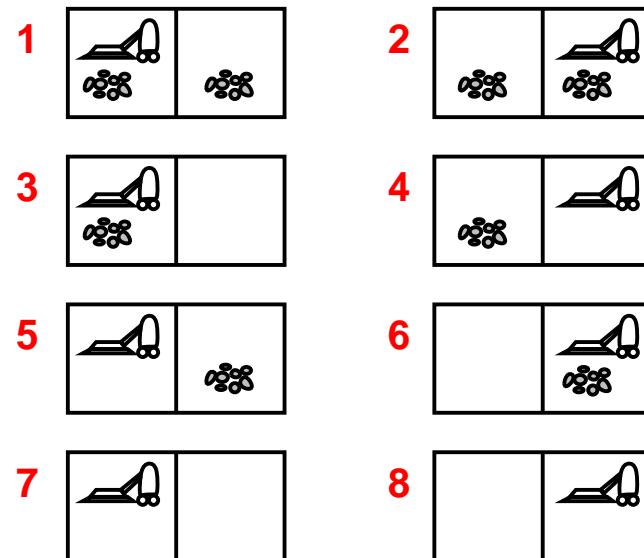
Contingency, start in #5

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

Local sensing: dirt, location only.

[Solution??](#)

[*Right, if dirt then Suck*]



Single-state problem formulation

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

initial state e.g., “at Arad”

successor function $S(x)$ = set of action–state pairs

e.g., $S(Arad) = \{\langle Arad \rightarrow Zerind, Zerind \rangle, \dots\}$

goal test, can be

explicit, e.g., x = “at Bucharest”

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

path cost (additive)

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

$c(x, a, y)$ is the **step cost**, assumed to be ≥ 0

A **solution** is a sequence of actions

leading from the initial state to a goal state

Selecting a state space

Real world is absurdly complex

⇒ state space must be **abstracted** for problem solving

(Abstract) state = set of real states

(Abstract) action = complex combination of real actions

e.g., “Arad → Zerind” represents a complex set
of possible routes, detours, rest stops, etc.

For guaranteed realizability, **any** real state “in Arad”

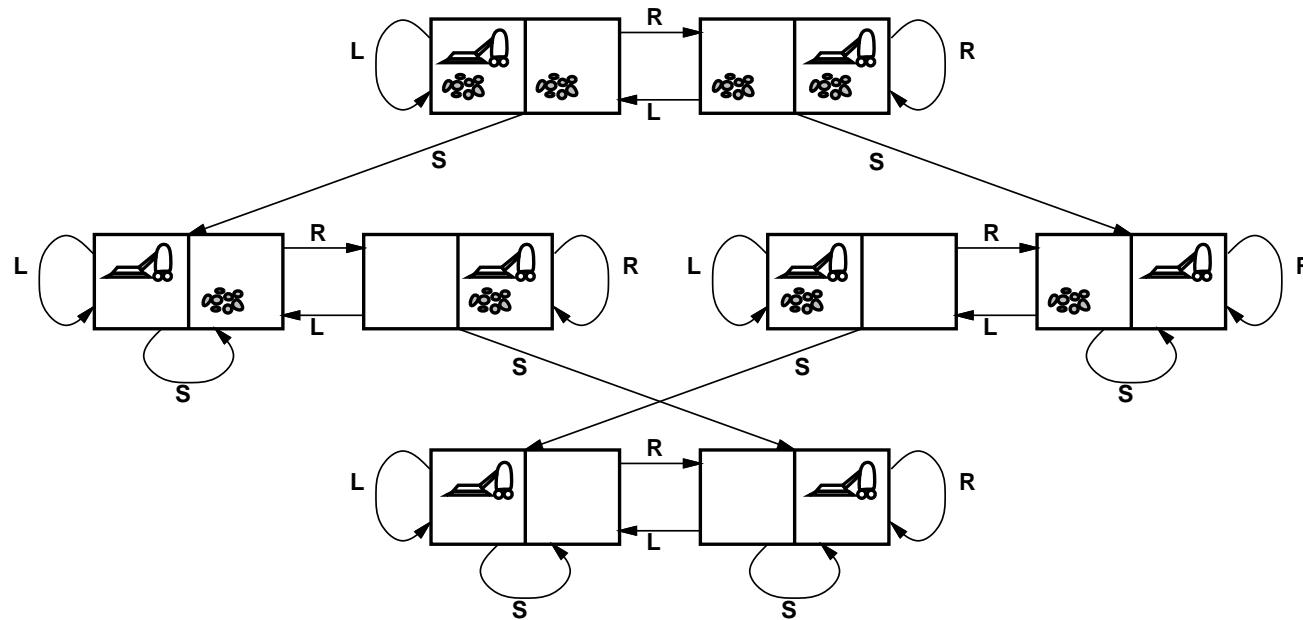
must get to some real state “in Zerind”

(Abstract) solution =

set of real paths that are solutions in the real world

Each abstract action should be “easier” than the original problem!

Example: vacuum world state space graph



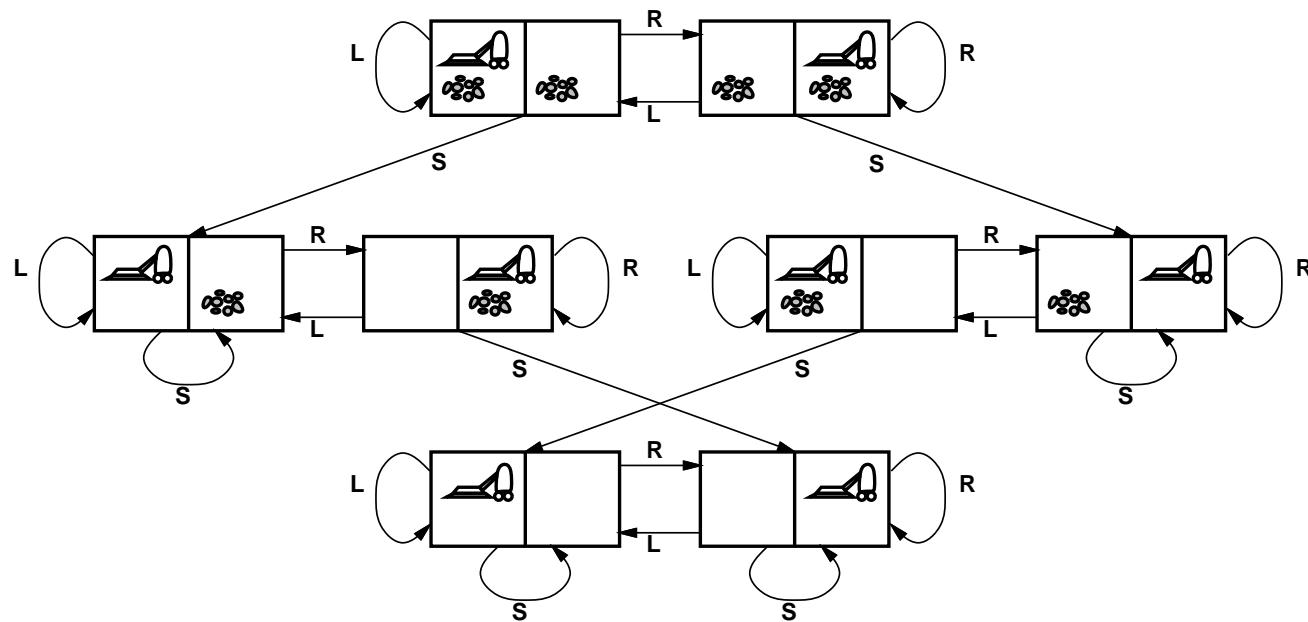
states??

actions??

goal test??

path cost??

Example: vacuum world state space graph



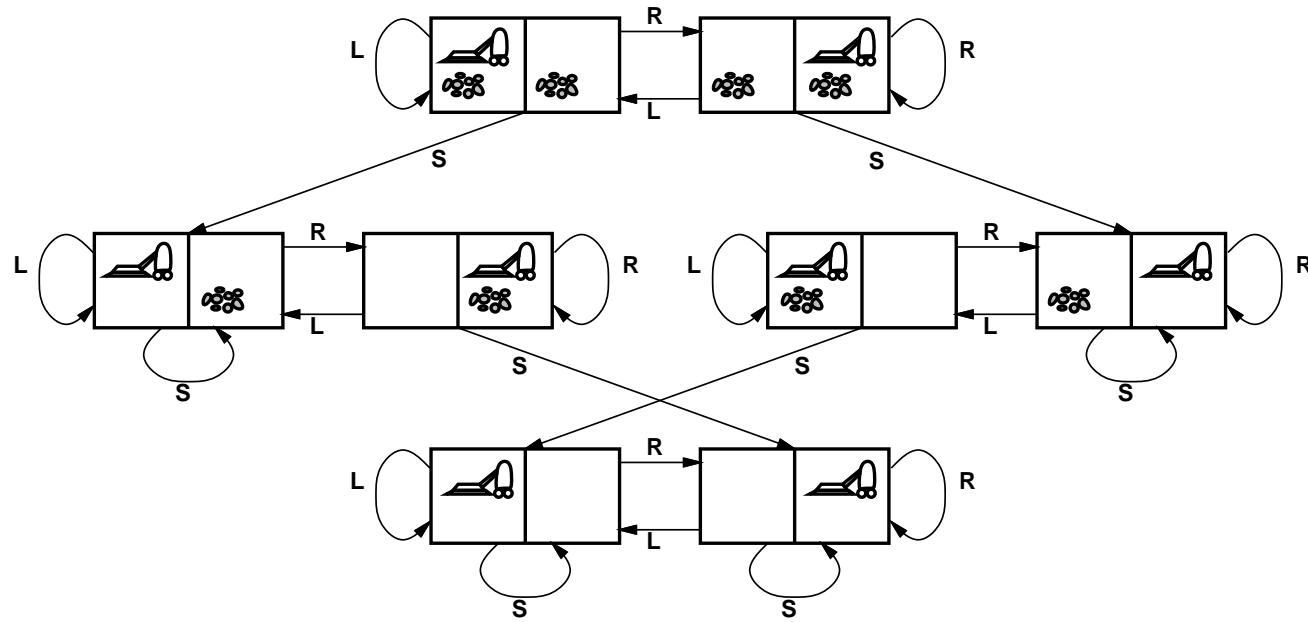
states??: integer dirt and robot locations (ignore dirt amounts etc.)

actions??

goal test??

path cost??

Example: vacuum world state space graph



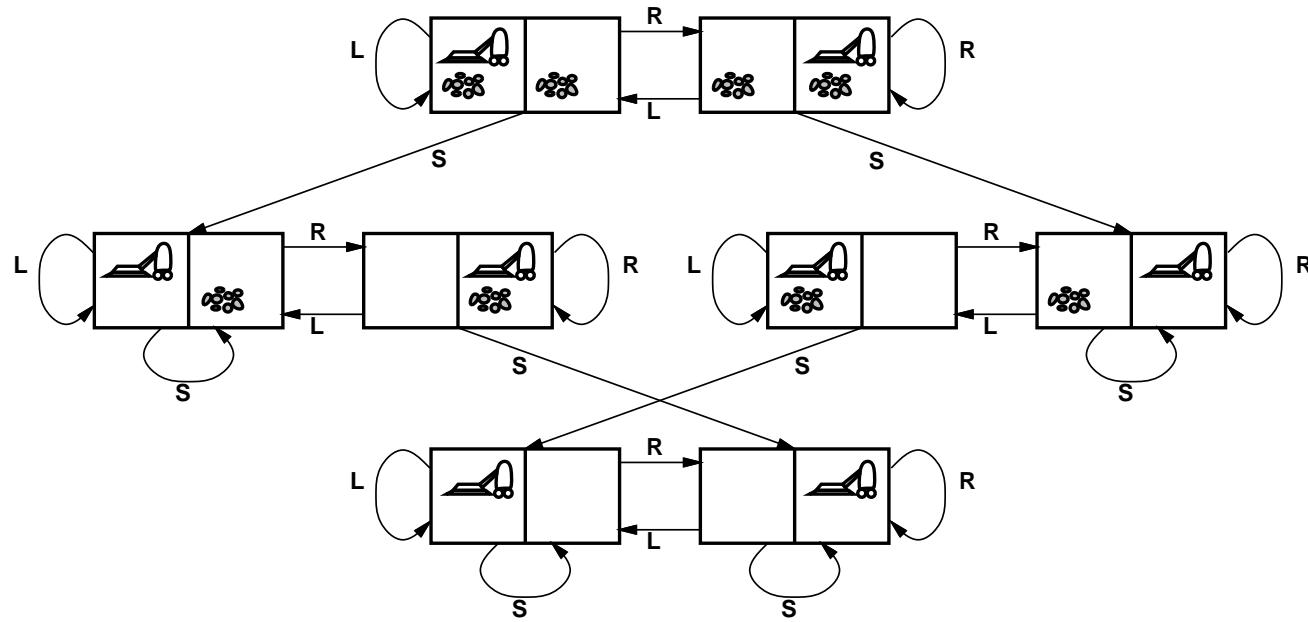
states??: integer dirt and robot locations (ignore dirt amounts etc.)

actions??: *Left*, *Right*, *Suck*, *NoOp*

goal test??

path cost??

Example: vacuum world state space graph



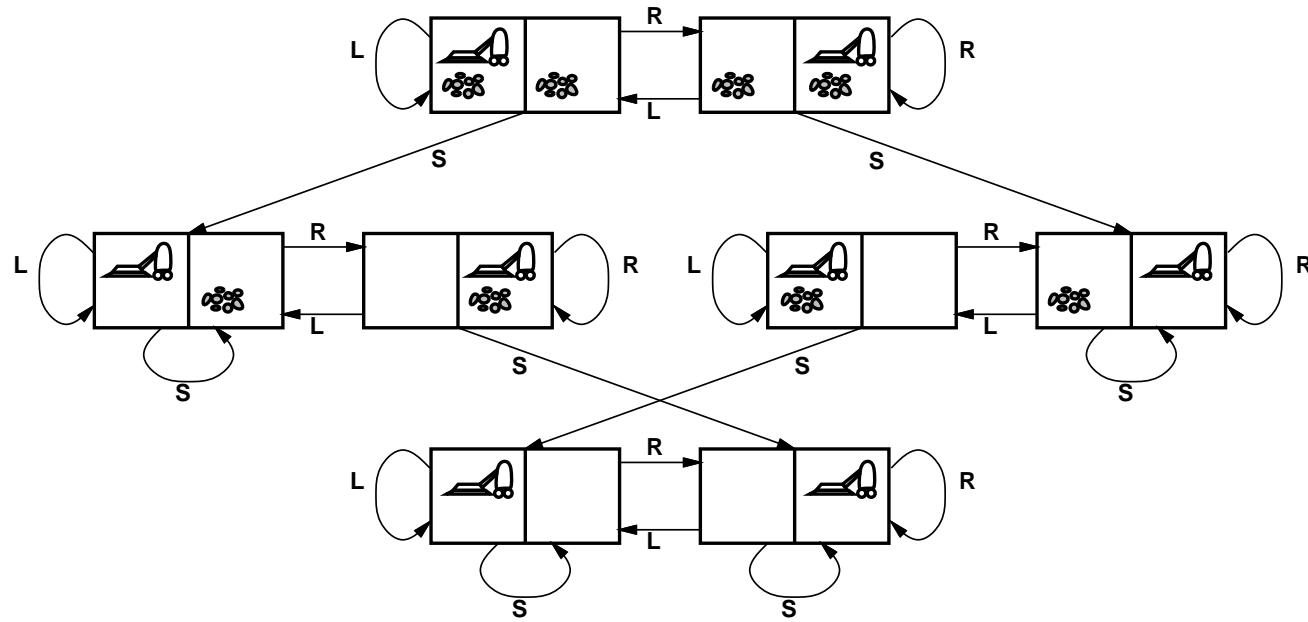
states??: integer dirt and robot locations (ignore dirt amounts etc.)

actions??: *Left*, *Right*, *Suck*, *NoOp*

goal test??: no dirt

path cost??

Example: vacuum world state space graph



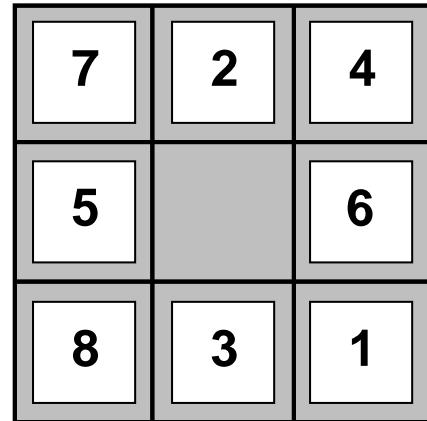
states??: integer dirt and robot locations (ignore dirt amounts etc.)

actions??: *Left*, *Right*, *Suck*, *NoOp*

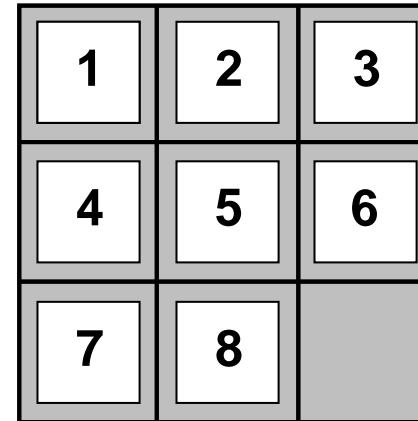
goal test??: no dirt

path cost??: 1 per action (0 for *NoOp*)

Example: The 8-puzzle



Start State



Goal State

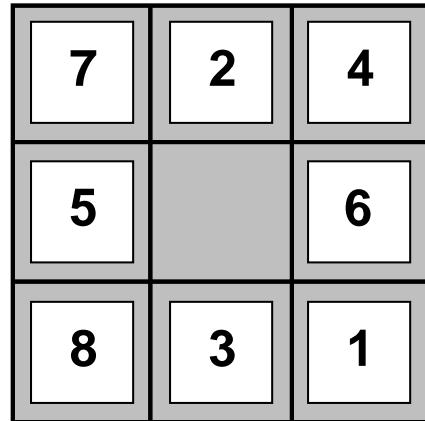
states??

actions??

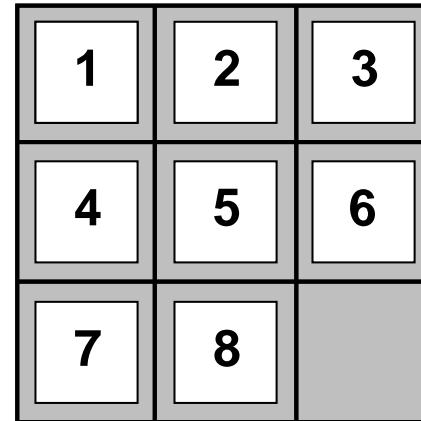
goal test??

path cost??

Example: The 8-puzzle



Start State



Goal State

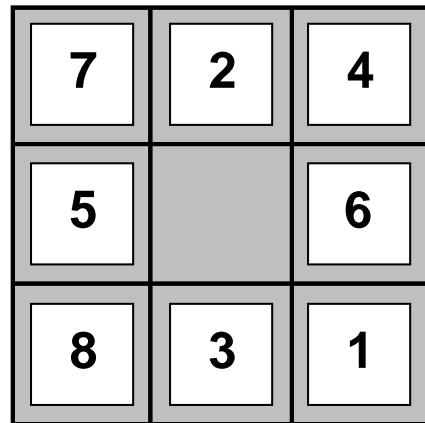
states??: integer locations of tiles (ignore intermediate positions)

actions??

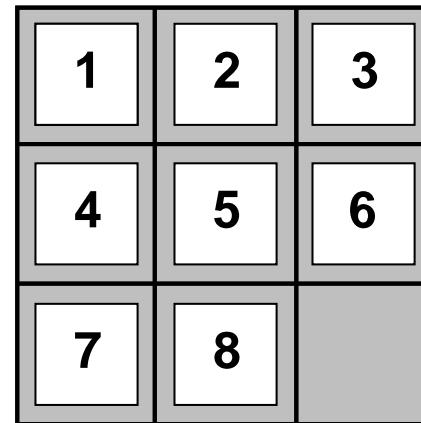
goal test??

path cost??

Example: The 8-puzzle



Start State



Goal State

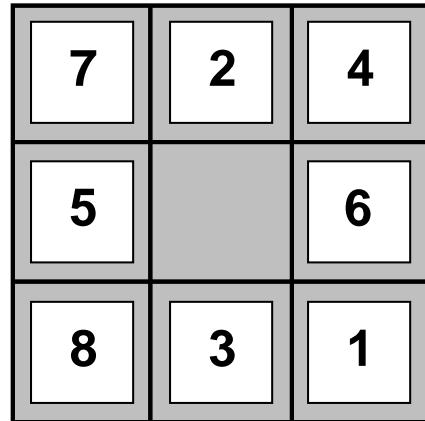
states??: integer locations of tiles (ignore intermediate positions)

actions??: move blank left, right, up, down (ignore unjamming etc.)

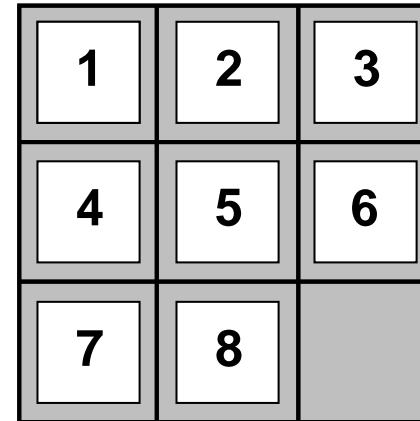
goal test??

path cost??

Example: The 8-puzzle



Start State



Goal State

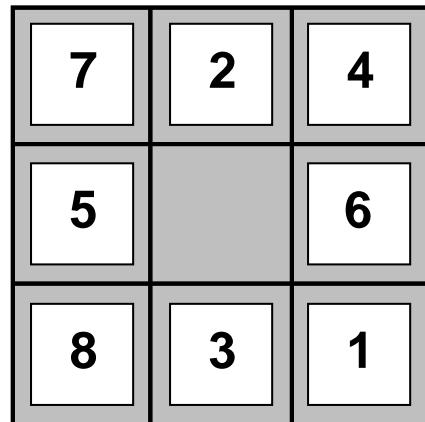
states??: integer locations of tiles (ignore intermediate positions)

actions??: move blank left, right, up, down (ignore unjamming etc.)

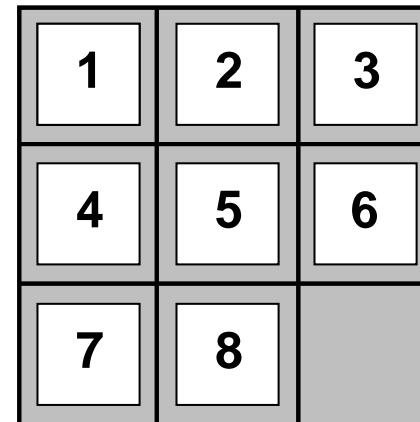
goal test??: = goal state (given)

path cost??

Example: The 8-puzzle



Start State



Goal State

states??: integer locations of tiles (ignore intermediate positions)

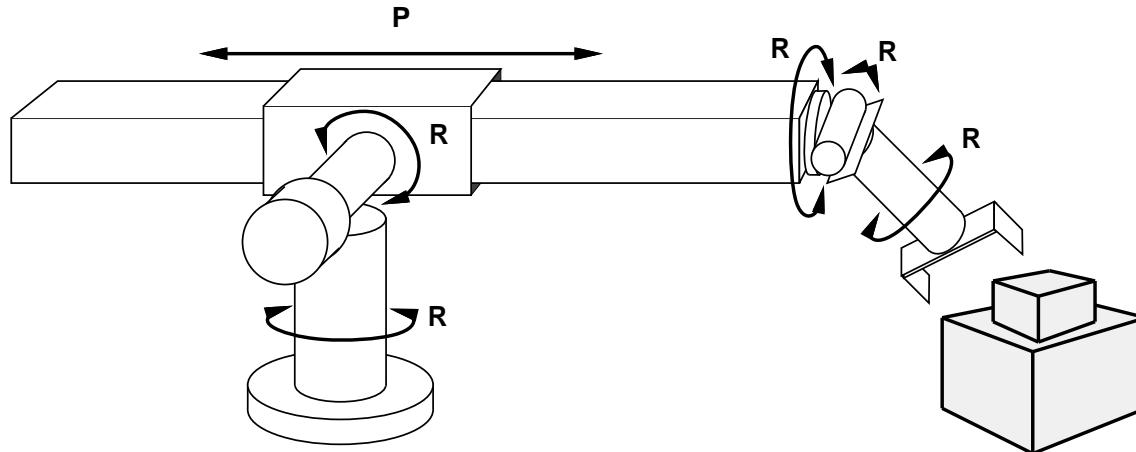
actions??: move blank left, right, up, down (ignore unjamming etc.)

goal test??: = goal state (given)

path cost??: 1 per move

[Note: optimal solution of n -Puzzle family is NP-hard]

Example: robotic assembly



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

actions??: continuous motions of robot joints

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

path cost??: time to execute

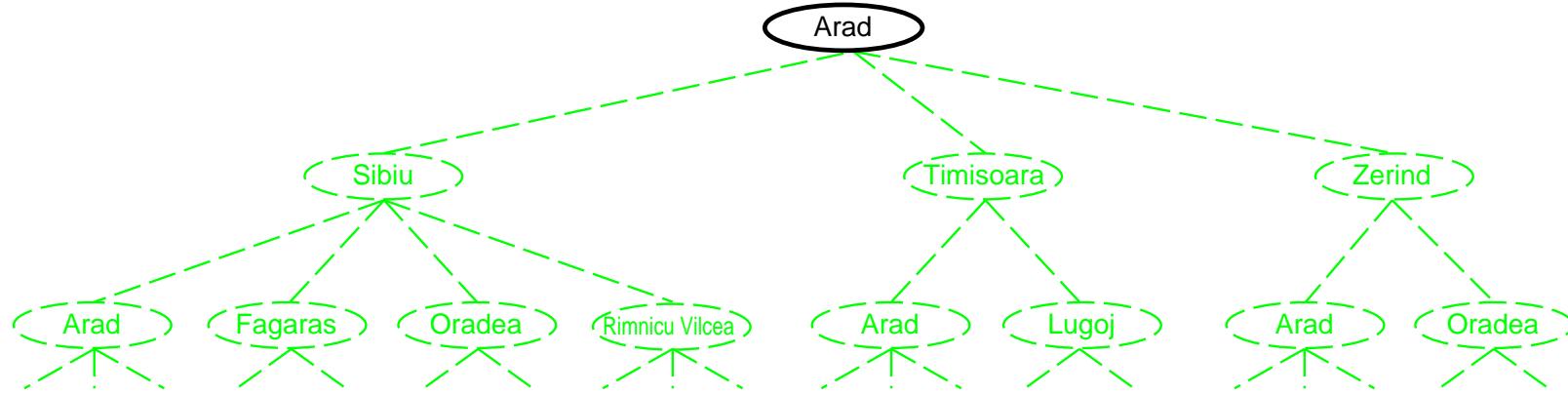
Tree search algorithms

Basic idea:

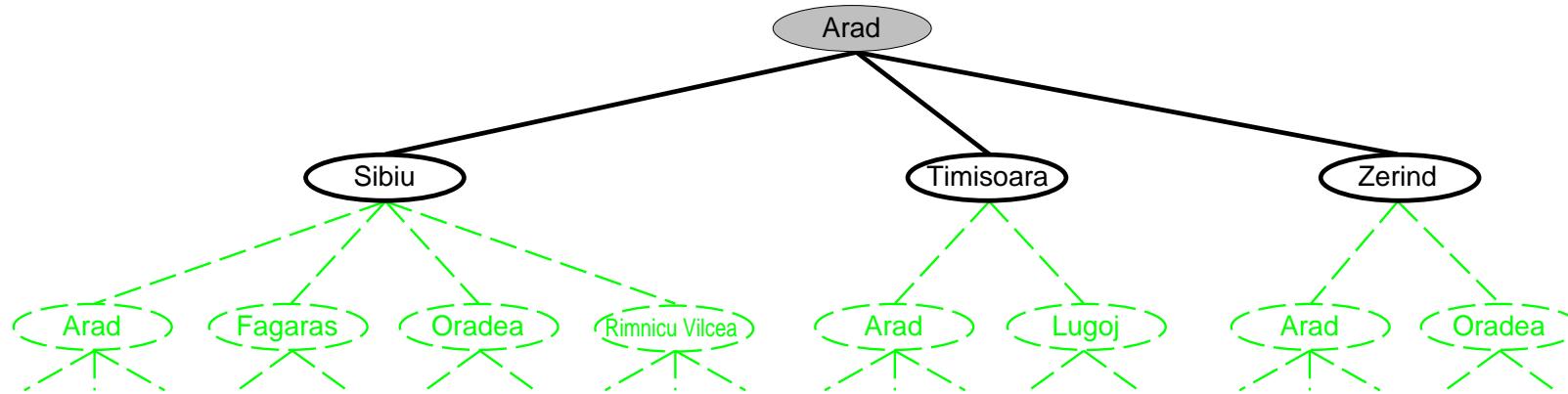
offline, simulated exploration of state space
by generating successors of already-explored states
(a.k.a. **expanding** states)

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

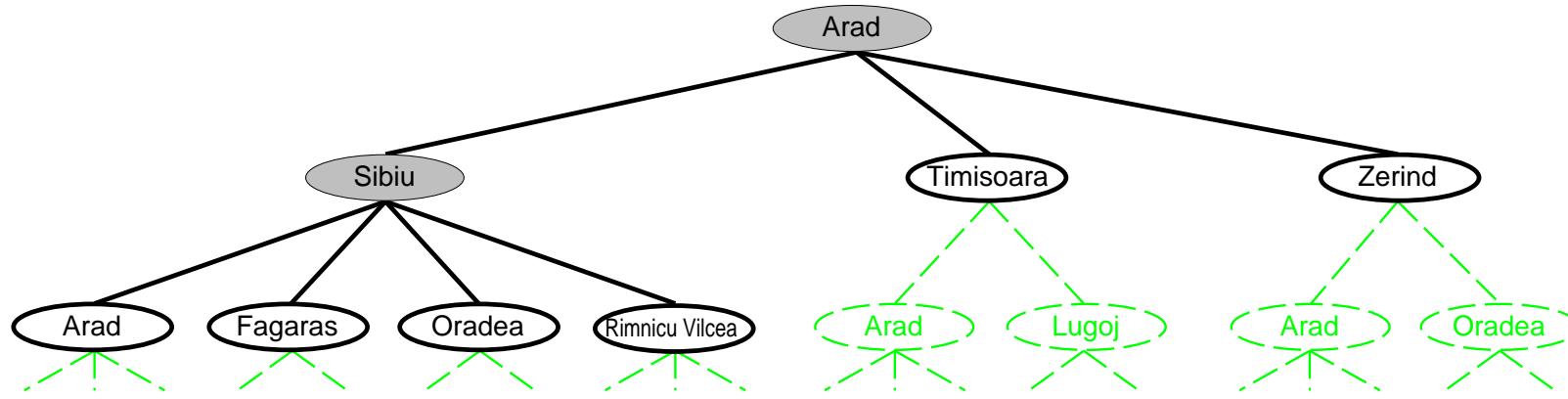
Tree search example



Tree search example



Tree search example

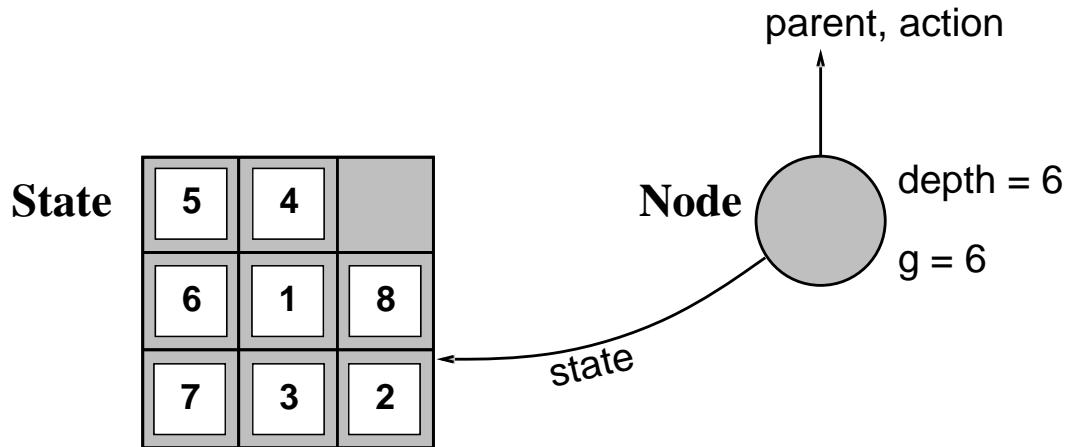


Implementation: states vs. nodes

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 SUCCESSORFn of the problem to create the corresponding states.

Implementation: general tree search

```
function TREE-SEARCH(problem, fringe) returns a solution, or failure
  fringe  $\leftarrow$  INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
  loop do
    if fringe is empty then return failure
    node  $\leftarrow$  REMOVE-FRONT(fringe)
    if GOAL-TEST(problem, STATE(node)) then return node
    fringe  $\leftarrow$  INSERTALL(EXPAND(node, problem), fringe)
```

```
function EXPAND(node, problem) returns a set of nodes
  successors  $\leftarrow$  the empty set
  for each action, result in SUCCESSOR-FN(problem, STATE[node]) do
    s  $\leftarrow$  a new NODE
    PARENT-NODE[s]  $\leftarrow$  node; ACTION[s]  $\leftarrow$  action; STATE[s]  $\leftarrow$  result
    PATH-COST[s]  $\leftarrow$  PATH-COST[node] + STEP-COST(node, action, s)
    DEPTH[s]  $\leftarrow$  DEPTH[node] + 1
    add s to successors
  return successors
```

Search strategies

A strategy is defined by picking the **order of node expansion**

Strategies are evaluated along the following dimensions:

completeness—does it always find a solution if one exists?

time complexity—number of nodes generated/expanded

space complexity—maximum number of nodes in memory

optimality—does it always find a least-cost solution?

Time and space complexity are measured in terms of

b —maximum branching factor of the search tree

d —depth of the least-cost solution

m —maximum depth of the state space (may be ∞)

Uninformed search strategies

Uninformed strategies use only the information available in the problem definition

Breadth-first search

Uniform-cost search

Depth-first search

Depth-limited search

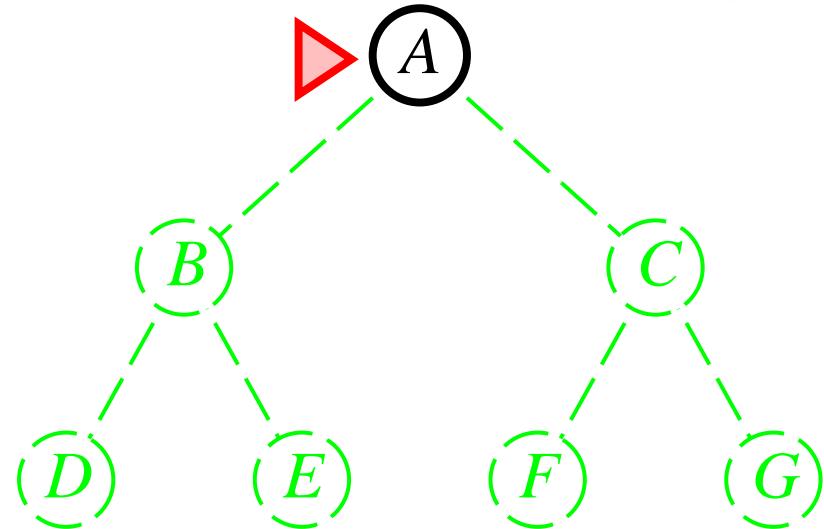
Iterative deepening search

Breadth-first search

Expand shallowest unexpanded node

Implementation:

fringe is a FIFO queue, i.e., new successors go at end

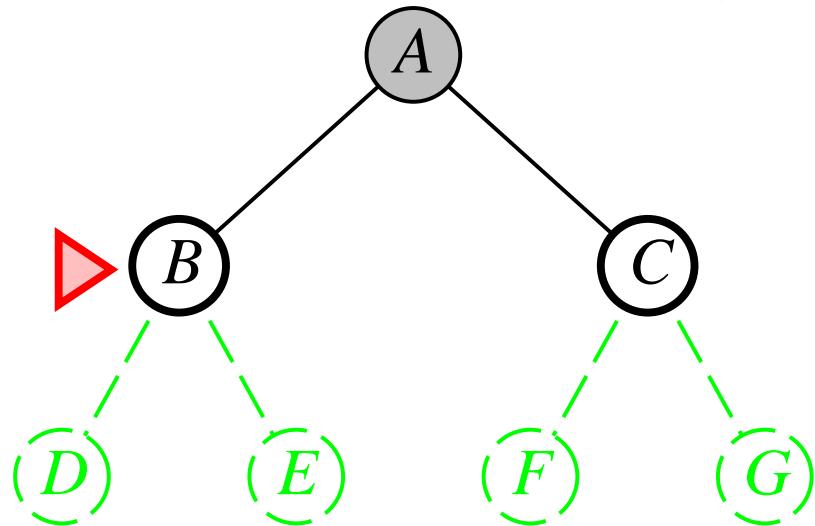


Breadth-first search

Expand shallowest unexpanded node

Implementation:

fringe is a FIFO queue, i.e., new successors go at end

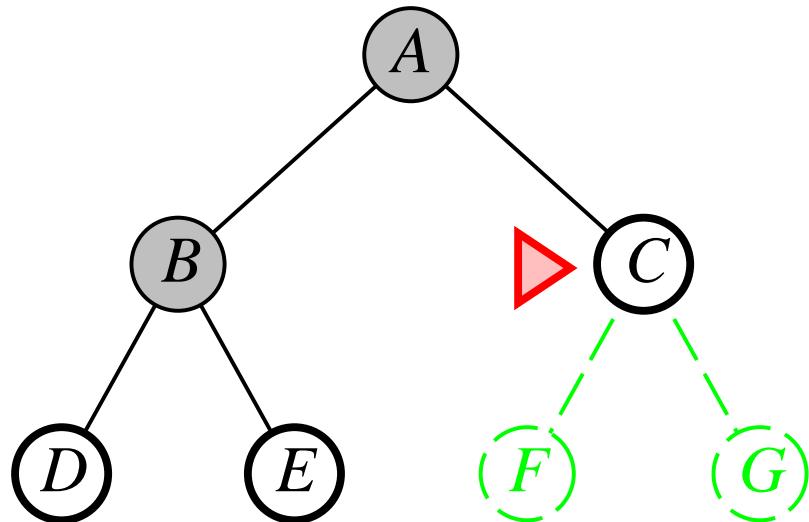


Breadth-first search

Expand shallowest unexpanded node

Implementation:

fringe is a FIFO queue, i.e., new successors go at end

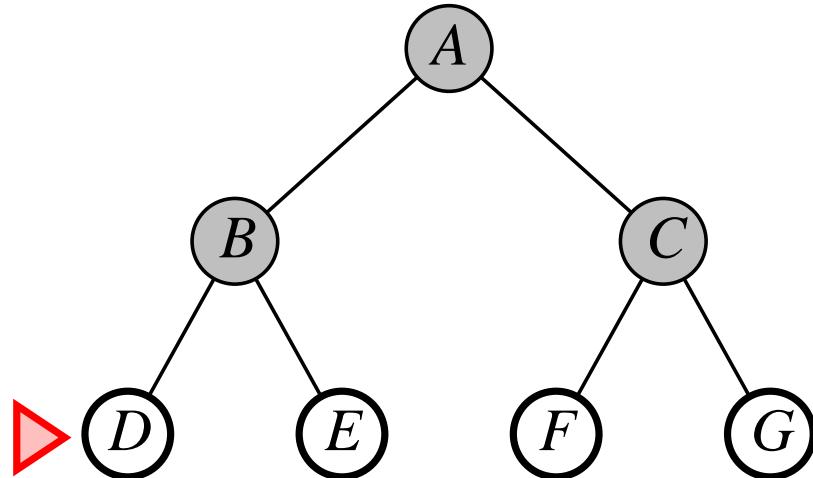


Breadth-first search

Expand shallowest unexpanded node

Implementation:

fringe is a FIFO queue, i.e., new successors go at end



Properties of breadth-first search

Complete??

Properties of breadth-first search

Complete?? Yes (if b is finite)

Time??

Properties of breadth-first search

Complete?? Yes (if b is finite)

Time?? $1 + b + b^2 + b^3 + \dots + b^d + b(b^d - 1) = O(b^{d+1})$, i.e., exp. in d

Space??

Properties of breadth-first search

Complete?? Yes (if b is finite)

Time?? $1 + b + b^2 + b^3 + \dots + b^d + b(b^d - 1) = O(b^{d+1})$, i.e., exp. in d

Space?? $O(b^{d+1})$ (keeps every node in memory)

Optimal??

Properties of breadth-first search

Complete?? Yes (if b is finite)

Time?? $1 + b + b^2 + b^3 + \dots + b^d + b(b^d - 1) = O(b^{d+1})$, i.e., exp. in d

Space?? $O(b^{d+1})$ (keeps every node in memory)

Optimal?? Yes (if cost = 1 per step); not optimal in general

Space is the big problem; can easily generate nodes at 100MB/sec
so 24hrs = 8640GB.

Uniform-cost search

Expand least-cost unexpanded node

Implementation:

fringe = queue ordered by path cost, lowest first

Equivalent to breadth-first if step costs all equal

Complete?? Yes, if step cost $\geq \epsilon$

Time?? # of nodes with $g \leq$ cost of optimal solution, $O(b^{\lceil C^*/\epsilon \rceil})$
where C^* is the cost of the optimal solution

Space?? # of nodes with $g \leq$ cost of optimal solution, $O(b^{\lceil C^*/\epsilon \rceil})$

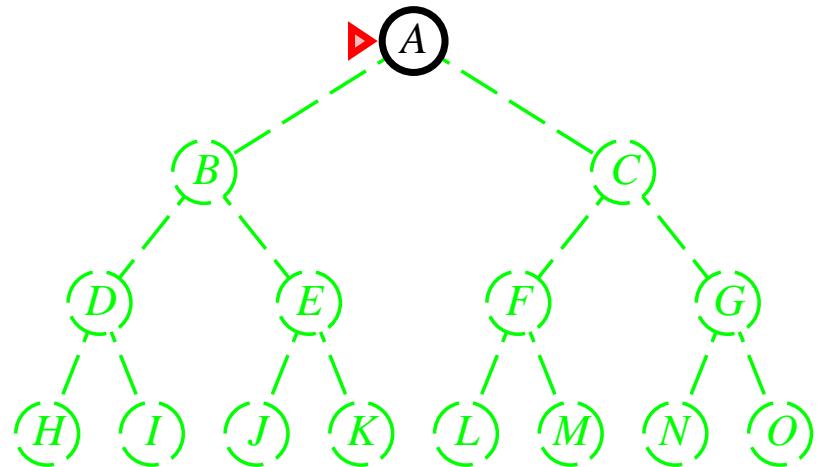
Optimal?? Yes—nodes expanded in increasing order of $g(n)$

Depth-first search

Expand deepest unexpanded node

Implementation:

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

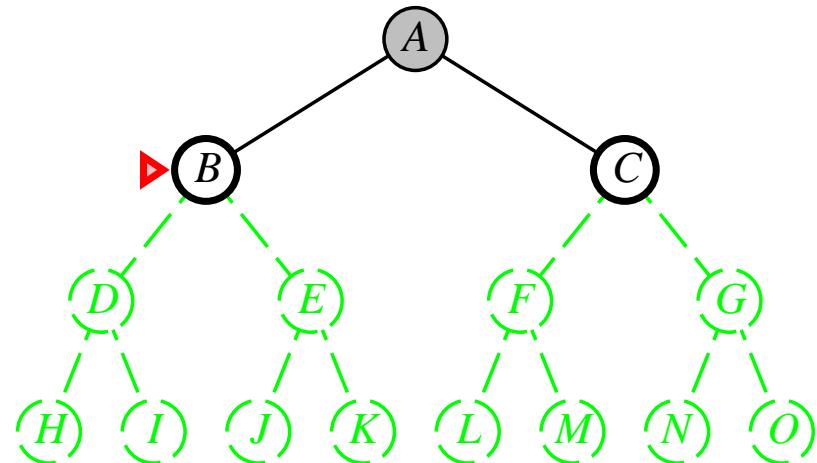


Depth-first search

Expand deepest unexpanded node

Implementation:

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

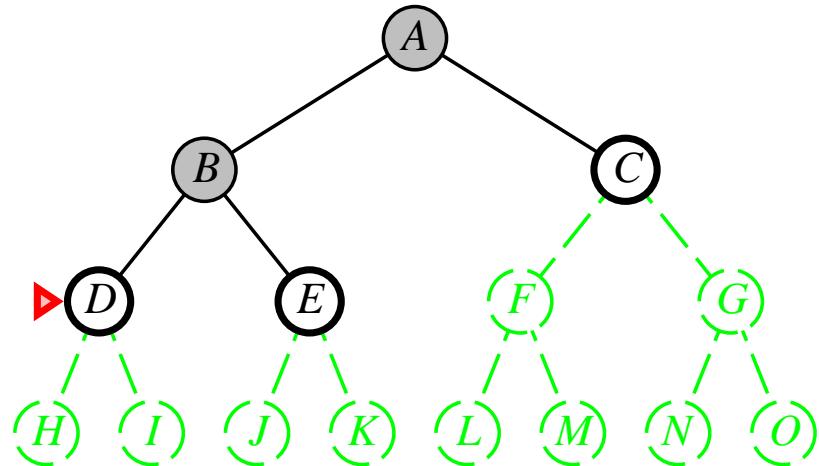


Depth-first search

Expand deepest unexpanded node

Implementation:

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

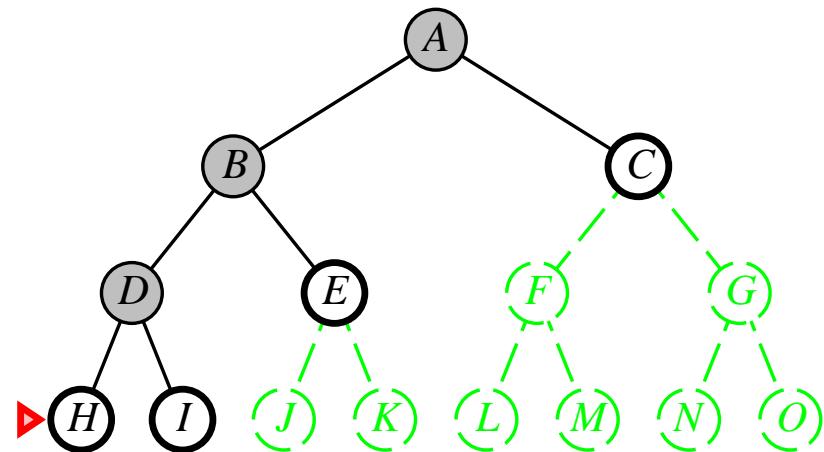


Depth-first search

Expand deepest unexpanded node

Implementation:

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

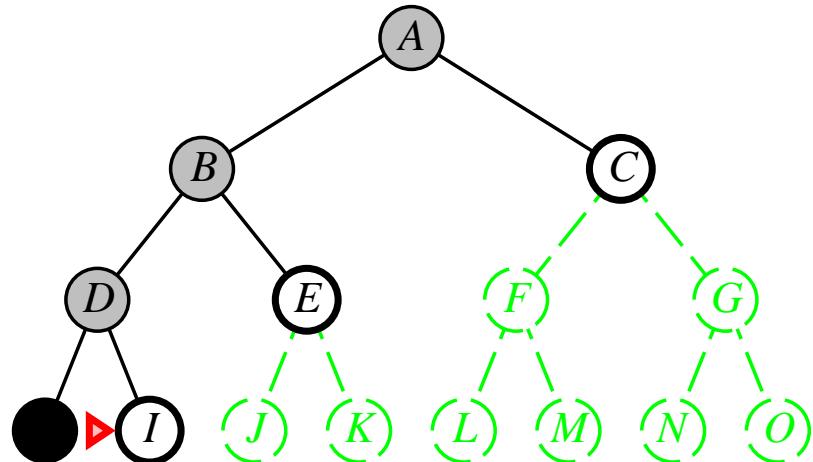


Depth-first search

Expand deepest unexpanded node

Implementation:

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

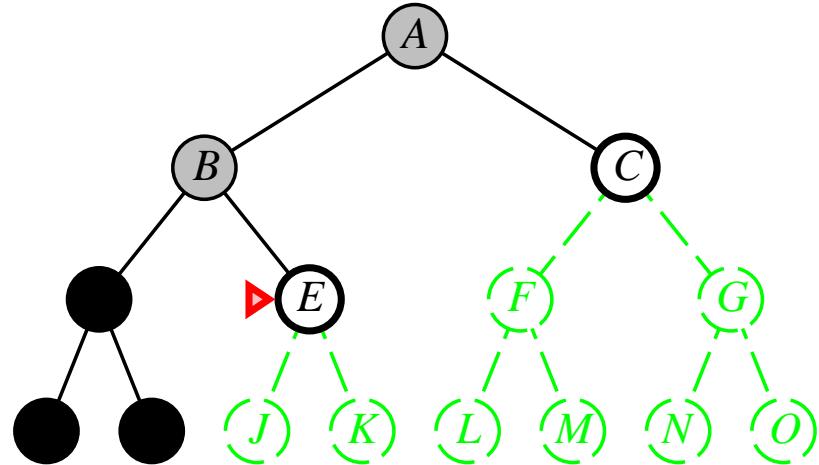


Depth-first search

Expand deepest unexpanded node

Implementation:

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

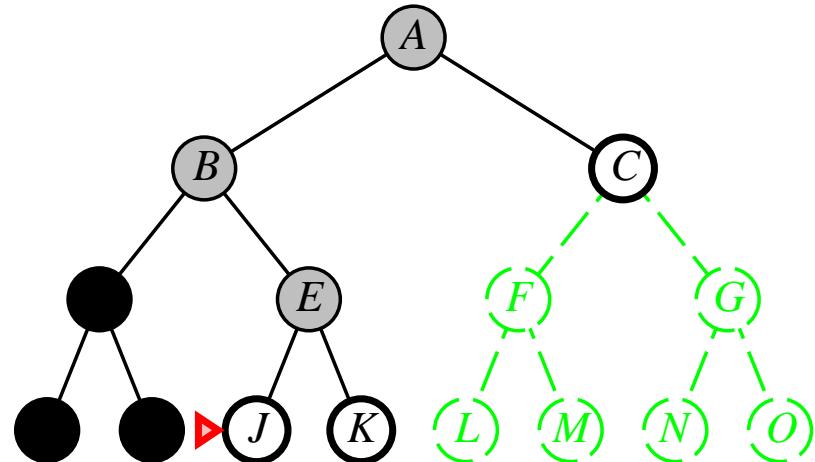


Depth-first search

Expand deepest unexpanded node

Implementation:

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

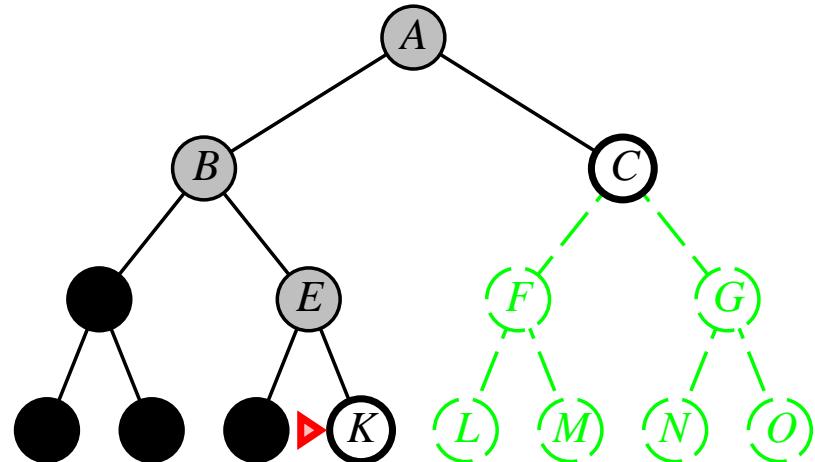


Depth-first search

Expand deepest unexpanded node

Implementation:

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

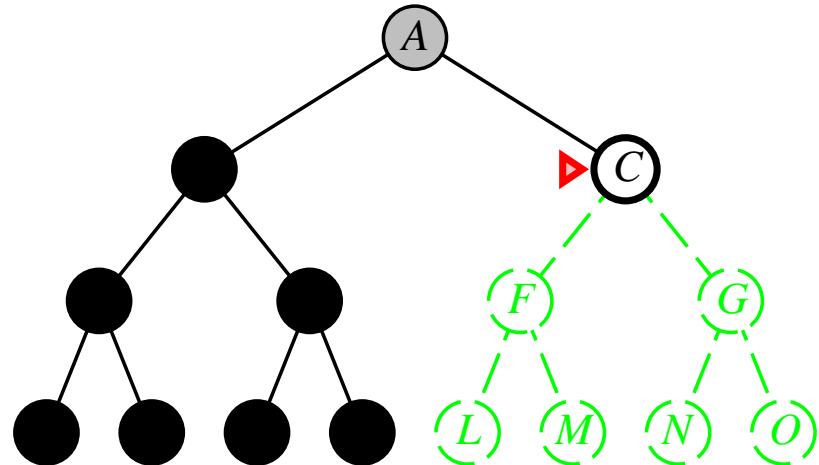


Depth-first search

Expand deepest unexpanded node

Implementation:

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

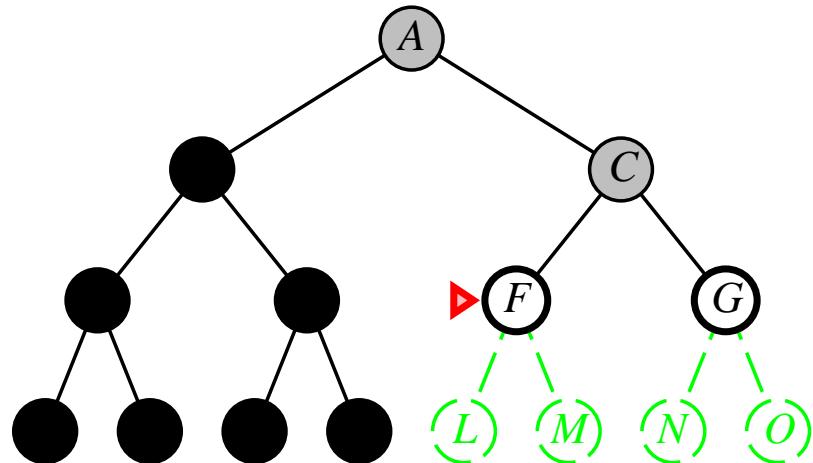


Depth-first search

Expand deepest unexpanded node

Implementation:

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

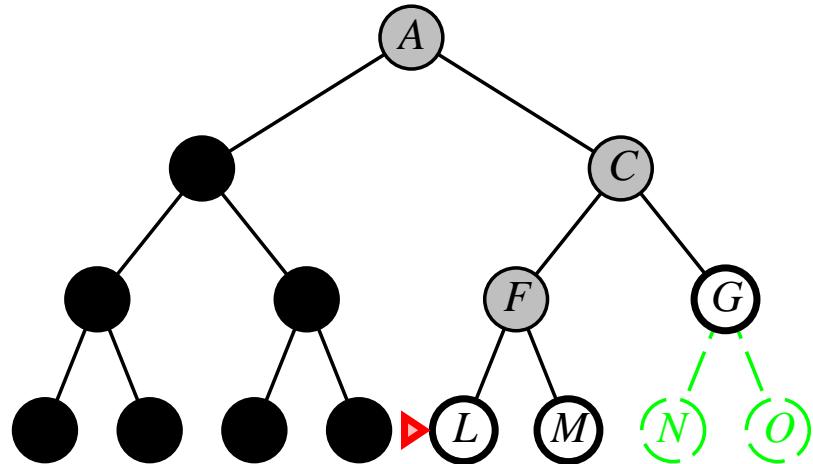


Depth-first search

Expand deepest unexpanded node

Implementation:

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

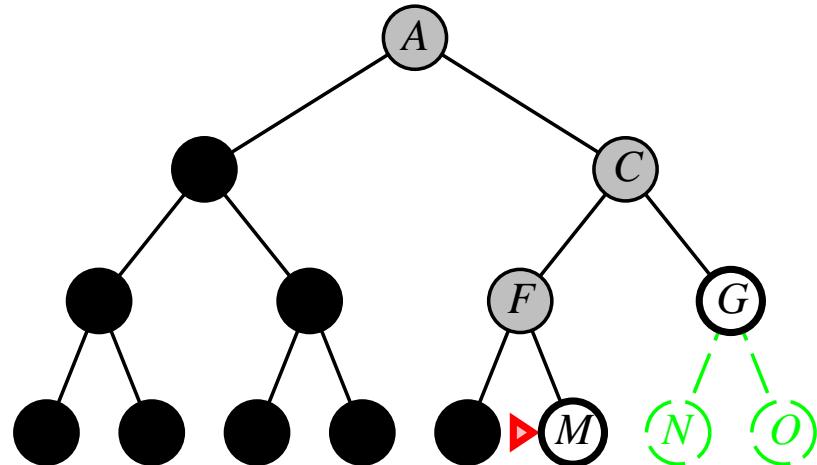


Depth-first search

Expand deepest unexpanded node

Implementation:

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



Properties of depth-first search

Complete??

Properties of depth-first search

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

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

Time??

Properties of depth-first search

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

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

Time?? $O(b^m)$: terrible if m is much larger than d
but if solutions are dense, may be much faster than breadth-first

Space??

Properties of depth-first search

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

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

Time?? $O(b^m)$: terrible if m is much larger than d
but if solutions are dense, may be much faster than breadth-first

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

Optimal??

Properties of depth-first search

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

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

Time?? $O(b^m)$: terrible if m is much larger than d
but if solutions are dense, may be much faster than breadth-first

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

Optimal?? No

Depth-limited search

= depth-first search with depth limit l ,
i.e., nodes at depth l have no successors

Recursive implementation:

```
function DEPTH-LIMITED-SEARCH( problem, limit) returns soln/fail/cutoff
    RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[problem]), problem, limit)
function RECURSIVE-DLS(node, problem, limit) returns soln/fail/cutoff
    cutoff-occurred?  $\leftarrow$  false
    if GOAL-TEST(problem, STATE[node]) then return node
    else if DEPTH[node] = limit then return cutoff
    else for each successor in EXPAND(node, problem) do
        result  $\leftarrow$  RECURSIVE-DLS(successor, problem, limit)
        if result = cutoff then cutoff-occurred?  $\leftarrow$  true
        else if result  $\neq$  failure then return result
    if cutoff-occurred? then return cutoff else return failure
```

Iterative deepening search

```
function ITERATIVE-DEEPENING-SEARCH( problem ) returns a solution
    inputs: problem, a problem
    for depth  $\leftarrow 0$  to  $\infty$  do
        result  $\leftarrow$  DEPTH-LIMITED-SEARCH( problem, depth )
        if result  $\neq$  cutoff then return result
    end
```

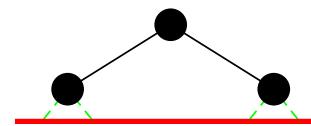
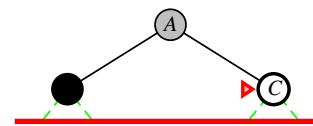
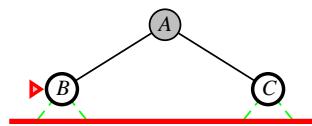
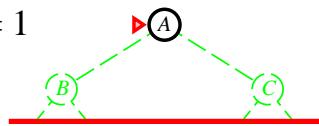
Iterative deepening search $l = 0$

Limit = 0



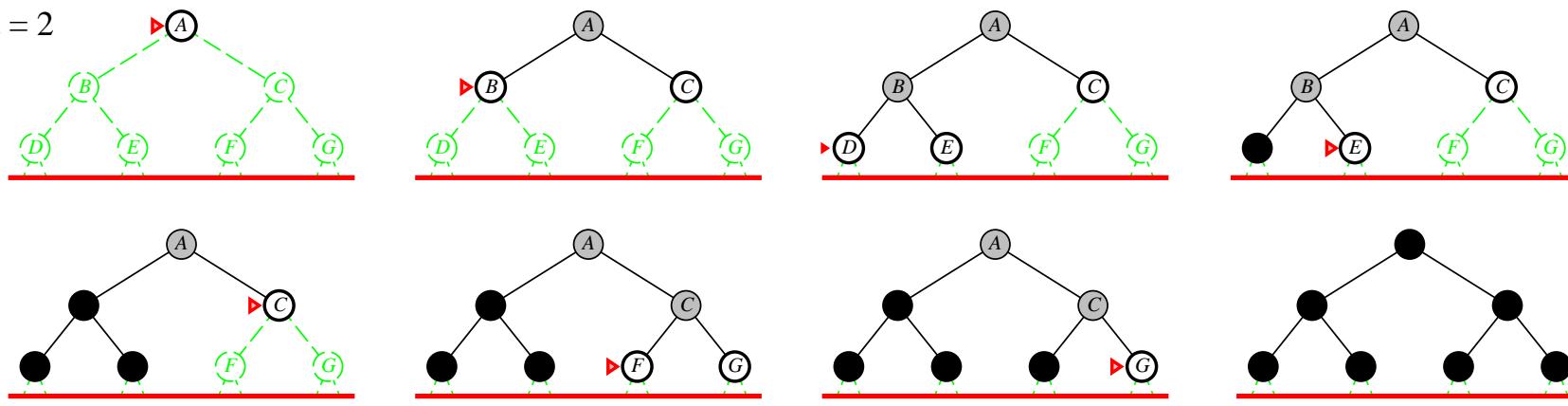
Iterative deepening search $l = 1$

Limit = 1

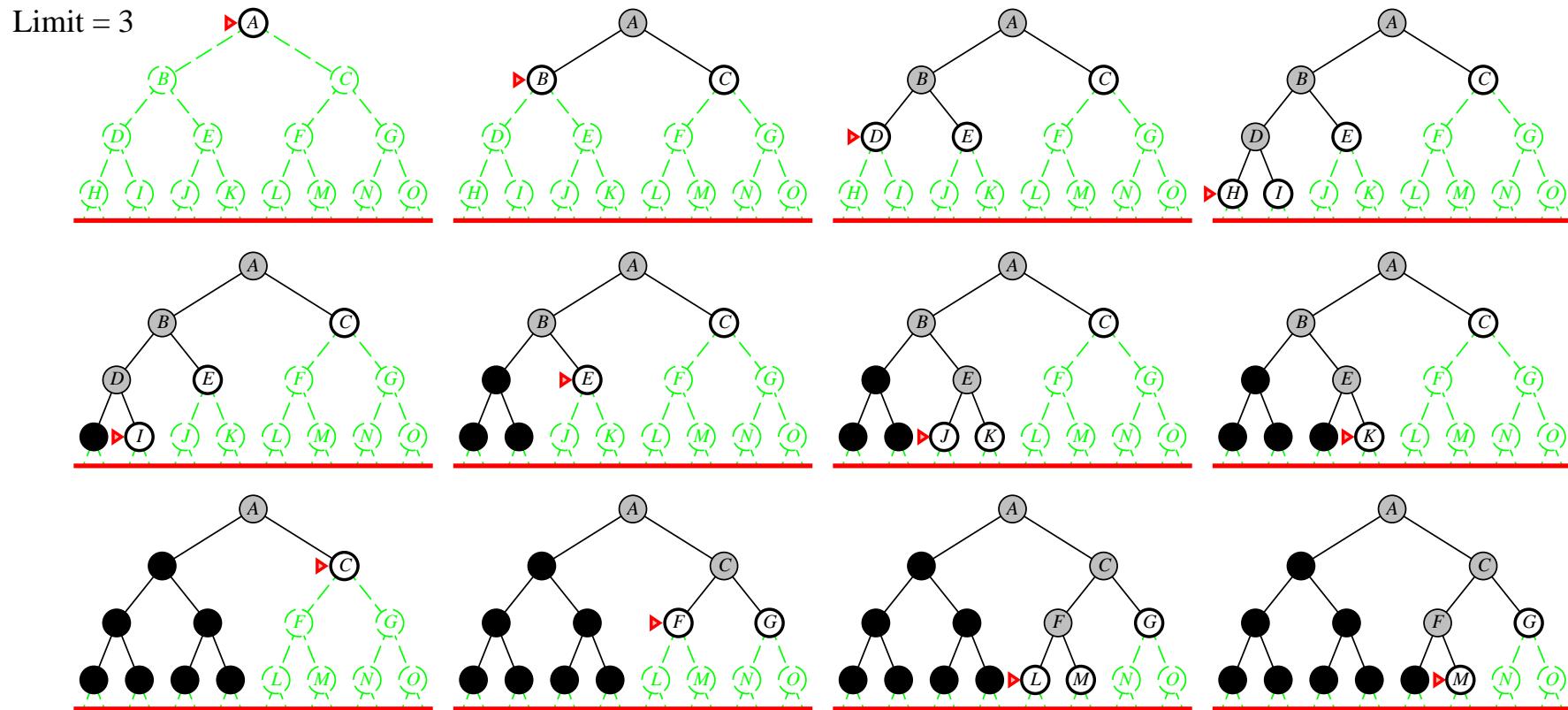


Iterative deepening search $l = 2$

Limit = 2



Iterative deepening search $l = 3$



Properties of iterative deepening search

Complete??

Properties of iterative deepening search

Complete?? Yes

Time??

Properties of iterative deepening search

Complete?? Yes

Time?? $(d + 1)b^0 + db^1 + (d - 1)b^2 + \dots + b^d = O(b^d)$

Space??

Properties of iterative deepening search

Complete?? Yes

Time?? $(d + 1)b^0 + db^1 + (d - 1)b^2 + \dots + b^d = O(b^d)$

Space?? $O(bd)$

Optimal??

Properties of iterative deepening search

Complete?? Yes

Time?? $(d + 1)b^0 + db^1 + (d - 1)b^2 + \dots + b^d = O(b^d)$

Space?? $O(bd)$

Optimal?? Yes, if step cost = 1

Can be modified to explore uniform-cost tree

Numerical comparison for $b = 10$ and $d = 5$, solution at far right leaf:

$$N(\text{IDS}) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450$$

$$N(\text{BFS}) = 10 + 100 + 1,000 + 10,000 + 100,000 + 999,990 = 1,111,100$$

IDS does better because other nodes at depth d are not expanded

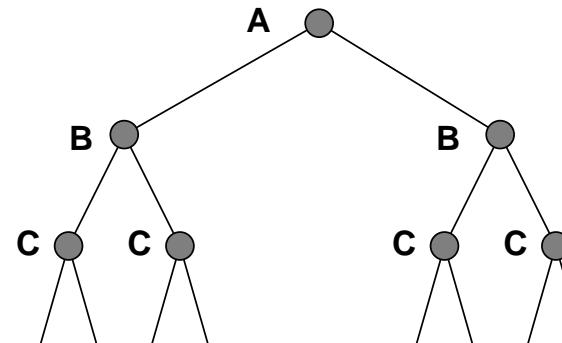
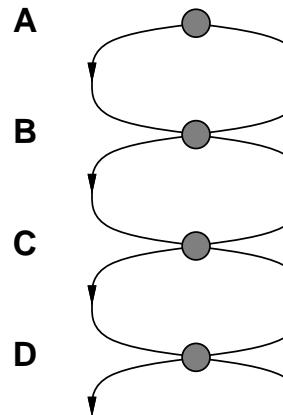
BFS can be modified to apply goal test when a node is **generated**

Summary of algorithms

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

Repeated states

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



Graph search

```
function GRAPH-SEARCH(problem, fringe) returns a solution, or failure
  closed  $\leftarrow$  an empty set
  fringe  $\leftarrow$  INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
  loop do
    if fringe is empty then return failure
    node  $\leftarrow$  REMOVE-FRONT(fringe)
    if GOAL-TEST(problem, STATE[node]) then return node
    if STATE[node] is not in closed then
      add STATE[node] to closed
      fringe  $\leftarrow$  INSERTALL(EXPAND(node, problem), fringe)
  end
```

Summary

Problem formulation usually requires abstracting away real-world details to define a state space that can feasibly be explored

Variety of uninformed search strategies

Iterative deepening search uses only linear space
and not much more time than other uninformed algorithms

Graph search can be exponentially more efficient than tree search

INFORMED SEARCH ALGORITHMS

CHAPTER 4, SECTIONS 1–2

Outline

- ◊ Best-first search
- ◊ A* search
- ◊ Heuristics

Review: Tree search

```
function TREE-SEARCH(problem, fringe) returns a solution, or failure
  fringe  $\leftarrow$  INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
  loop do
    if fringe is empty then return failure
    node  $\leftarrow$  REMOVE-FRONT(fringe)
    if GOAL-TEST[problem] applied to STATE(node) succeeds return node
    fringe  $\leftarrow$  INSERTALL(EXPAND(node, problem), fringe)
```

A strategy is defined by picking the **order of node expansion**

Best-first search

Idea: use an **evaluation function** for each node

- estimate of “desirability”

⇒ Expand most desirable unexpanded node

Implementation:

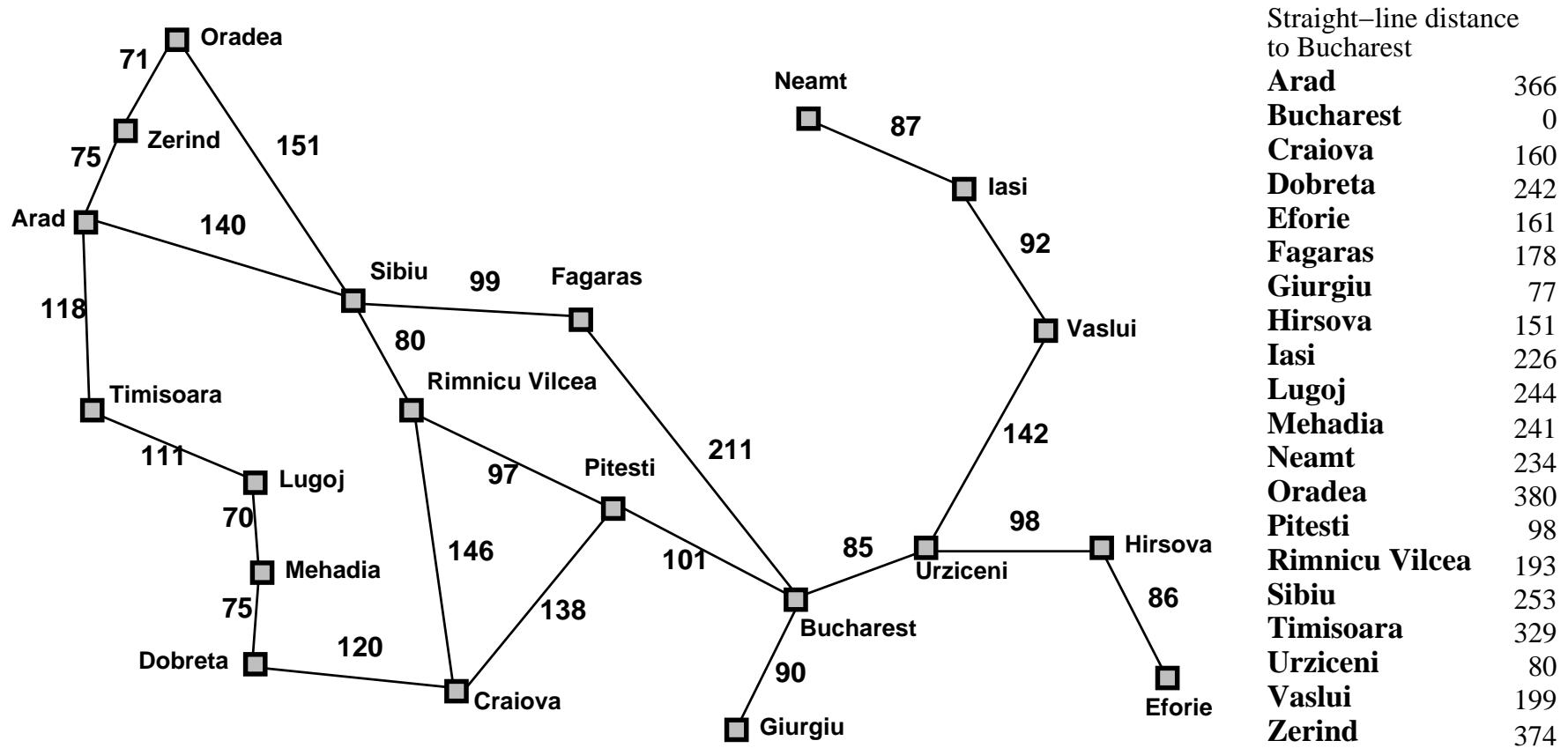
fringe is a queue sorted in decreasing order of desirability

Special cases:

greedy search

A* search

Romania with step costs in km



Greedy search

Evaluation function $h(n)$ (**heuristic**)
= estimate of cost from n to the closest goal

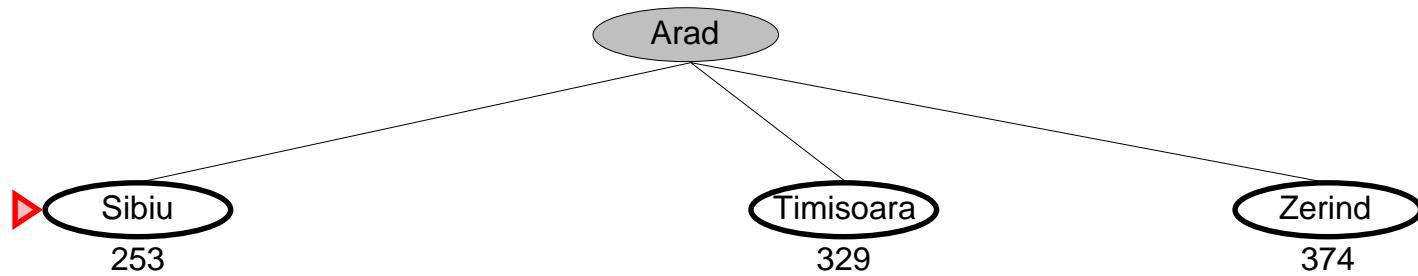
E.g., $h_{\text{SLD}}(n)$ = straight-line distance from n to Bucharest

Greedy search expands the node that **appears** to be closest to goal

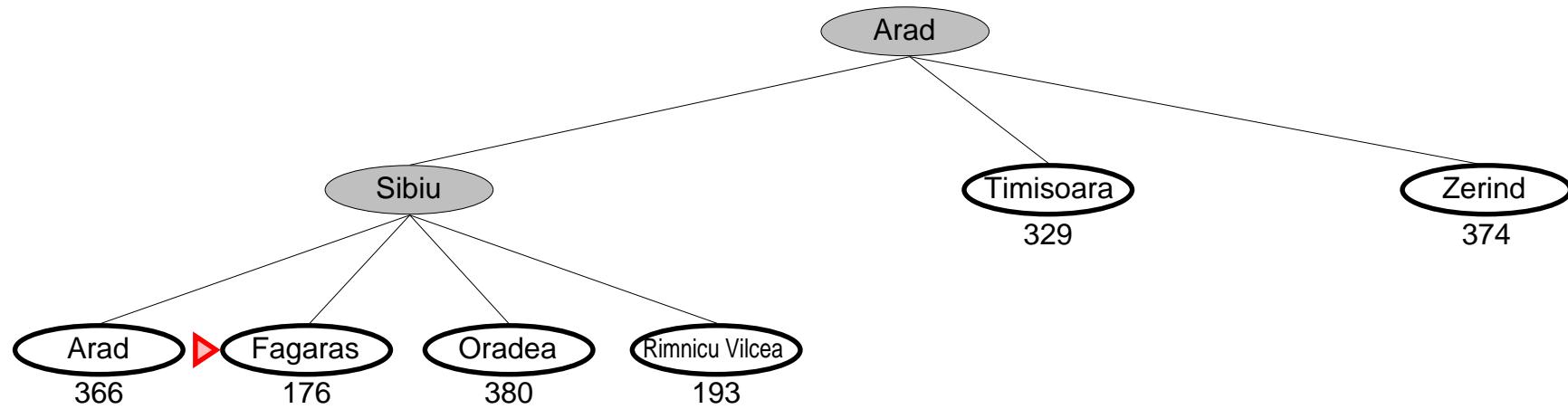
Greedy search example



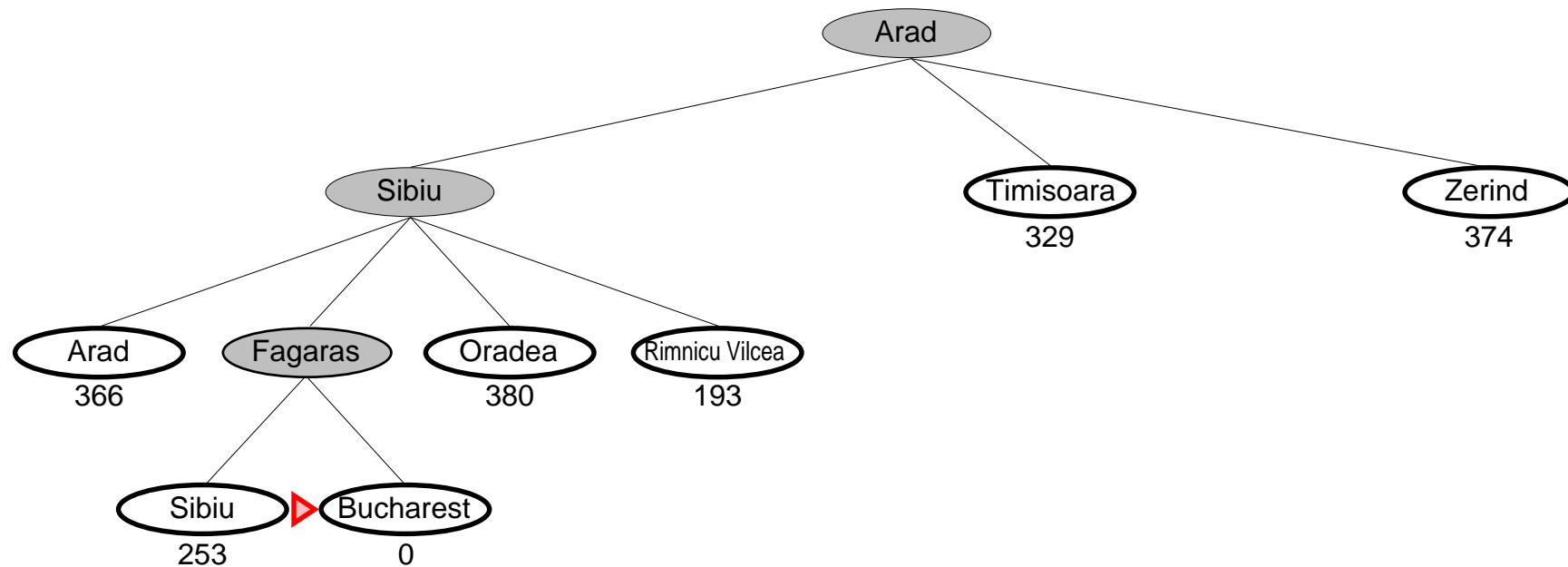
Greedy search example



Greedy search example



Greedy search example



Properties of greedy search

Complete??

Properties of greedy search

Complete?? No—can get stuck in loops, e.g., with Oradea as goal,

lasi → Neamt → laси → Neamt →

Complete in finite space with repeated-state checking

Time??

Properties of greedy search

Complete?? No—can get stuck in loops, e.g.,

lasi → Neamt → lași → Neamț →

Complete in finite space with repeated-state checking

Time?? $O(b^m)$, but a good heuristic can give dramatic improvement

Space??

Properties of greedy search

Complete?? No—can get stuck in loops, e.g.,

lasi → Neamt → lași → 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??

Properties of greedy search

Complete?? No—can get stuck in loops, e.g.,

lasi → Neamt → lași → Neamț →

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

$g(n)$ = cost so far to reach n

$h(n)$ = estimated cost to goal from n

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

A* search uses an **admissible** heuristic

i.e., $h(n) \leq h^*(n)$ where $h^*(n)$ is the **true** cost from n .

(Also require $h(n) \geq 0$, so $h(G) = 0$ for any goal G .)

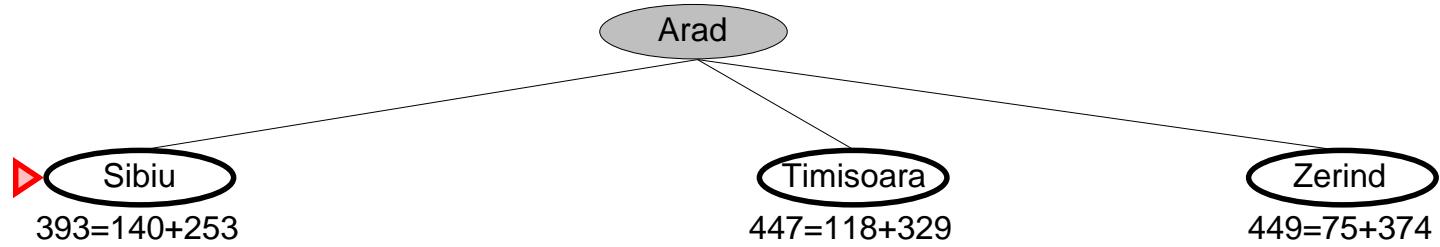
E.g., $h_{\text{SLD}}(n)$ never overestimates the actual road distance

Theorem: A* search is optimal

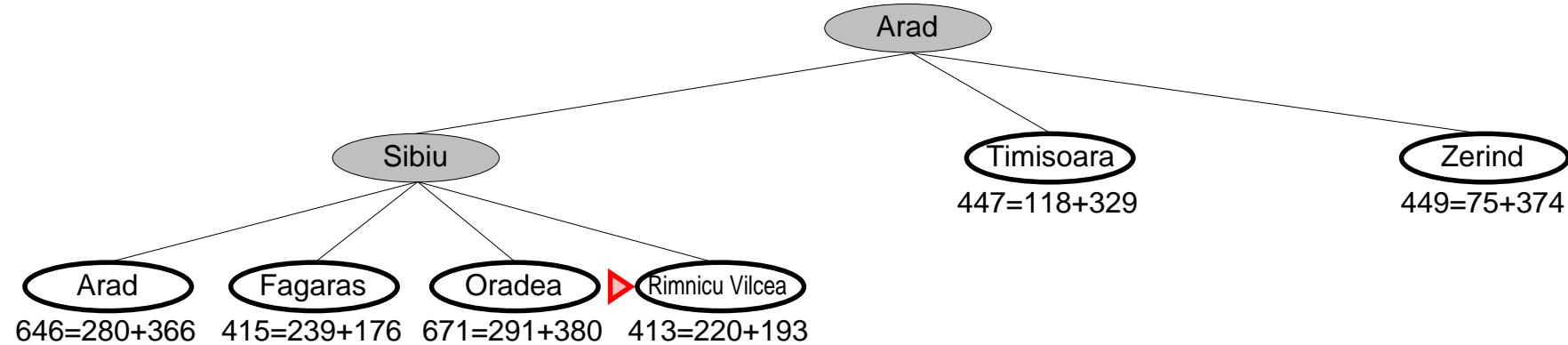
A* search example

► Arad
366=0+366

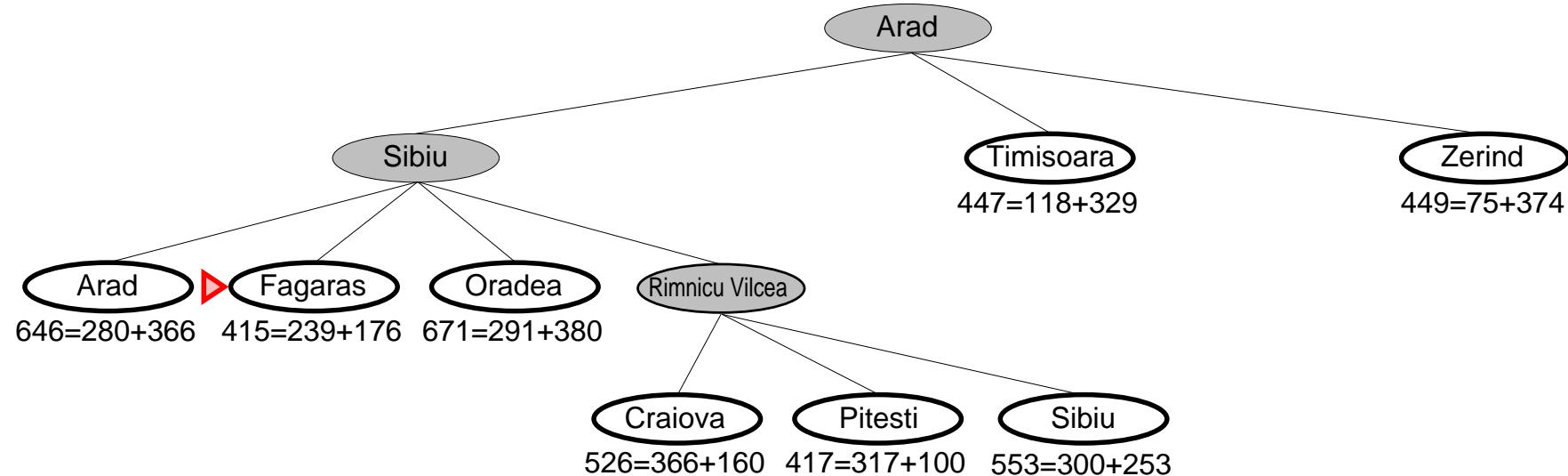
A* search example



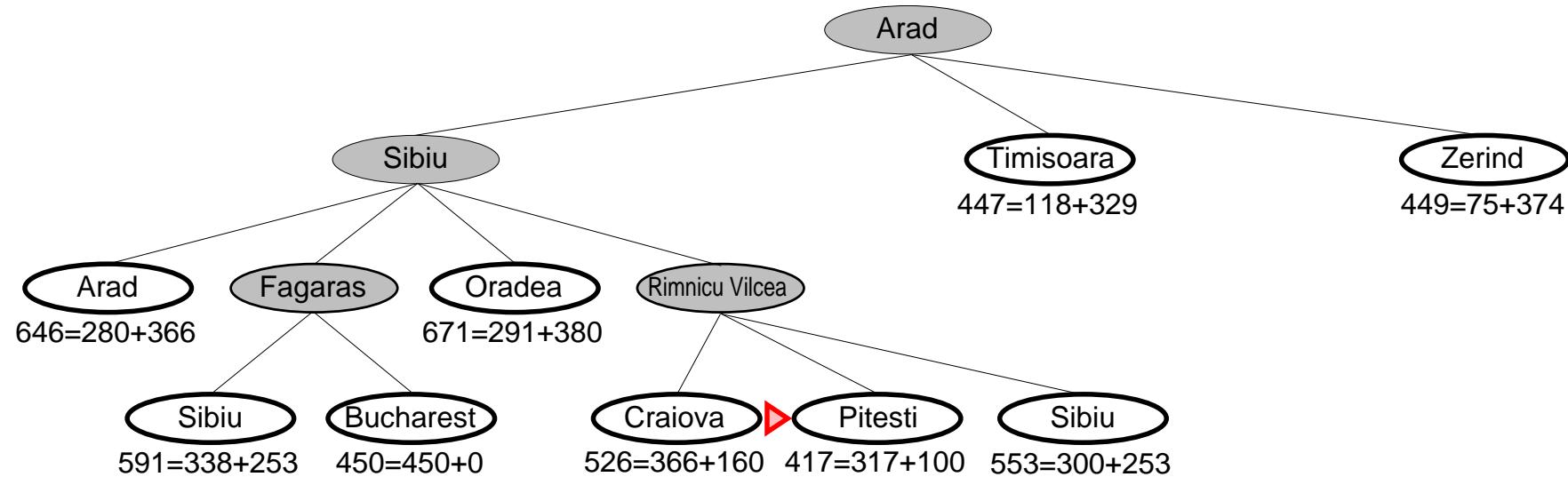
A* search example



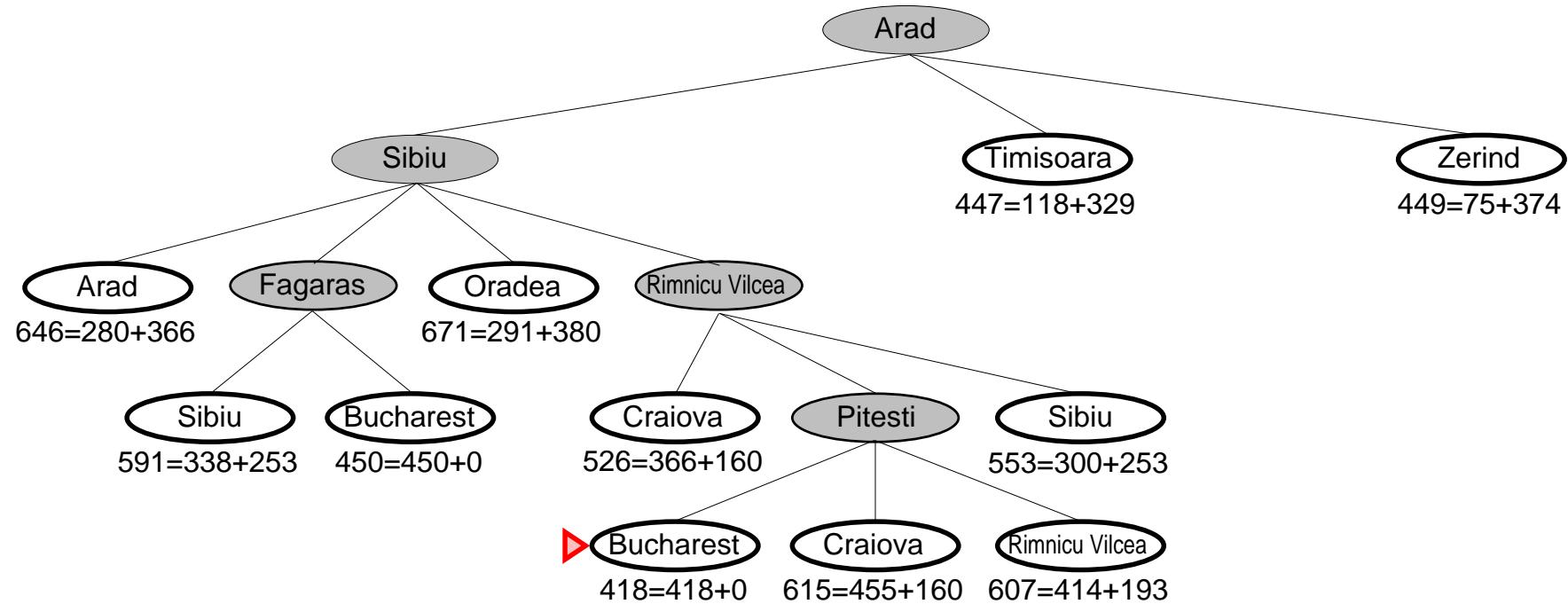
A* search example



A* search example

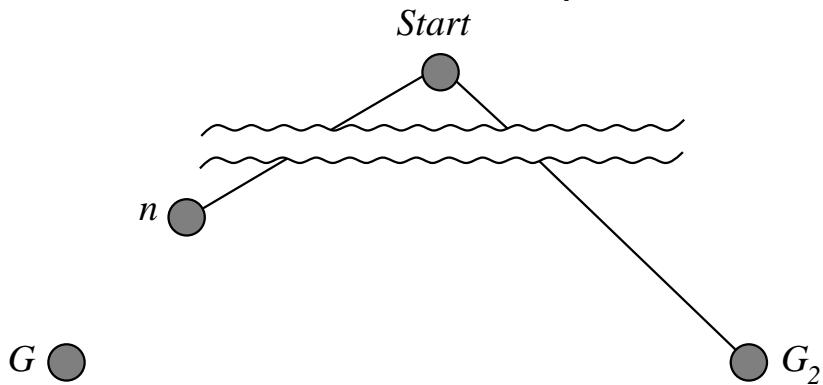


A* search example



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 .



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

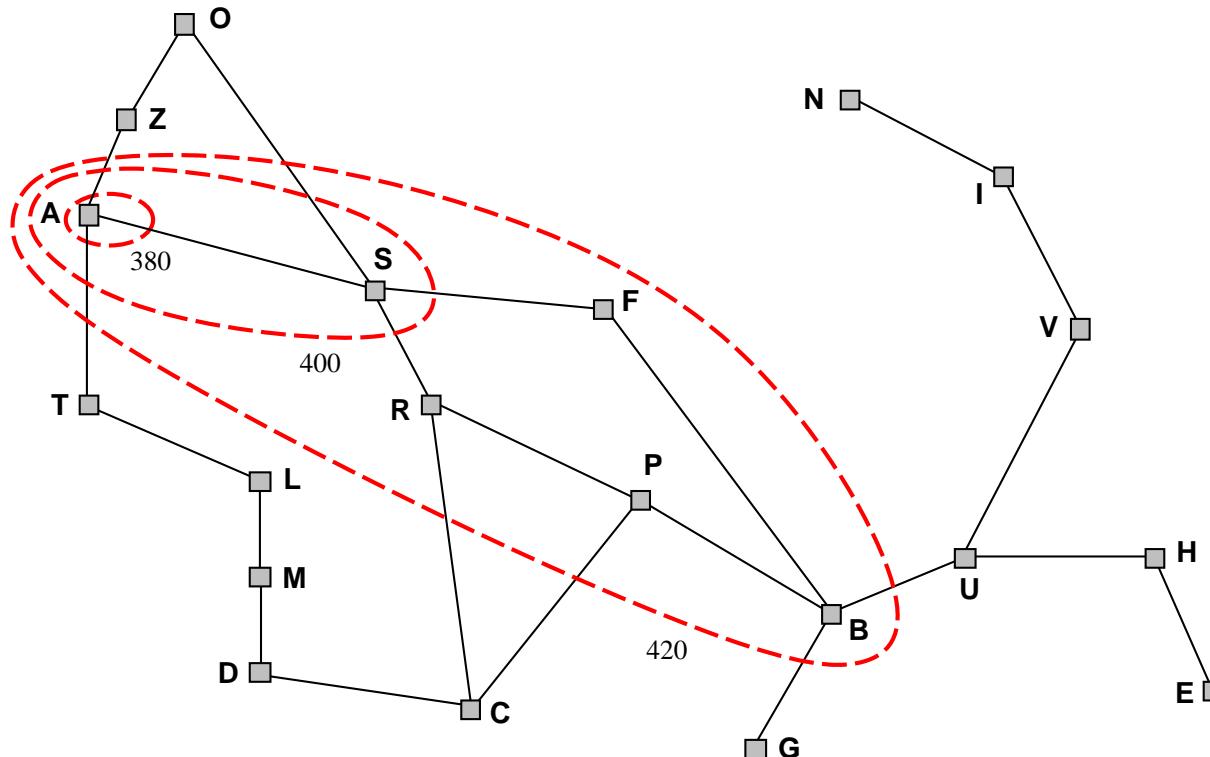
Since $f(G_2) > f(n)$, A* will never select G_2 for expansion

Optimality of A* (more useful)

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 $f_i < f_{i+1}$



Properties of A^*

Complete??

Properties of A*

Complete?? Yes, unless there are infinitely many nodes with $f \leq f(G)$

Time??

Properties of \mathbf{A}^*

Complete?? Yes, unless there are infinitely many nodes with $f \leq f(G)$

Time?? Exponential in [relative error in $h \times$ length of soln.]

Space??

Properties of \mathbf{A}^*

Complete?? Yes, unless there are infinitely many nodes with $f \leq f(G)$

Time?? Exponential in [relative error in $h \times$ length of soln.]

Space?? Keeps all nodes in memory

Optimal??

Properties of A*

Complete?? Yes, unless there are infinitely many nodes with $f \leq f(G)$

Time?? Exponential in [relative error in $h \times$ length of soln.]

Space?? Keeps all nodes in memory

Optimal?? Yes—cannot expand f_{i+1} until f_i is finished

A* expands all nodes with $f(n) < C^*$

A* expands some nodes with $f(n) = C^*$

A* expands no nodes with $f(n) > C^*$

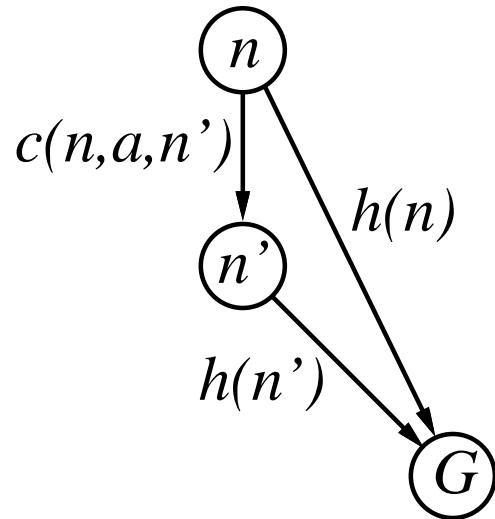
Proof of lemma: Consistency

A heuristic is **consistent** if

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

If h is consistent, we have

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



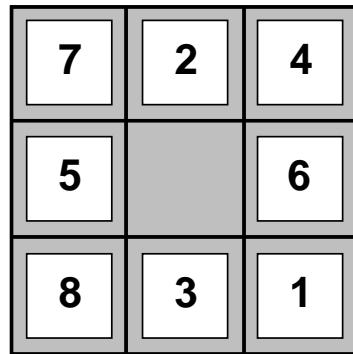
I.e., $f(n)$ is nondecreasing along any path.

Admissible heuristics

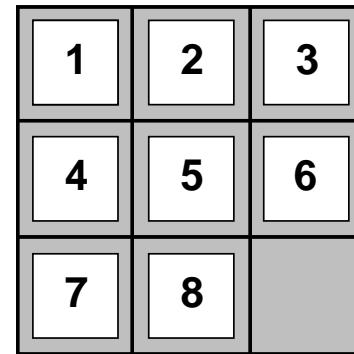
E.g., for the 8-puzzle:

$h_1(n)$ = 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

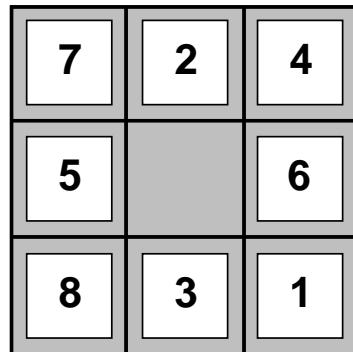
$$\begin{aligned} h_1(S) &= ?? \\ \underline{h_2(S)} &= ?? \end{aligned}$$

Admissible heuristics

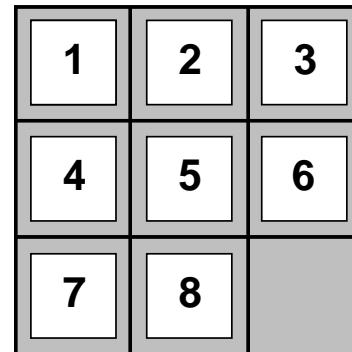
E.g., for the 8-puzzle:

$h_1(n)$ = 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

$$h_1(S) = ?? \ 6$$

$$\underline{h_2(S) = ??} \ 4+0+3+3+1+0+2+1 = 14$$

Dominance

If $h_2(n) \geq h_1(n)$ for all n (both admissible)
then h_2 dominates h_1 and is better for search

Typical search costs:

$d = 14$ IDS = 3,473,941 nodes

$A^*(h_1)$ = 539 nodes

$A^*(h_2)$ = 113 nodes

$d = 24$ IDS \approx 54,000,000,000 nodes

$A^*(h_1)$ = 39,135 nodes

$A^*(h_2)$ = 1,641 nodes

Given any admissible heuristics h_a, h_b ,

$$h(n) = \max(h_a(n), h_b(n))$$

is also admissible and dominates h_a, h_b

Relaxed problems

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

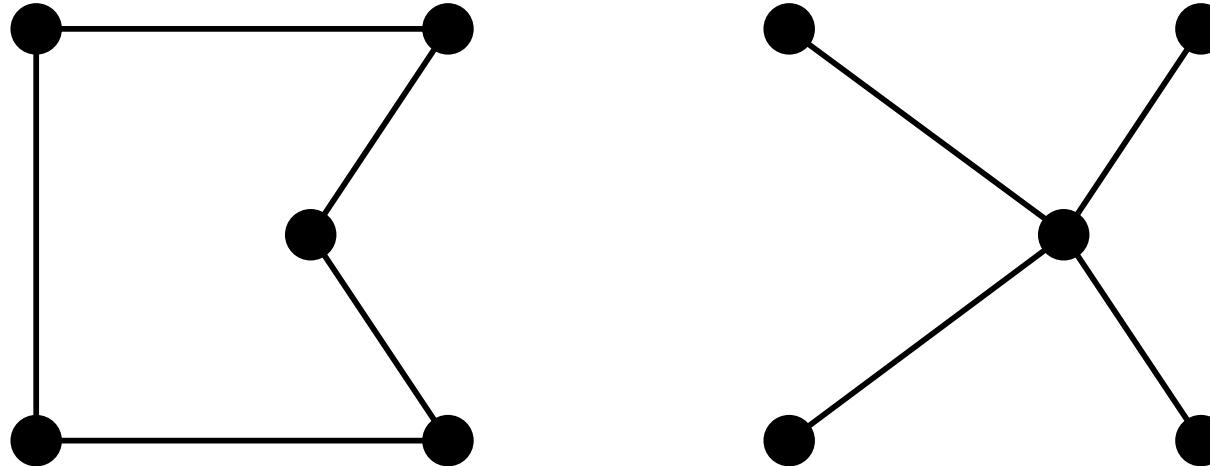
If the rules are relaxed so that a tile can move to **any adjacent square**, then $h_2(n)$ gives the shortest solution

Key point: the optimal solution cost of a relaxed problem is no greater than the optimal solution cost of the real problem

Relaxed problems contd.

Well-known example: travelling salesperson problem (TSP)

Find the shortest tour visiting all cities exactly once



Minimum spanning tree can be computed in $\mathcal{O}(n^2)$
and is a lower bound on the shortest (open) tour

Summary

Heuristic functions estimate costs of shortest paths

Good heuristics can dramatically reduce search cost

Greedy best-first search expands lowest h

- incomplete and not always optimal

A* search expands lowest $g + h$

- complete and optimal
- also optimally efficient (up to tie-breaks, for forward search)

Admissible heuristics can be derived from exact solution of relaxed problems

A* Search

- The most common informed search algorithm is A* search (pronounced “A-star search”), a best-first search that uses the evaluation function,

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

where $g(n)$ is the path cost from the initial state to node n , and $h(n)$ is the estimated cost of the shortest path from n to a goal state, so we have,

f(n) = estimated cost of the best path that continues from n to a goal.

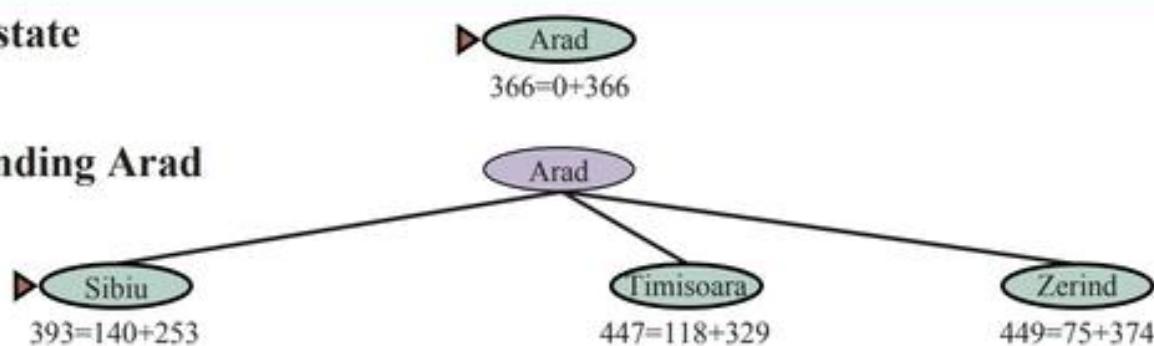
Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

Figure 3.16 Values of h_{SLD} —straight-line distances to Bucharest.

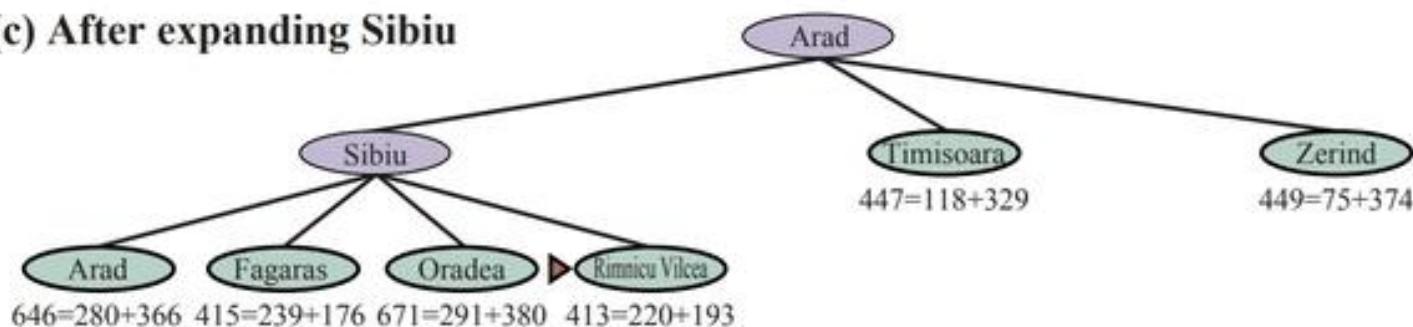
(a) The initial state



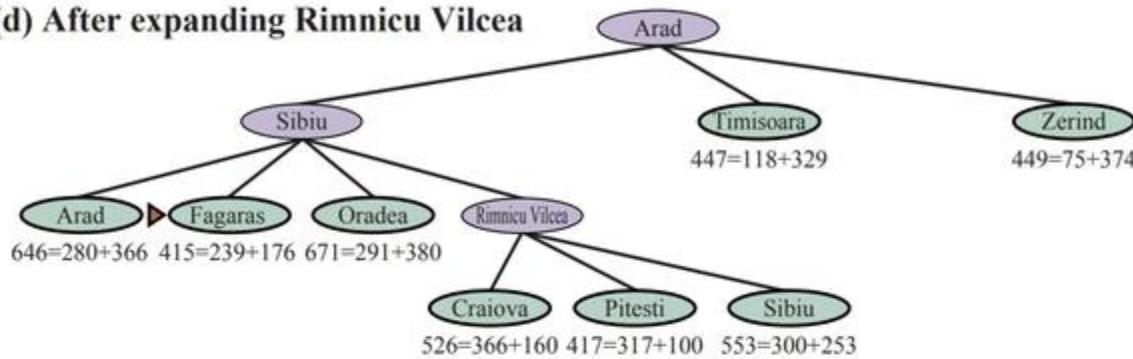
(b) After expanding Arad



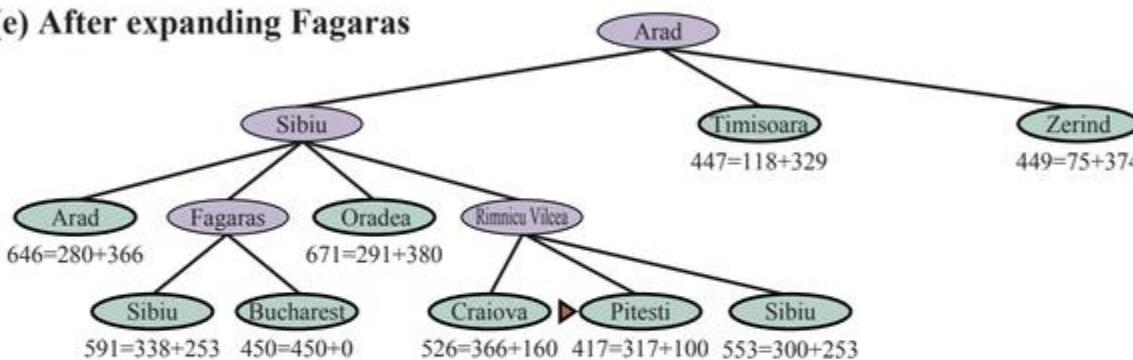
(c) After expanding Sibiu



(d) After expanding Rimnicu Vilcea



(e) After expanding Fagaras



(f) After expanding Pitesti

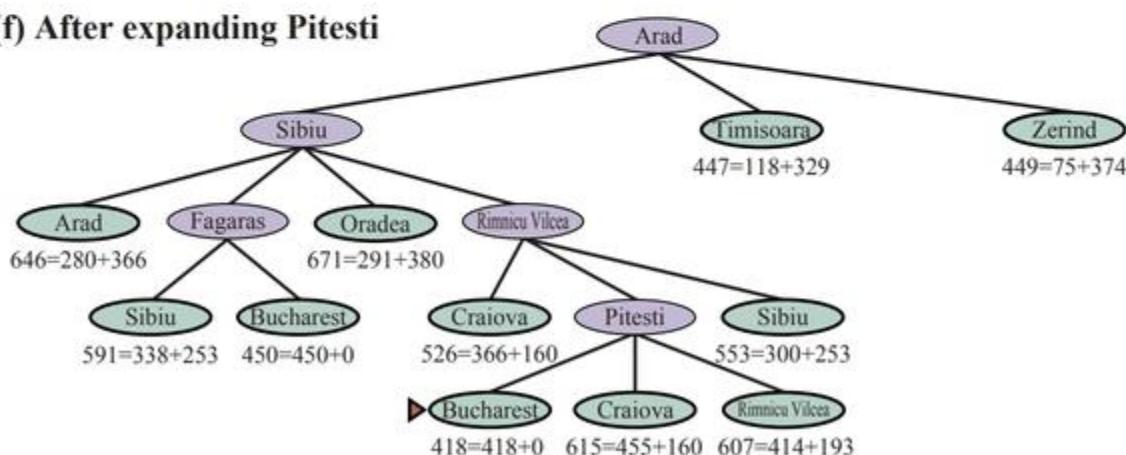


Figure 3.18 Stages in an A* search for Bucharest. Nodes are labeled with $f = g + h$. The h values are the straight-line distances to Bucharest taken from Figure 3.16.

A* search is complete.!!

- *Whether A* is cost-optimal depends on certain properties of the heuristic.*
- A key property is admissibility an admissible heuristic is one that never overestimates the cost to reach a goal. (An admissible heuristic is therefore optimistic.)
- With an admissible heuristic, A* is cost-optimal.
- Suppose the optimal path has cost C^* , but the algorithm returns a path with cost $C > C^*$. **Then there must be some node n which is on the optimal path and is unexpanded.**

- The notation $g^*(n)$ denote the cost of the optimal path from the start to n , and $h^*(n)$ denote the cost of the optimal path from n to the nearest goal, we have:

$$f(n) > C^* \text{ (otherwise } n \text{ would have been expanded)}$$

$$f(n) = g(n) + h(n) \text{ (by definition)}$$

$$f(n) = g^*(n) + h(n) \text{ (because } n \text{ is on an optimal path)}$$

$$f(n) \leq g^*(n) + h^*(n) \text{ (because of admissibility, } h(n) \leq h^*(n))$$

$$f(n) \leq C^* \text{ (by definition, } C^* = g^*(n) + h^*(n))$$

- a suboptimal path must be wrong—it must be that A* returns only cost-optimal paths.

- A heuristic $i(n)$ is consistent if, for every node n and every successor n' of n generated by an action a , we have:

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

- This is a form of the triangle inequality, which stipulates that a side of a triangle cannot be longer than the sum of the other two sides.

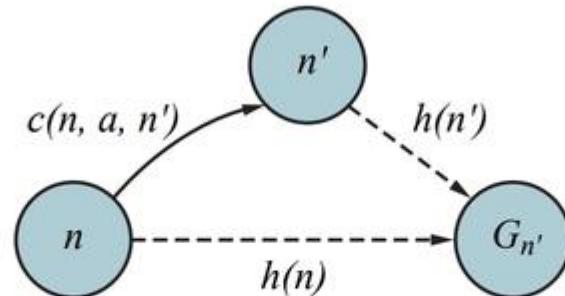


Figure 3.19 Triangle inequality: If the heuristic h is **consistent**, then the single number $h(n)$ will be less than the sum of the cost $c(n, a, n')$ of the action from n to n' plus the heuristic estimate $h(n')$.

Search contours

- A useful way to visualize a search is to draw contours in the state space, just like the contours in a topographic map.

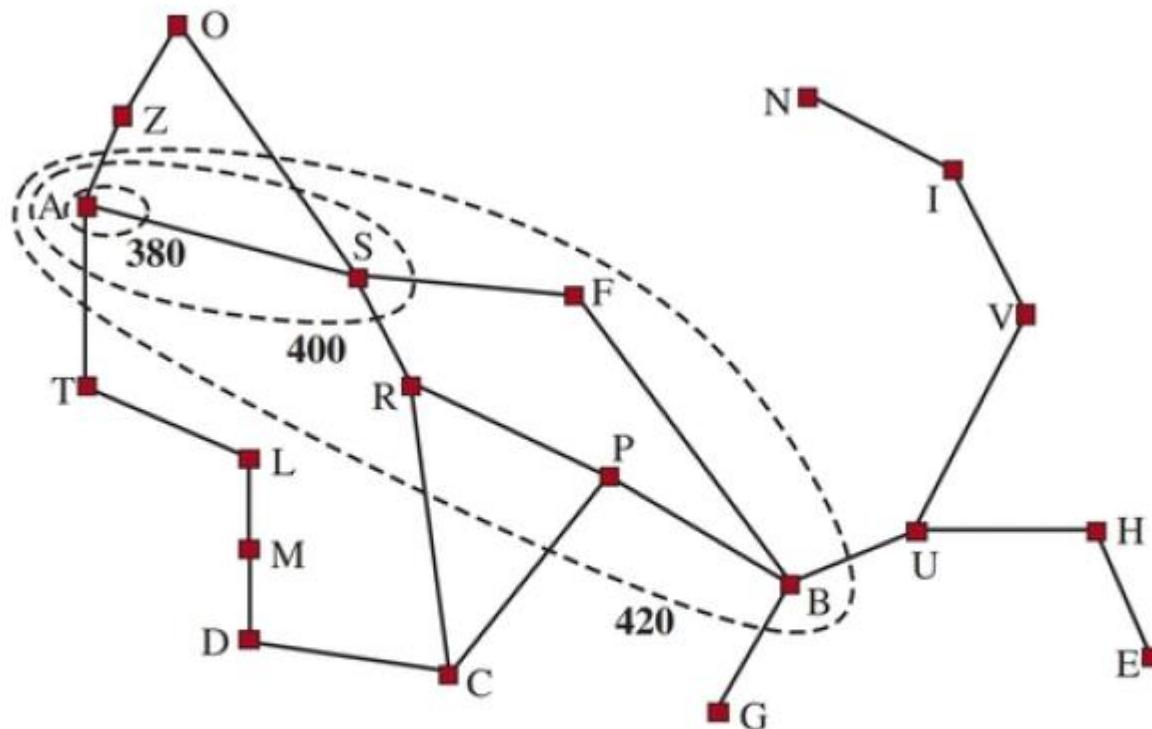


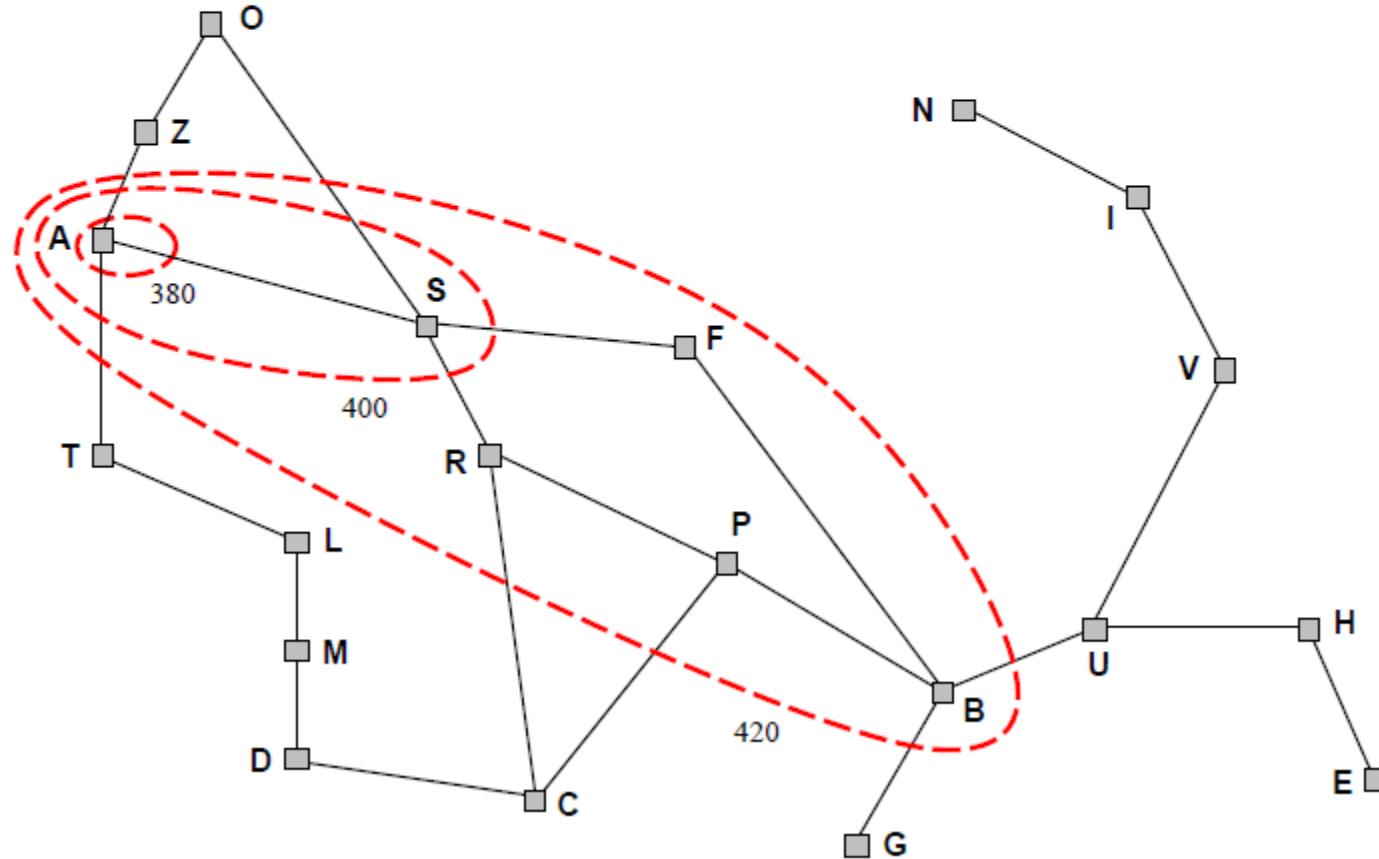
Figure 3.20 Map of Romania showing contours at $f = 380$, $f = 400$, and $f = 420$, with Arad as the start state. Nodes inside a given contour have $f = g + h$ costs less than or equal to the contour value.

Optimality of A* (more useful)

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 $f_i < f_{i+1}$



Properties of A*

Complete?? Yes, unless there are infinitely many nodes with $f \leq f(G)$

Time?? Exponential in [relative error in $h \times$ length of soln.]

Space?? Keeps all nodes in memory

Optimal?? Yes—cannot expand f_{i+1} until f_i is finished

A* expands all nodes with $f(n) < C^*$

A* expands some nodes with $f(n) = C^*$

A* expands no nodes with $f(n) > C^*$

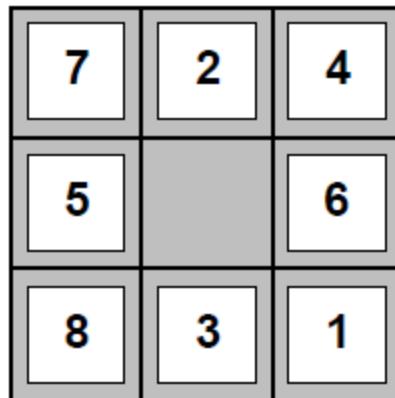
Admissible heuristics

E.g., for the 8-puzzle:

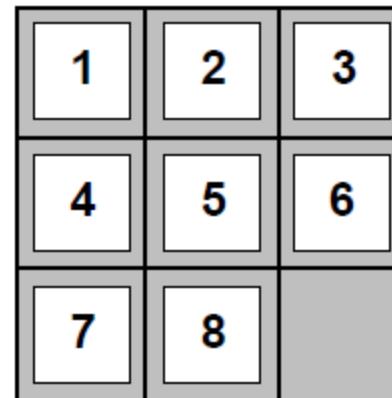
$h_1(n)$ = number of misplaced tiles

$h_2(n)$ = total Manhattan distance

(i.e., no. of squares from desired location of each tile)



Start State



Goal State

$$h_1(S) = ??$$

$$h_2(S) = ??$$

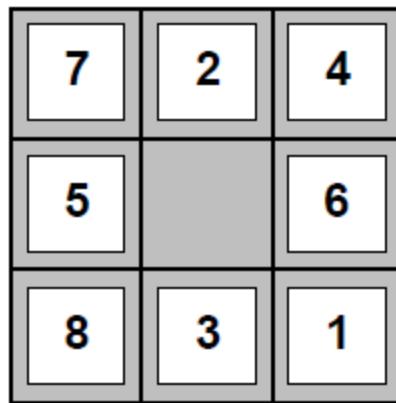
Admissible heuristics

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

$$h_1(S) = ?? \quad 6$$

$$h_2(S) = ?? \quad 4+0+3+3+1+0+2+1 = 14$$

Dominance

If $h_2(n) \geq h_1(n)$ for all n (both admissible)
then h_2 dominates h_1 and is better for search

Typical search costs:

$d = 14$ IDS = 3,473,941 nodes

$A^*(h_1)$ = 539 nodes

$A^*(h_2)$ = 113 nodes

$d = 24$ IDS \approx 54,000,000,000 nodes

$A^*(h_1)$ = 39,135 nodes

$A^*(h_2)$ = 1,641 nodes

Given any admissible heuristics h_a , h_b ,

$$h(n) = \max(h_a(n), h_b(n))$$

is also admissible and dominates h_a , h_b

Relaxed problems

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

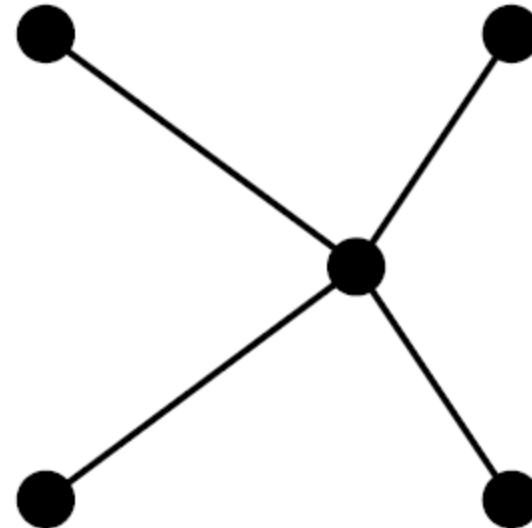
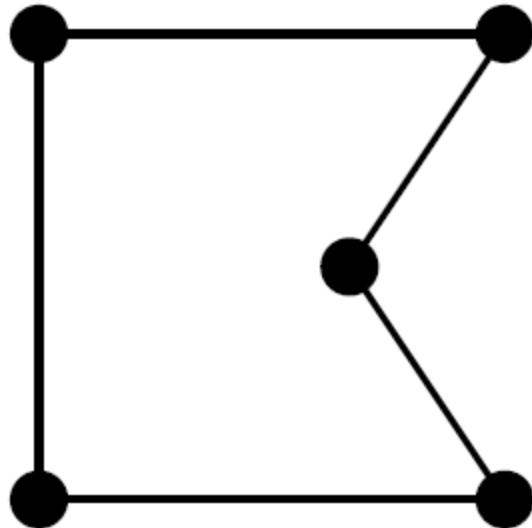
If the rules are relaxed so that a tile can move to **any adjacent square**, then $h_2(n)$ gives the shortest solution

Key point: the optimal solution cost of a relaxed problem is no greater than the optimal solution cost of the real problem

- A problem with fewer restrictions on the actions is called a relaxed problem.
- The state-space graph of the relaxed problem is a supergraph of the original state space because the removal of restrictions creates added edges in the graph.
- *Hence, the cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem.*

Relaxed problems contd.

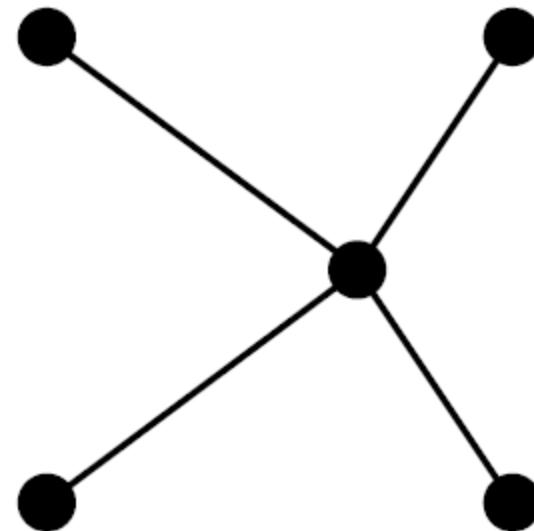
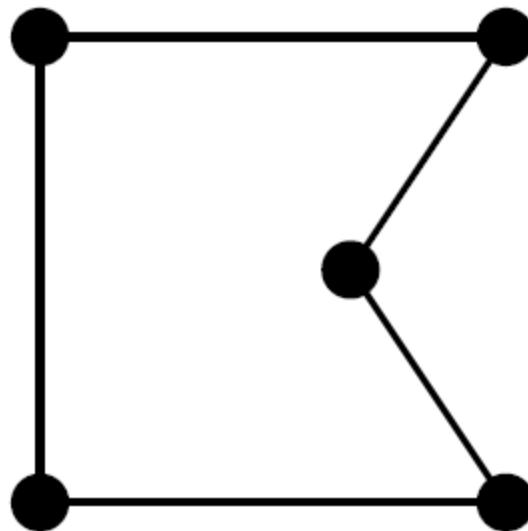
Well-known example: travelling salesperson problem (TSP)
Find the shortest tour visiting all cities exactly once



Relaxed problems contd.

Well-known example: travelling salesperson problem (TSP)

Find the shortest tour visiting all cities exactly once



Minimum spanning tree can be computed in $O(n^2)$
and is a lower bound on the shortest (open) tour

Summary

Heuristic functions estimate costs of shortest paths

Good heuristics can dramatically reduce search cost

Greedy best-first search expands lowest h

- incomplete and not always optimal

A^* search expands lowest $g + h$

- complete and optimal
- also optimally efficient (up to tie-breaks, for forward search)

Admissible heuristics can be derived from exact solution of relaxed problems

Local Search Algorithms

Local Search Algorithms

- Local search algorithms operate by searching from a start state to neighboring states, without keeping track of the paths, nor the set of states that have been reached.
- That means they are not systematic—they might never explore a portion of the search space where a solution actually resides.
- However, they have two key advantages:
 - (1) they use very little memory; and
 - (2) they can often find reasonable solutions in large or infinite state spaces for which systematic algorithms are unsuitable.
- Local search algorithms can also solve **optimization problems**, in which the aim is to find the best state according to an **objective function**.

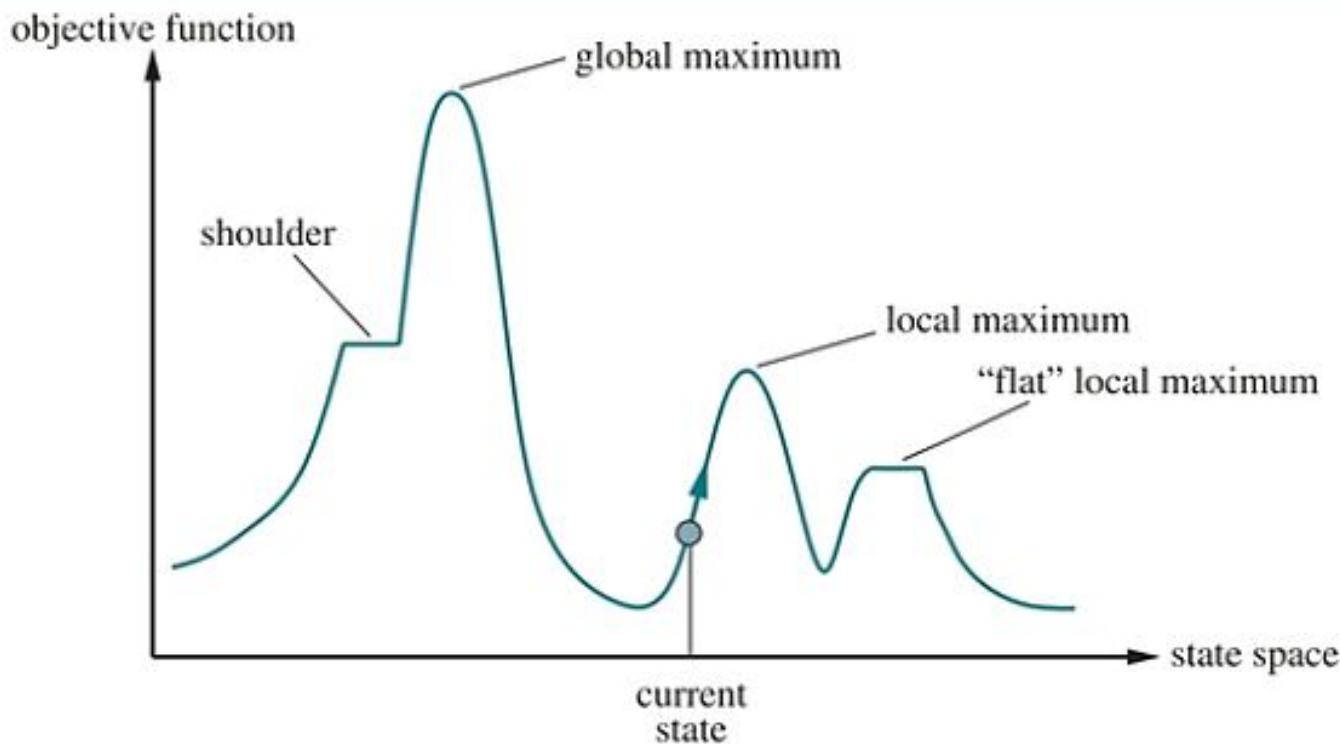


Figure 4.1 A one-dimensional state-space landscape in which elevation corresponds to the objective function. The aim is to find the global maximum.

- Each point (state) in the landscape has an “elevation,” defined by the value of the objective function.
- If elevation corresponds to an objective function then the aim is to find the highest peak—a ***global maximum***—and we call the process **hill climbing**.
- If elevation corresponds to cost, then the aim is to find the lowest valley—a ***global minimum***—and we call it **gradient descent**.

Hill-climbing search

- It keeps track of one current state and on each iteration moves to the neighbouring state with highest value
 - that is, it heads in the direction that provides the steepest ascent.
- It terminates when it reaches a “peak” where no neighbour has a higher value.
- Hill climbing does not look ahead beyond the immediate neighbours of the current state.

```
function HILL-CLIMBING(problem) returns a state that is a local maximum  
    current  $\leftarrow$  problem.INITIAL  
    while true do  
        neighbor  $\leftarrow$  a highest-valued successor state of current  
        if VALUE(neighbor)  $\leq$  VALUE(current) then return current  
        current  $\leftarrow$  neighbor
```

Figure 4.2 The hill-climbing search algorithm, which is the most basic local search technique. At each step the current node is replaced by the best neighbor.

Hill Climbing: Example (Minimizing h)

start

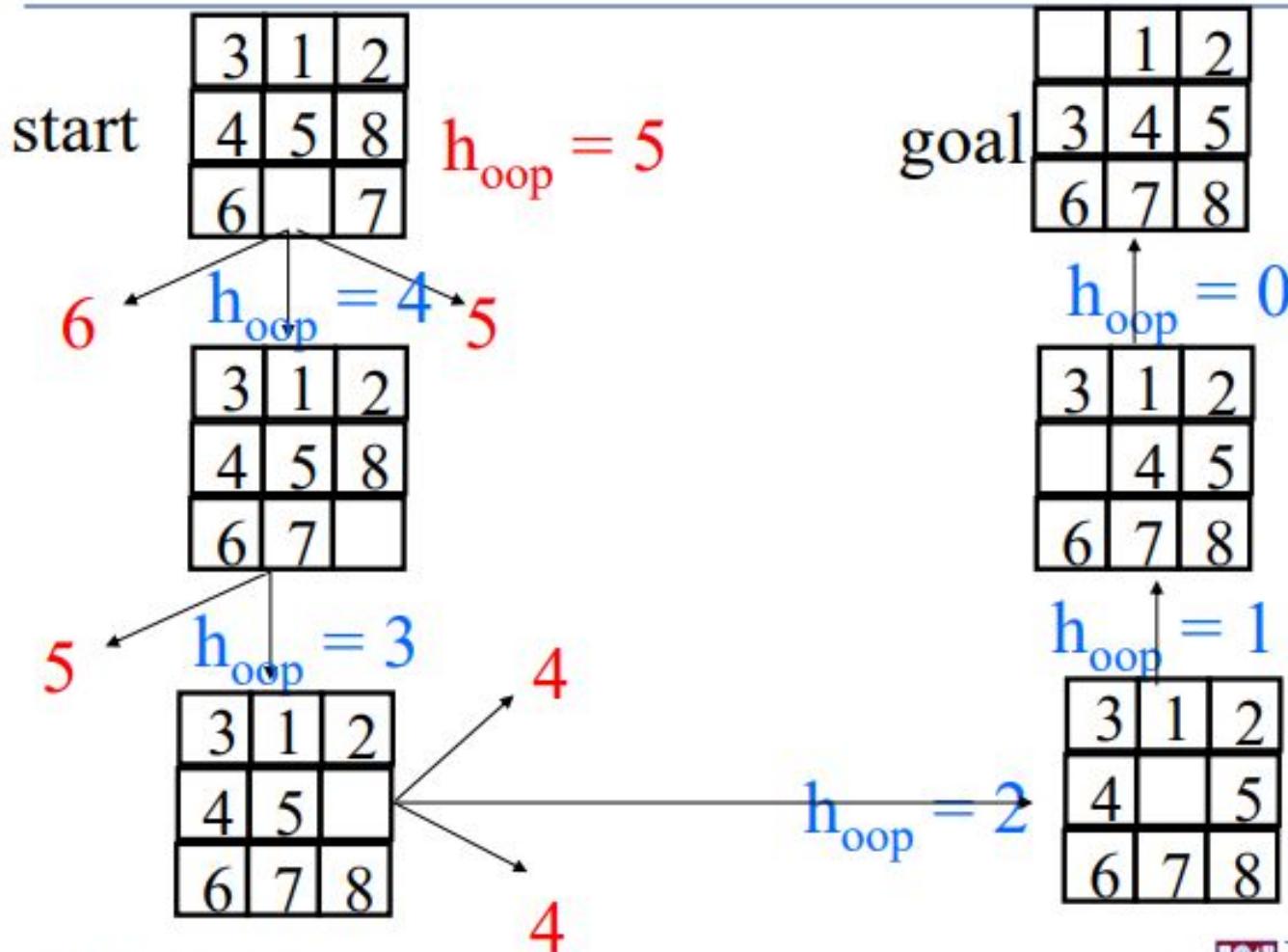
3	1	2
4	5	8
6		7

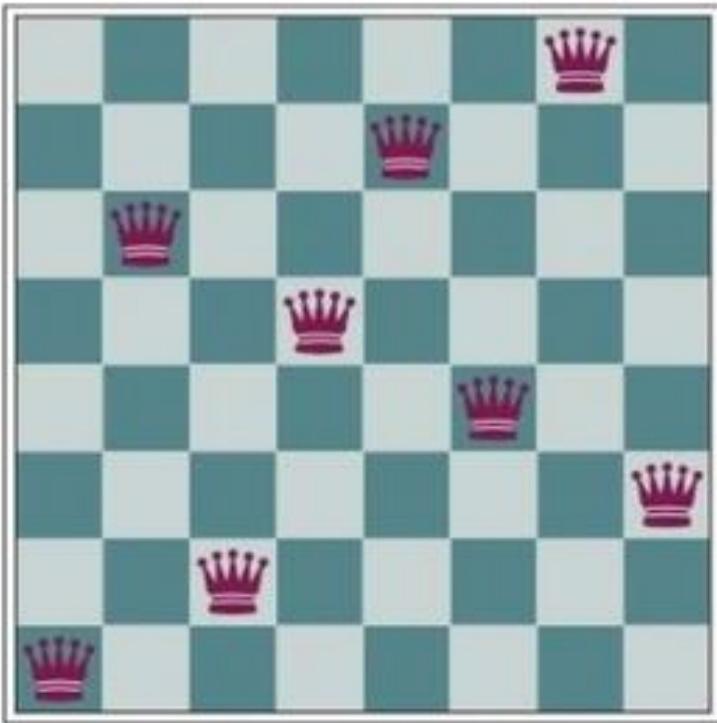
$h_{oop} = 5$

goal

	1	2
3	4	5
6	7	8

Hill Climbing: Example (Minimizing h)





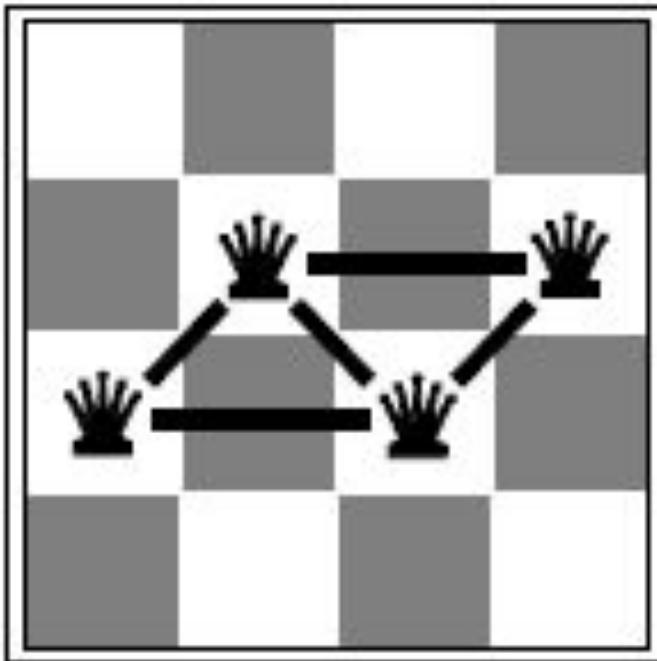
(a)

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

(b)

Figure 4.3 (a) The 8-queens problem: place 8 queens on a chess board so that no queen attacks another. (A queen attacks any piece in the same row, column, or diagonal.) This position is almost a solution, except for the two queens in the fourth and seventh columns that attack each other along the diagonal. (b) An 8-queens state with heuristic cost estimate $h=17$. The board shows the value of h for each possible successor obtained by moving a queen within its column. There are 8 moves that are tied for best, with $h=12$. The hill-climbing algorithm will pick one of these.

Reduce the number of conflicts



h = 5

- Hill climbing is sometimes called greedy local search because it grabs a good neighbor state without thinking ahead about where to go next.
- Hill climbing can get stuck for any of the following reasons:
 - Local maxima: A local maximum is a peak that is higher than each of its neighboring states but lower than the global maximum.
 - Ridges: Ridges result in a sequence of local maxima that is very difficult for greedy algorithms to navigate.
 - Plateaus: A plateau is a flat area of the state-space landscape

Types of Hill Climbing Algorithms

- **Stochastic hill climbing:** chooses at random from among the uphill moves; the probability of selection can vary with the steepness of the uphill move.
- **First-choice hill climbing:** implements stochastic hill climbing by generating successors randomly until one is generated that is better than the current state.
- **Random-restart hill climbing:** It conducts a series of hill-climbing searches from randomly generated initial states, until a goal is found.

- A hill-climbing algorithm that never makes “downhill” moves toward states with lower value (or higher cost) is always vulnerable to getting stuck in a local maximum.
- In contrast, a purely random walk that moves to a successor state without concern for the value will eventually stumble upon the global maximum, but will be extremely inefficient.
- Combination of hill climbing with a random walk in a way that yields both efficiency and completeness.

Simulated annealing

- In metallurgy, annealing is the process used to temper or harden metals and glass by heating them to a high temperature and then gradually cooling them, thus allowing the material to reach a low-energy crystalline state.
- Simulated-annealing solution is to start by shaking hard (i.e., at a high temperature) and then gradually reduce the intensity of the shaking (i.e., lower the temperature).
- Instead of picking the best move, however, it picks a random move. If the move improves the situation, it is always accepted.
- Otherwise, the algorithm accepts the move with some probability less than 1.

- Basic idea:
 - Allow “bad” moves occasionally, depending on “temperature”
 - High temperature => more bad moves allowed, shake the system out of its local minimum
 - Gradually reduce temperature according to some schedule
 - Sounds pretty flaky, doesn’t it?

Simulated annealing algorithm

```
function SIMULATED-ANNEALING(problem,schedule) returns a state
    current ← problem.initial-state
    for t = 1 to ∞ do
        T ← schedule(t)
        if T = 0 then return current
        next ← a randomly selected successor of current
        ΔE ← next.value – current.value
        if ΔE > 0 then current ← next
        else current ← next only with probability  $e^{\Delta E/T}$ 
```

Simulated Annealing

- The probability decreases exponentially with the “badness” b of the move—the amount ΔE by which the evaluation is worsened.
- The probability also decreases as the “temperature” T goes down: “bad” moves are more likely to be allowed at the start when T is high, and they become more unlikely as T decreases.
- If the schedule lowers T to 0 slowly enough, then a property of the Boltzmann distribution,
$$e^{\Delta E/T}$$
- Is that all the probability is concentrated on the global maxima, which the algorithm will find with probability approaching 1.

Simulated Annealing

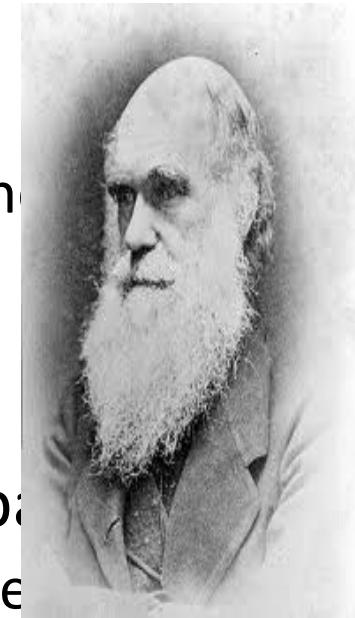
- Is this convergence an interesting guarantee?
- Sounds like magic, but reality is reality:
 - The more downhill steps you need to escape a local optimum, the less likely you are to ever make them all in a row
 - “Slowly enough” may mean exponentially slowly
 - Random restart hillclimbing also converges to optimal state...
- Simulated annealing and its relatives are a key workhorse in VLSI layout and other optimal configuration problems

```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
  current  $\leftarrow$  problem.INITIAL
  for t = 1 to  $\infty$  do
    T  $\leftarrow$  schedule(t)
    if T = 0 then return current
    next  $\leftarrow$  a randomly selected successor of current
     $\Delta E \leftarrow \text{VALUE}(\textit{current}) - \text{VALUE}(\textit{next})$ 
    if  $\Delta E > 0$  then current  $\leftarrow$  next
    else current  $\leftarrow$  next only with probability  $e^{-\Delta E/T}$ 
```

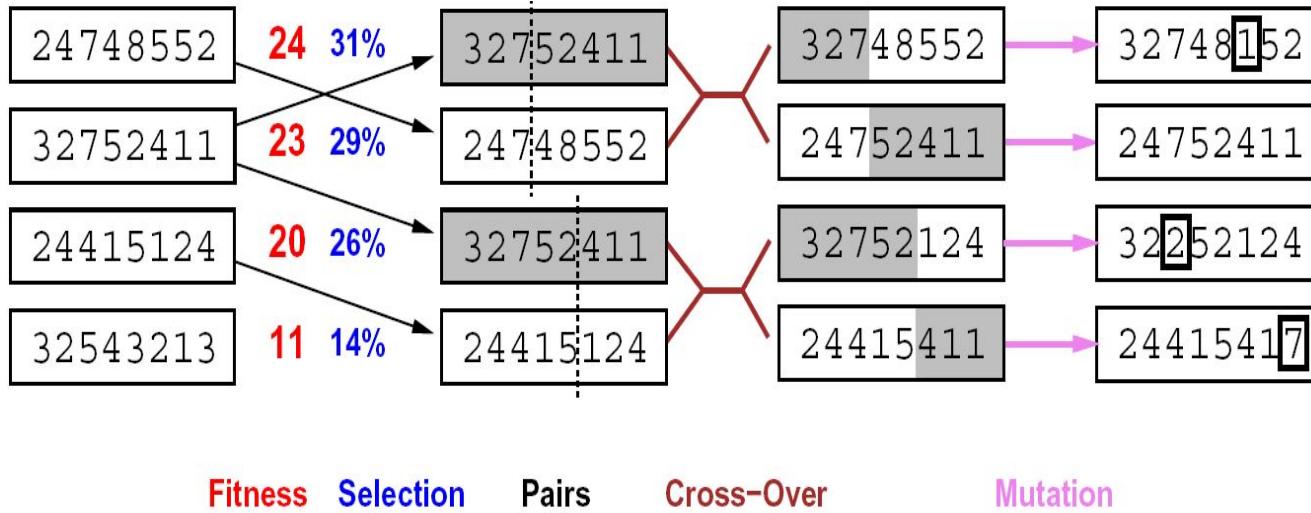
Figure 4.5 The simulated annealing algorithm, a version of stochastic hill climbing where some downhill moves are allowed. The *schedule* input determines the value of the “temperature” *T* as a function of time.

Local beam search

- Basic idea:
 - K copies of a local search algorithm, initialized randomly
 - For each iteration
 - Generate ALL successors from K current states
 - Choose best K of these to be the new current states
- Why is this different from K local searches in parallel?
 - The searches **communicate**! “Come over here, there’s greener!”
- What other well-known algorithm does this remind you of?
 - Evolution!

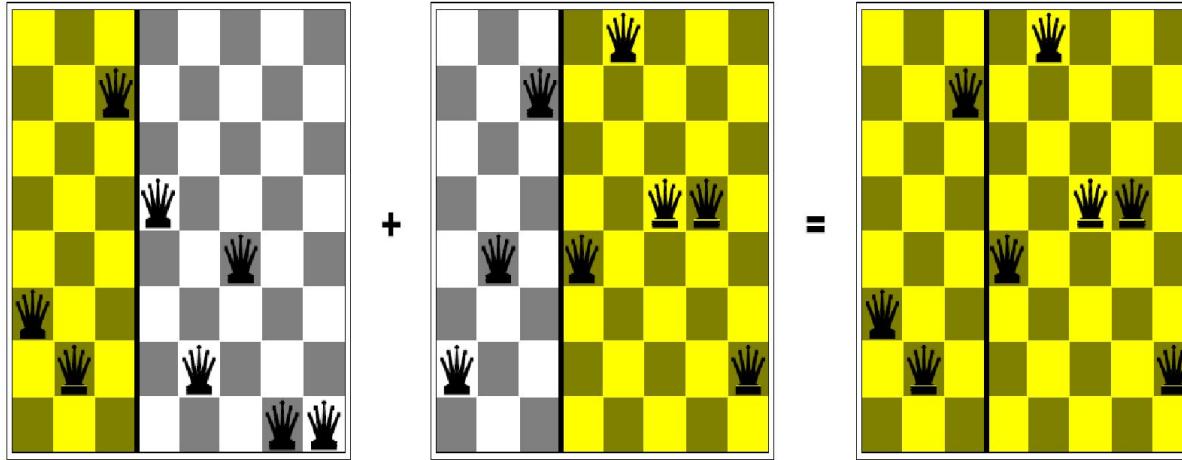


Genetic algorithms



- Genetic algorithms use a natural selection metaphor
 - Resample K individuals at each step (selection) weighted by fitness function
 - Combine by pairwise crossover operators, plus mutation to give variety

Example: N-Queens



- Does crossover make sense here?
- What would mutation be?
- What would a good fitness function be?

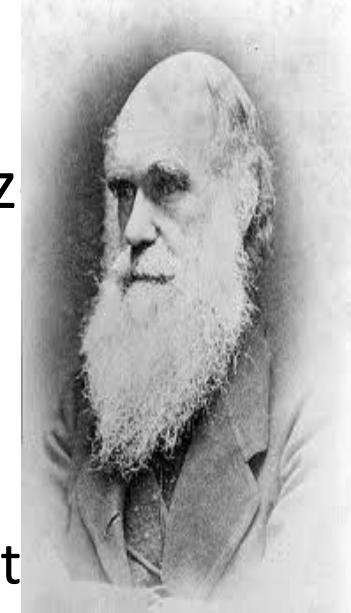
Summary

- Many configuration and optimization problems can be formulated as local search
- General families of algorithms:
 - Hill-climbing, continuous optimization
 - Simulated annealing (and other stochastic methods)
 - Local beam search: multiple interaction searches
 - Genetic algorithms: break and recombine states

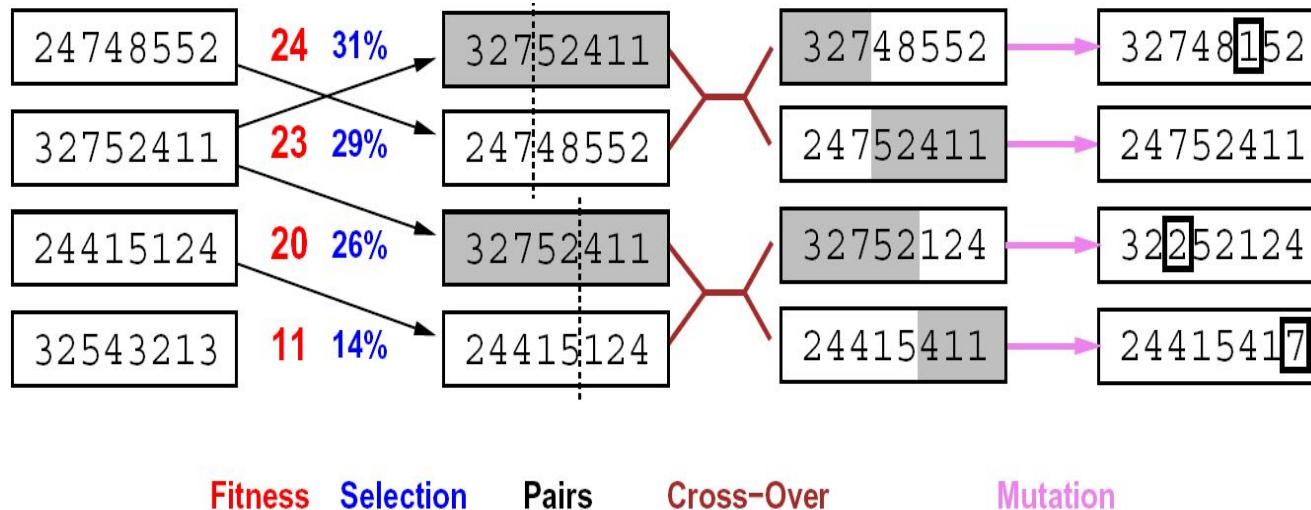
Many machine learning algorithms are local searches

Local beam search

- Basic idea:
 - K copies of a local search algorithm, initialized randomly
 - For each iteration
 - Generate ALL successors from K current states
 - Choose **best K** of these to be the new current state
- Or, K chosen randomly with a bias towards good ones
- Why is this different from K local searches in parallel?
 - The searches **communicate**! “Come over here, the grass is greener!”
- What other well-known algorithm does this

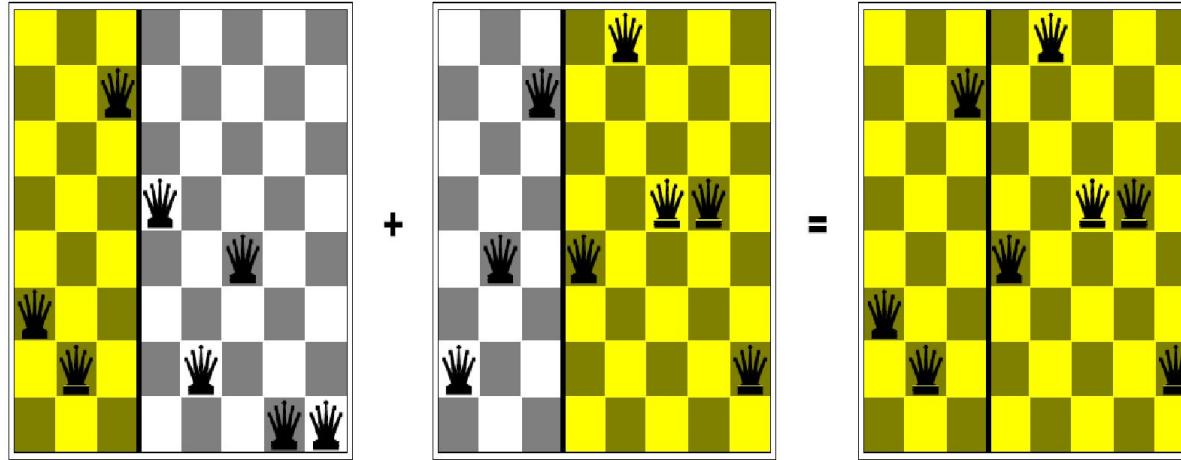


Genetic algorithms



- Genetic algorithms use a natural selection metaphor
 - Resample K individuals at each step (selection) weighted by fitness function
 - Combine by pairwise crossover operators, plus mutation to give variety

Example: N-Queens



- Does crossover make sense here?
- What would mutation be?
- What would a good fitness function be?

Summary

- Many configuration and optimization problems can be formulated as local search
- General families of algorithms:
 - Hill-climbing, continuous optimization
 - Simulated annealing (and other stochastic methods)
 - Local beam search: multiple interaction searches
 - Genetic algorithms: break and recombine states

Genetic Algorithms

Credits:

Richard Frankel

Stanford University

And

Anas S. To'meh

Outline

- Motivations/applicable situations
- What are genetic algorithms?
- Example
- Pros and cons

Motivation

- Searching some search spaces with traditional search methods would be intractable. This is often true when states/candidate solutions have a large number of successors.
 - Example: Designing the surface of an aircraft.

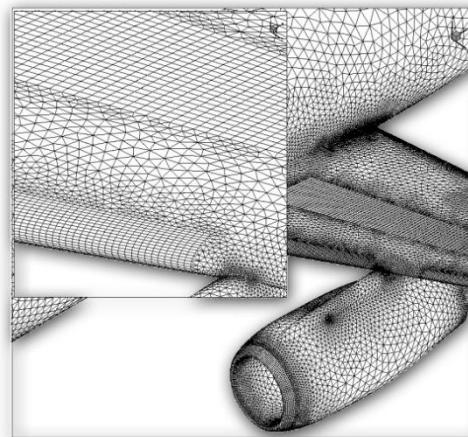
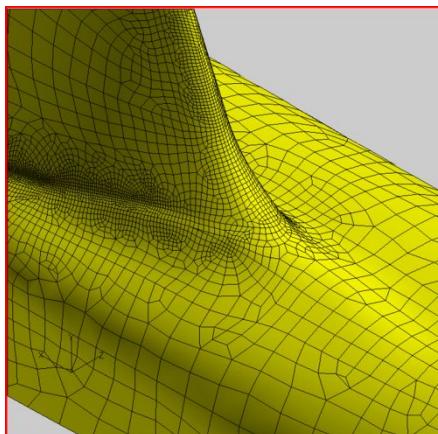


Image source: <http://www.centaursoft.com/cms/index.php?section=25>

Applicable situations

- Often used for optimization (scheduling, design, etc.) problems, though can be used for many other things as well, as we'll see a bit later.
 - Good problem for GA: Scheduling air traffic
 - Bad problems for GA: Finding large primes (why?)

Applicable situations

- Genetic algorithms work best when the “fitness landscape” is continuous (in some dimensions). This is also true of standard search, e.g. A*.
 - Intuitively, this just means that we can find a heuristic that gives a rough idea of how close a candidate is to being a solution.

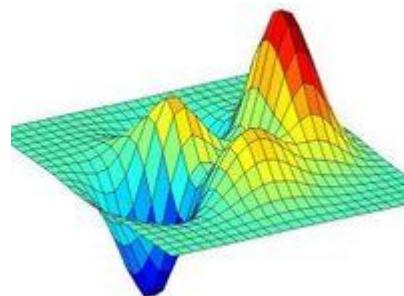


Image source: scholarpedia.org

So what *is* a genetic algorithm?

- Genetic algorithms are a randomized heuristic search strategy.
- Basic idea: Simulate natural selection, where the population is composed of *candidate solutions*.
- Focus is on evolving a population from which strong and diverse candidates can emerge via mutation and crossover (mating).

Basic algorithm

- Create an initial population, either random or “blank”.
- While the best candidate so far is not a solution:
 - Create new population using successor functions.
 - Evaluate the fitness of each candidate in the population.
- Return the best candidate found.

Simple example – alternating string

- Let's try to evolve a length 4 alternating string
- Initial population: $C1=1000$, $C2=0011$
- We roll the dice and end up creating $C1' = \text{cross } (C1, C2) = 1011$ and $C2' = \text{cross } (C1, C1) = 1000$.
- We mutate $C1'$ and the fourth bit flips, giving 1010. We mutate $C2'$ and get 1001.
- We run our solution test on each. If $C1'$ is a solution, so we return it and are done.

Basic components

- Candidate representation
 - Important to choose this well. More work here means less work on the successor functions.
- Successor function(s)
 - Mutation, crossover
- Fitness function
- Solution test
- Some parameters
 - Population size
 - Generation limit

Candidate representation

- We want to encode candidates in a way that makes mutation and crossover easy.
- The typical candidate representation is a binary string. This string can be thought of as the genetic code of a candidate – thus the term “genetic algorithm”!
 - Other representations are possible, but they make crossover and mutation harder.

Candidate representation example

- Let's say we want to represent a rule for classifying bikes as mountain bikes or hybrid, based on these attributes*:
 - Make (Bridgestone, Cannondale, Nishiki, or Gary Fisher)
 - Tire type (knobby, treads)
 - Handlebar type (straight, curved)
 - Water bottle holder (*Boolean*)
- We can encode a rule as a binary string, where each bit represents whether a value is accepted.

Make	Tires	Handlebars	Water bottle
B C N G	K T	S C	Y N

*Bikes scheme used with permission from Mark Maloof.

Candidate representation example

- The candidate will be a bit string of length 10, because we have 10 possible attribute values.
- Let's say we want a rule that will match any bike that is made by Bridgestone or Cannondale, has treaded tires, and has straight handlebars. This rule could be represented as 1100011011:

Make	Tires		Handlebars		Water bottle				
1 B	1 C	0 N	0 G	0 K	1 T	1 S	0 C	1 Y	1 N

Successor functions

- Mutation – Given a candidate, return a slightly different candidate.
- Crossover – Given two candidates, produce one that has elements of each.
- We don't always generate a successor for each candidate. Rather, we generate a successor *population* based on the candidates in the current population, weighted by fitness.

Successor functions

- If your candidate representation is just a binary string, then these are easy:
 - Mutate(c): Copy c as c' . For each bit b in c' , flip b with probability p . Return c' .
 - Cross (c_1, c_2): Create a candidate c such that $c[i] = c_1[i]$ if $i \% 2 = 0$, $c[i] = c_2[i]$ otherwise. Return c .
 - Alternatively, any other scheme such that c gets roughly equal information from c_1 and c_2 .

Fitness function

- The fitness function is analogous to a heuristic that estimates how close a candidate is to being a solution.
- In general, the fitness function should be consistent for better performance.
- However, even if it is, there are no guarantees. This is a probabilistic algorithm!

Solution test

- Given a candidate, return whether the candidate is a solution.
- Often just answers the question “does the candidate satisfy some set of constraints?”
- Optional! Sometimes you just want to do the best you can in a given number of generations.

New population generation

- How do we come up with a new population?
 - Given a population P , generate P' by performing crossover $|P|$ times, each time selecting candidates with probability proportional to their fitness.
 - Get P'' by mutating each candidate in P' .
 - Return P'' .

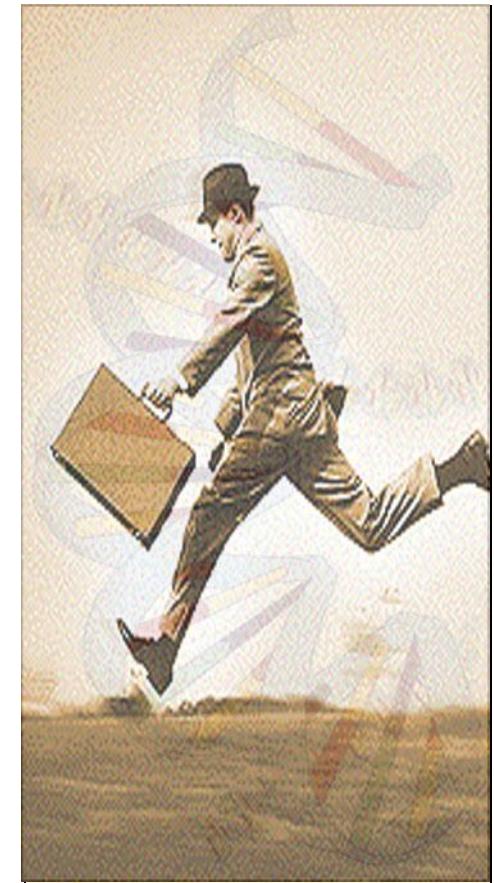
Basic algorithm (recap)

- Create an initial population, either random or “blank”.
- While the best candidate so far is not a solution:
 - Create new population using successor functions.
 - Evaluate the fitness of each candidate in the population.
- Return the best candidate found.

Knapsack Problem

Problem Description:

- You are going on a picnic.
- And have a number of items that you could take along.
- Each item has a weight and a benefit or value.
- You can take one of each item at most.
- There is a capacity limit on the weight you can carry.
- You should carry items with max. values.



Knapsack Problem

Example:

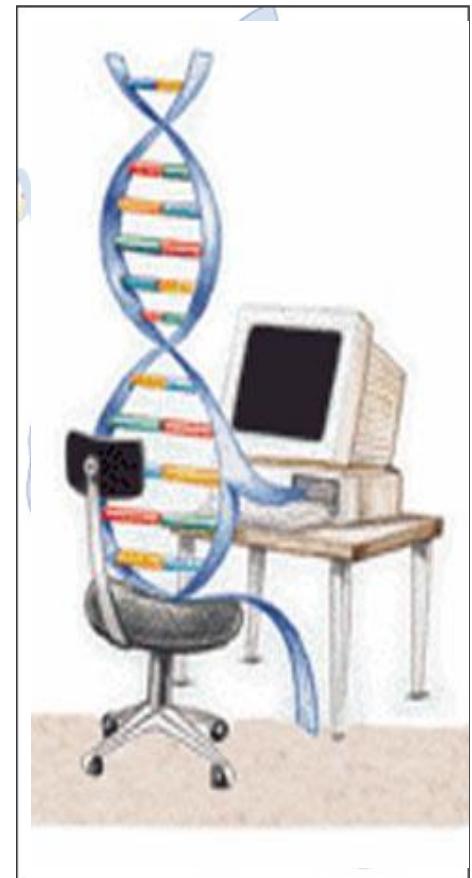
- Item: 1 2 3 4 5 6 7
- Benefit: 5 8 3 2 7 9 4
- Weight: 7 8 4 10 4 6 4
- Knapsack holds a maximum of 22 pounds
- Fill it to get the maximum benefit



Genetic Algorithm

Outline of the Basic Genetic Algorithm

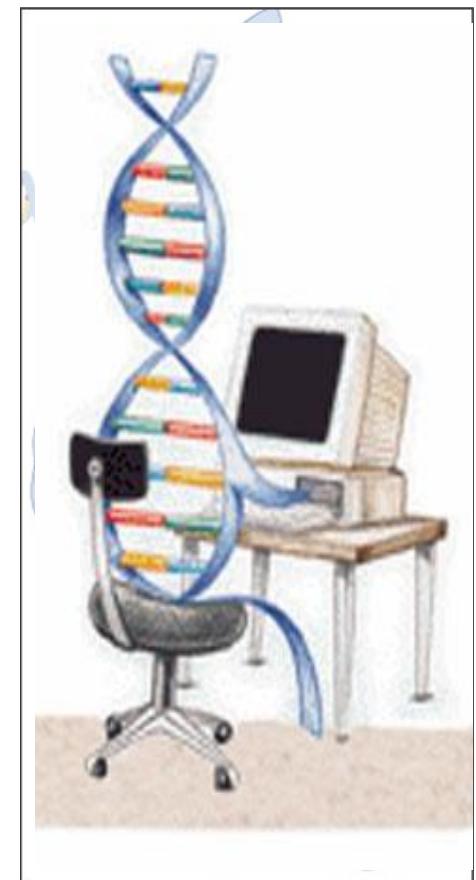
1. **[Start]**
 - ✓ Encoding: represent the individual.
 - ✓ Generate random population of n chromosomes (suitable solutions for the problem).
2. **[Fitness]** Evaluate the fitness of each chromosome.
3. **[New population]** repeating following steps until the new population is complete.
4. **[Selection]** Select the best two parents.
5. **[Crossover]** cross over the parents to form a new offspring (children).



Genetic Algorithm

Outline of the Basic Genetic Algorithm Cont.

6. **[Mutation]** With a mutation probability.
7. **[Accepting]** Place new offspring in a new population.
8. **[Replace]** Use new generated population for a further run of algorithm.
9. **[Test]** If the end condition is satisfied, then **stop**.
10. **[Loop]** Go to step 2 .



Basic Steps

Start

- Encoding: 0 = not exist, 1 = exist in the Knapsack

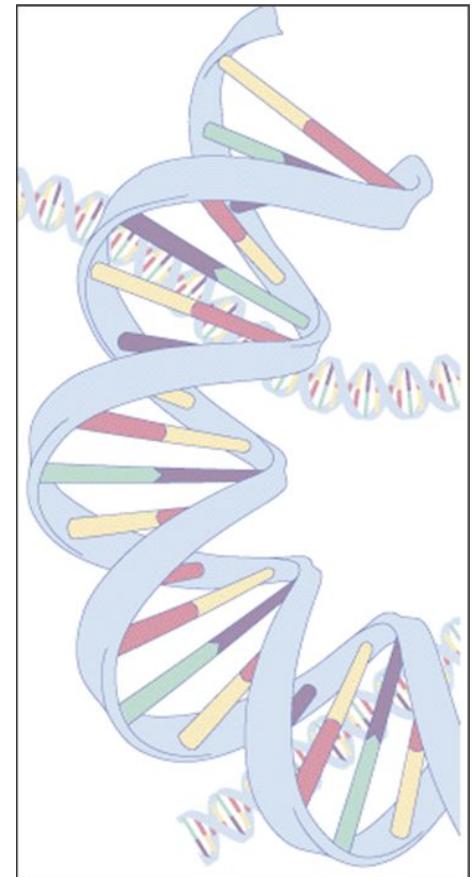
Chromosome: 1010110

Item.	1	2	3	4	5	6	7
Chro	1	0	1	0	1	1	0
Exist?	y	n	y	n	y	y	n

=> Items taken: 1, 3 , 5, 6.

- Generate random population of n chromosomes:

- 0101010
- 1100100
- 0100011



Basic Steps Cont.

Fitness & Selection

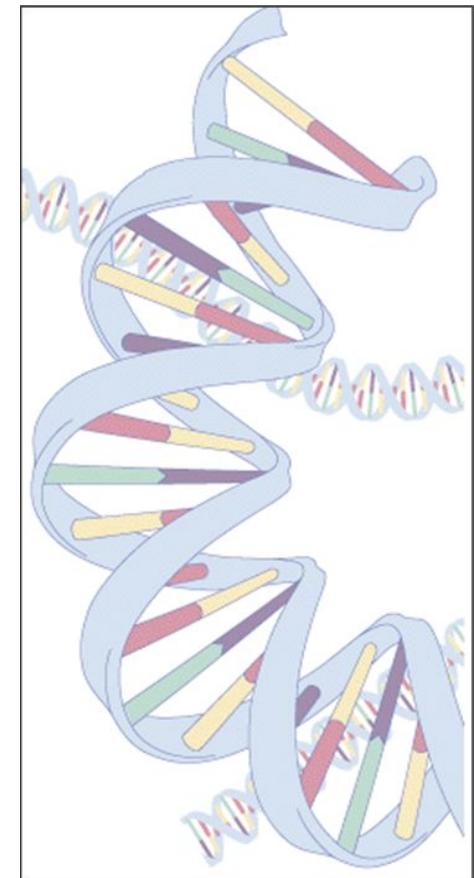
a) 0101010: Benefit= 19, Weight= 24

Item	1	2	3	4	5	6	7
Chro	0	1	0	1	0	1	0
Benefit	5	8	3	2	7	9	4
Weight	7	8	4	10	4	6	4

b) 1100100: Benefit= 20, Weight= 19.

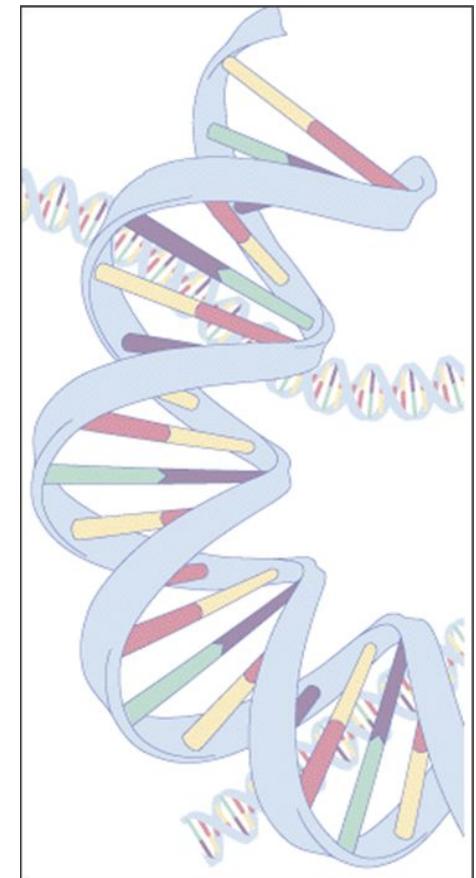
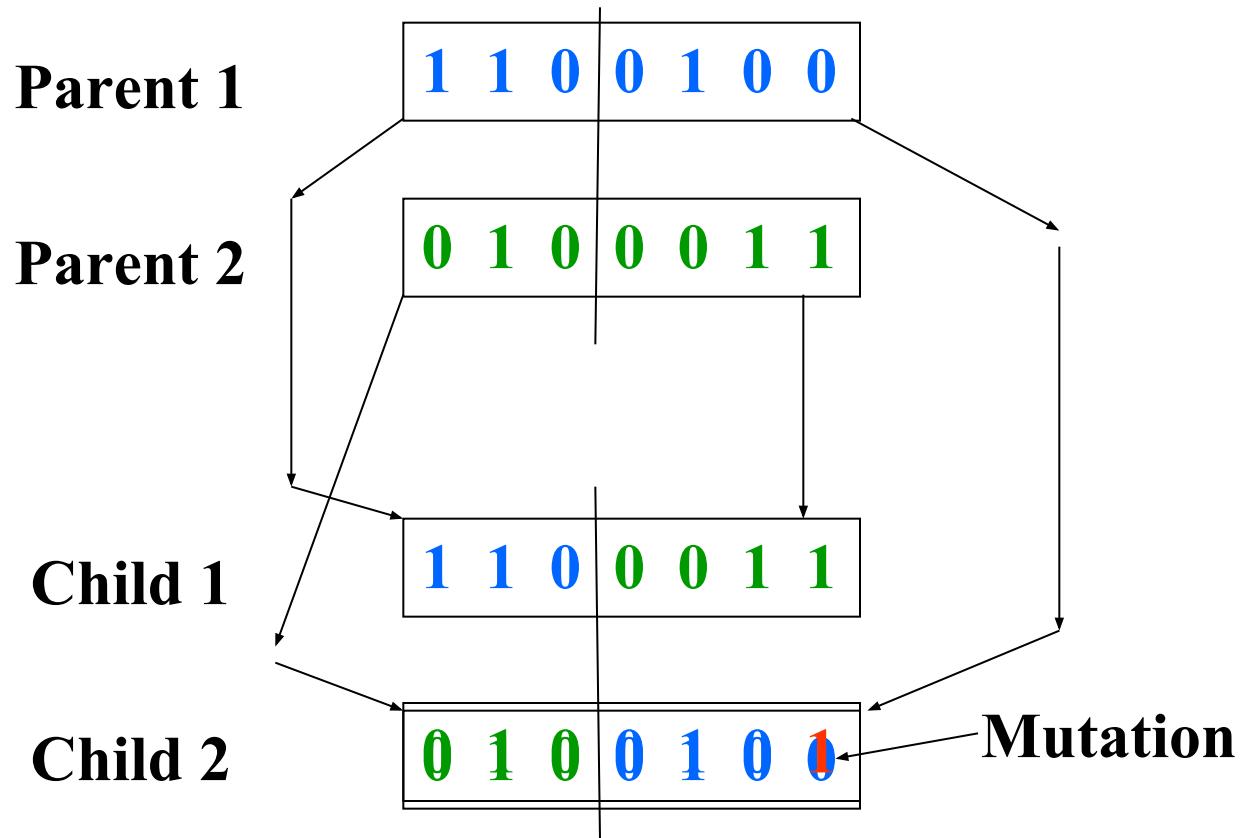
c) 0100011: Benefit= 21, Weight= 18.

=> We select Chromosomes b & c.



Basic Steps Cont.

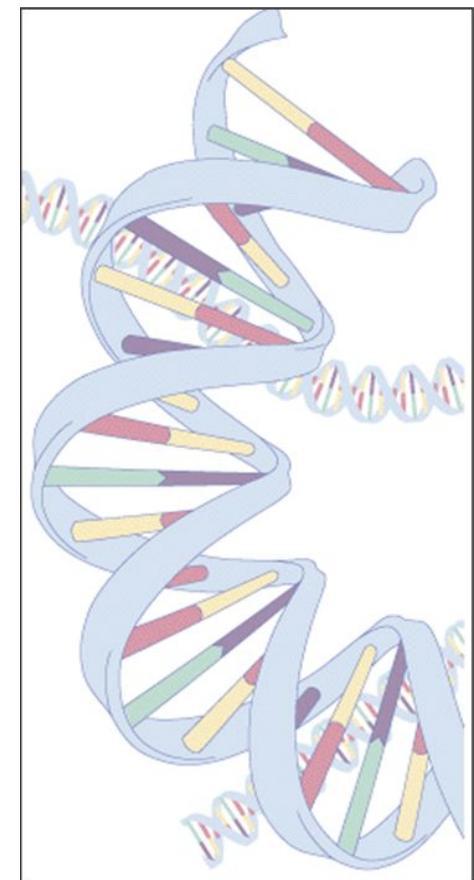
Crossover & Mutation



Basic Steps Cont.

Accepting, Replacing & Testing

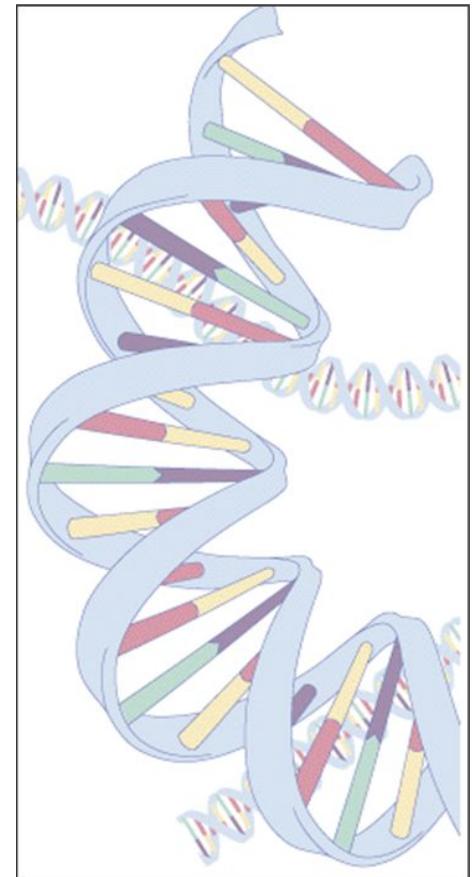
- ✓ Place new offspring in a new population.
- ✓ Use new generated population for a further run of algorithm.
- ✓ If the end condition is satisfied, then **stop**.
End conditions:
 - Number of populations.
 - Improvement of the best solution.
- ✓ Else, return to step 2 **[Fitness]**.



Genetic Algorithm

Conclusion

- GA is nondeterministic – two runs may end with different results
- There's no indication whether best individual is optimal



Pros and Cons

- Pros
 - Faster (and lower memory requirements) than searching a very large search space.
 - Easy, in that if your candidate representation and fitness function are correct, a solution can be found without any explicit analytical work.
- Cons
 - Randomized – not optimal or even complete.
 - Can get stuck on local maxima, though crossover can help mitigate this.
 - It can be hard to work out how best to represent a candidate as a bit string (or otherwise).

Constraint Satisfaction Problem

Constraint Satisfaction Problem (CSP)

- A problem is solved when each variable has a value that satisfies all the constraints on the variable. A problem described this way is called a **constraint satisfaction problem**, or CSP.
- The main idea is to eliminate large portions of the search space all at once by identifying variable/value combinations that violate the constraints.

Definition of CSP

- Constraint satisfaction problem consists of three components, X, D, and C:
 - X is a set of variables, $\{X_1, \dots, X_n\}$.
 - D is a set of domains, $\{D_1, \dots, D_n\}$, one for each variable.
 - C is a set of constraints that specify allowable combinations of values.
- A domain, D, consists of a set of allowable values, $\{v_1, \dots, v_n\}$, for variable X;
 - For example, a Boolean variable would have the domain {true, false}.
- Each constraint C_j consists of a pair (scope, rel),
 - where scope is a tuple of variables that participate in the constraint
 - rel is a relation that defines the values that those variables can take on.

- A relation can be represented as an explicit set of all tuples of values that satisfy the constraint, or as a function that can compute whether a tuple is a member of the relation.
 - For example, if X_1 and X_2 both have the domain $\{1,2,3\}$, then the constraint saying that X_1 must be greater than X_2 can be written as,
 $((X_1, X_2), \{(3,1), (3,2), (2,1)\})$ or as $((X_1, X_2), X_1 > X_2)$.

- CSPs deal with assignments of values to variables, $\{X_i = v_i, X_j = v_j, \dots\}$.
- An assignment that does not violate any constraints is called a **consistent or legal assignment**.
- A **complete assignment** is one in which every variable is assigned a value, and a **solution** to a CSP is a consistent, complete assignment.
- A **partial assignment** is one that leaves some variables unassigned, and a partial solution is a partial assignment that is **consistent**.

Example problem: Map coloring

- A map of Australia showing each of its states and territories.
- We are given the task of coloring each region either red, green, or blue in such a way that no two neighboring regions have the same color.



Example problem: Map coloring

- We define the variables to be the regions:

$$X = \{WA, NT, Q, NSW, V, SA, T\}.$$

- The domain of every variable is the set

$$D_i = \{\text{red, green, blue}\}.$$

- The constraints require neighboring regions to have distinct colors.
- Since there are nine places where regions border, there are nine constraints:

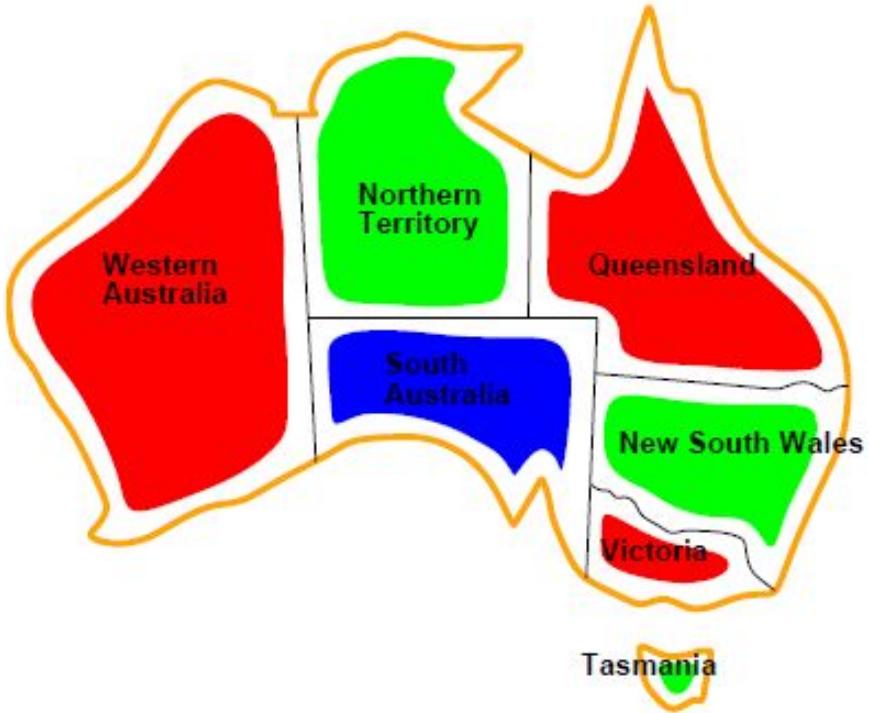
$$C = \{SA \neq WA, SA \neq NT, SA \neq Q, SA \neq NSW, SA \neq V, \\ WA \neq NT, NT \neq Q, Q \neq NSW, NSW \neq V\}.$$

Example problem: Map coloring

- Here we are using abbreviations; $SA \neq WA$ is a shortcut for $((SA, WA), SA \neq WA)$,
- Where $SA \neq WA$ can be fully enumerated in turn as
 $\{(red, green), (red, blue), (green, red), (green, blue),$
 $(blue, red), (blue, green)\}$.
- Solutions to this problem?

Example problem: Map coloring

- There are many possible solutions to this problem, such as
 $\{WA = \text{red}, NT = \text{green}, Q = \text{red}, NSW = \text{green}, V = \text{red}, SA = \text{blue}, T = \text{red}\}.$

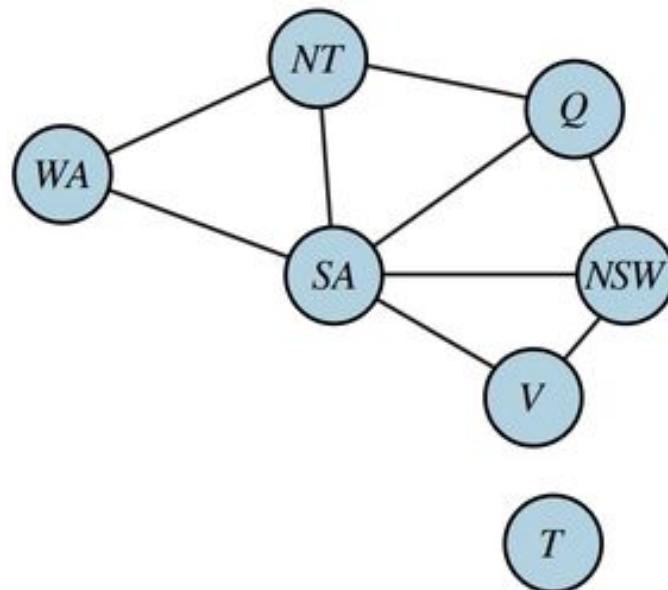


Solutions are assignments satisfying all constraints, e.g.,

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

Example problem: Map coloring

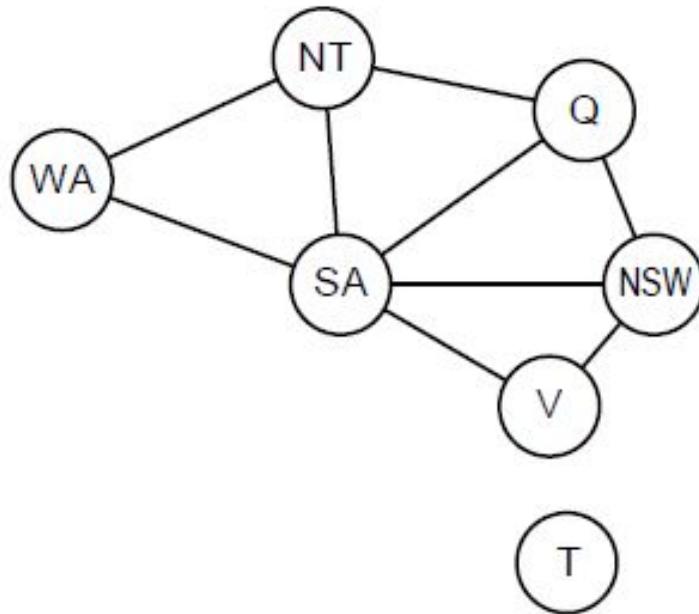
- It can be helpful to visualize a CSP as a constraint graph, as shown in Figure.
- The nodes of the graph correspond to variables of the problem, and an edge connects any two variables that participate in a constraint.



Constraint graph

Binary CSP: each constraint relates at most two variables

Constraint graph: nodes are variables, arcs show constraints



General-purpose CSP algorithms use the graph structure to speed up search. E.g., Tasmania is an independent subproblem!

Types of CSP

Discrete variables

finite domains; size $d \Rightarrow O(d^n)$ complete assignments

- ◊ e.g., Boolean CSPs, incl. Boolean satisfiability (NP-complete)

infinite domains (integers, strings, etc.)

- ◊ e.g., job scheduling, variables are start/end days for each job
- ◊ need a constraint language, e.g., $StartJob_1 + 5 \leq StartJob_3$
- ◊ linear constraints solvable, nonlinear undecidable

Continuous variables

- ◊ e.g., start/end times for Hubble Telescope observations
- ◊ linear constraints solvable in poly time by LP methods

Varieties of constraints

Unary constraints involve a single variable,

e.g., $SA \neq green$

Binary constraints involve pairs of variables,

e.g., $SA \neq WA$

Higher-order constraints involve 3 or more variables,

e.g., cryptarithmetic column constraints

Preferences (soft constraints), e.g., red is better than $green$

often representable by a cost for each variable assignment

→ constrained optimization problems

Cryptarithmetic Problem

- Cryptarithmetic Problem is a type of constraint satisfaction problem where the game is about digits and its unique replacement either with alphabets or other symbols.
- In **cryptarithmetic problem**, the digits (0-9) get substituted by some possible alphabets or symbols.
- The task in cryptarithmetic problem is to substitute each digit with an alphabet to get the result arithmetically correct.

Rules and constraints

- There should be a unique digit to be replaced with a unique alphabet.
- The result should satisfy the predefined arithmetic rules, i.e., $2+2=4$, nothing else.
- Digits should be from **0-9** only.
- The problem can be solved from both sides, i.e., **lefthand side (L.H.S)**, or **righthand side (R.H.S)**

Example-1

BASE

+BALL

GAMES

Example-1

$$\begin{array}{r} \text{B A S E} \\ + \text{B A L L} \\ \hline \text{G A M E S} \end{array}$$

Variables: {B,A,S,E,L,G,M}
Domain:{0,1,2,3,4,5,6,7,8,9}
Constraint: AllDiffz(B,A,S,E,L,G,M)

Example-1

BASE
+ **BALL** -

GAMES

B	5
B	5
G A	10
G	1
A	0

Example-1

B A S E
+ B A L L -
G A M E S

B	6
B	6
G A	12
G	1
A	2

Example-1

BASE

+BALL

GAMES

Cr	0	Cr	0
B	6	A	2
B	6	A	2
GA	12	M	4
G	1		
A	2		

Example-1

BASE
+ **BALL** -
GAMES

Cr	0	Cr	0	Cr	
B	6	A	2	S	3
B	6	A	2	L	5
GA	12	M	4	E	8
G	1				
A	2				

Example-1

BASE
+ **BALL** -
GAMES

Cr	0	Cr	0	C r		Cr	
B	6	A	2	S	3	E	8
B	6	A	2	L	5	L	5
GA	12	M	4	E	8	S	
G	1						
A	2						

Example-1 Solution

BASE

+BALL

GAMES

Cr	0	Cr	0	Cr		Cr	
B	7	A		S		E	
B	7	A		L		L	
GA	14	M		E		S	
G	1						
A	4						

Example-1 Solution

BASE

+BALL

GAMES

Cr	0	Cr	0	Cr		Cr	
B	7	A	4	S		E	
B	7	A	4	L		L	
GA	14	M	8	E		S	
G	1						
A	4						

Example-1 Solution

BASE

+BALL

GAMES

Cr	0	Cr	1	Cr		Cr	
B	7	A	4	S	8	E	
B	7	A	4	L	5	L	
GA	14	M	9	E	3	S	
G	1						
A	4						

Example-1 Solution

BASE

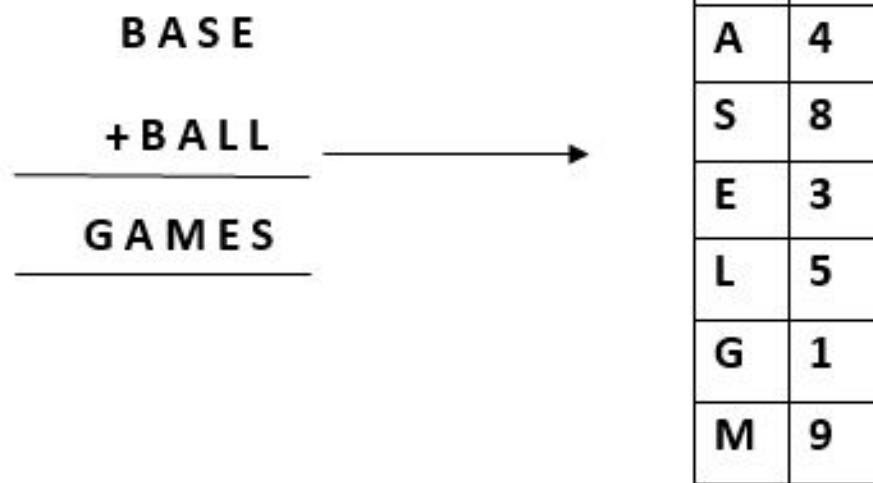
+BALL

GAMES

Cr	0	Cr	1	Cr	0	Cr	0
B	7	A	4	S	8	E	3
B	7	A	4	L	5	L	5
GA	14	M	9	E	3	S	8
G	1						
A	4						

BASE
+ **BALL**

GAMES



B	7
A	4
S	8
E	3
L	5
G	1
M	9

S E N D

+ M O R E

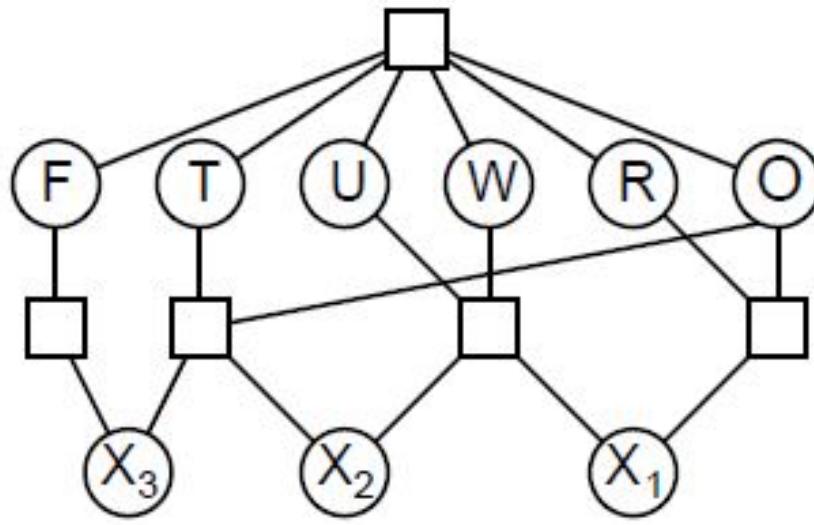
M O N E Y

S E N D
+ M O R E
—
M O N E Y
—

Cr		Cr		Cr		Cr	
S		E		N		D	
M		O		R		E	
MO		N		E		Y	
M							
O							

Example: Cryptarithmetic

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



Variables: $F \ T \ U \ W \ R \ O \ X_1 \ X_2 \ X_3$

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

Constraints

$\text{alldiff}(F, T, U, W, R, O)$

$O + O = R + 10 \cdot X_1$, etc.

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

<i>Cr</i>	0	<i>Cr</i>	0	<i>Cr</i>	0
T	7	W	3	O	4
T	7	W	3	O	4
FO	14	U	6	R	8
F	1				
O	4				

Real-world CSPs

Assignment problems

e.g., who teaches what class

Timetabling problems

e.g., which class is offered when and where?

Hardware configuration

Spreadsheets

Transportation scheduling

Factory scheduling

Floorplanning

Notice that many real-world problems involve real-valued variables

Standard search formulation (incremental)

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

States are defined by the values assigned so far

- ◊ **Initial state:** the empty assignment, {}
- ◊ **Successor function:** assign a value to an unassigned variable that does not conflict with current assignment.
⇒ fail if no legal assignments (not fixable!)
- ◊ **Goal test:** the current assignment is complete

- 1) This is the same for all CSPs! 😊
- 2) Every solution appears at depth n with n variables
⇒ use depth-first search
- 3) Path is irrelevant, so can also use complete-state formulation
- 4) $b = (n - \ell)d$ at depth ℓ , hence $n!d^n$ leaves!!!! 😞

Genetic Algorithm

Example

Maximizing a Function

- Consider the problem of maximizing the function, $f(x) = x^2$
- where x is permitted to vary between 0 to 31.

- Step 1: For using genetic algorithms approach, one must first code the decision variable ‘x’ into a finite length string.
- Using a five bit (binary integer) unsigned integer, numbers between 0(00000) and 31(11111) can be obtained.
- The objective function here is $f(x) = x^2$, which is to be maximized.
- A single generation of a genetic algorithm is performed here with encoding, selection, crossover and mutation.

Select initial population. Psize=4

String No.	Initial popu- lation (randomly selected)
1	01100
2	11001
3	00101
4	10011

- Step 2: Obtain the decoded x values for the initial population generated. Consider string 1, Thus for all the four strings the decoded values are obtained.
- Step 3: Calculate the fitness or objective function. This is obtained by simply squaring the ‘x’ value, since the given function is $f(x) = x^2$.
- When, $x = 12$, the fitness value is, $f(x) = 144$ for $x = 25$, $f(x) = 625$ and so on, until the entire population is computed

String No.	Initial population (randomly selected)	x value	Fitness value $f(x) = x^2$
1	01100	12	144
2	11001	25	625
3	00101	5	25
4	10011	19	361
Sum			1155
average			288.75
maximum			625

- Step 4: Compute the probability of selection,

$$\text{Prob}(x_i) = \frac{f(x_i)}{\sum_{i=1}^n f(x_i)}$$

where

n = no of populations

f(x)=fitness value corresponding to a particular individual in the population

$\Sigma f(x)$ - Summation of all the fitness value of the entire population.

For e.g: Considering string 1, Fitness $f(x) = 144$ $\Sigma f(x) = 1155$

The probability that string 1 occurs is given by, $P_1 = 144/1155 = 0.1247$

String No.	Initial popu- lation (randomly selected)	x value	Fitness value $f(x) = x^2$	Prob _i	Percentage probability
1	01100	12	144	0.1247	12.47%
2	11001	25	625	0.5411	54.11%
3	00101	5	25	0.0216	2.16%
4	10011	19	361	0.3126	31.26%
Sum			1155	1.0000	100%
average			288.75	0.2500	25%
maximum			625	0.5411	54.11%

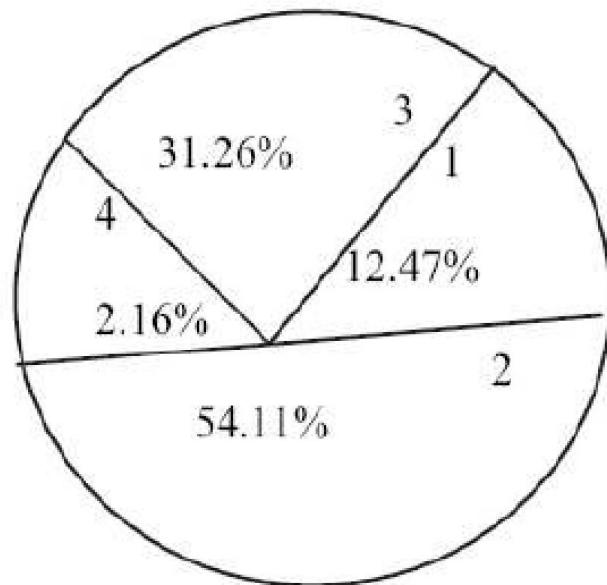
- Step 5: The next step is to calculate the expected count, which is calculated as,

$$\text{Expected count} = \frac{f(x_i)}{(\sum_{i=1}^n f(x_i))/n}$$

The expected count gives an idea of which population can be selected for further processing in the mating pool.

String No.	Initial popu- lation (randomly selected)	x value	Fitness value $f(x) = x^2$	Prob _i	Percentage probability	Expected count
1	01100	12	144	0.1247	12.47%	0.4987
2	11001	25	625	0.5411	54.11%	2.1645
3	00101	5	25	0.0216	2.16%	0.0866
4	10011	19	361	0.3126	31.26%	1.2502
Sum			1155	1.0000	100%	4.0000
average			288.75	0.2500	25%	1.0000
maximum			625	0.5411	54.11%	2.1645

- Step 6: Now the actual count is to be obtained to select the individuals, which would participate in the crossover cycle using Roulette wheel selection:



actual count

- String 1 occupies 12.47%, so there is a chance for it to occur at least once. Hence its actual count may be 1.
- With string 2 occupying 54.11% of the Roulette wheel, it has a fair chance of being selected twice. Thus its actual count can be considered as 2.
- On the other hand, string 3 has the least probability percentage of 2.16%, so their occurrence for next cycle is very poor. As a result, its actual count is 0.
- String 4 with 31.26% has at least one chance for occurring while Roulette wheel is spun, thus its actual count is 1.

String No.	Initial population (randomly selected)	x value	Fitness value $f(x) = x^2$	Prob _i	Percentage probability	Expected count	Actual count
1	01100	12	144	0.1247	12.47%	0.4987	1
2	11001	25	625	0.5411	54.11%	2.1645	2
3	00101	5	25	0.0216	2.16%	0.0866	0
4	10011	19	361	0.3126	31.26%	1.2502	1
Sum			1155	1.0000	100%	4.0000	4
average			288.75	0.2500	25%	1.0000	1
maximum			625	0.5411	54.11%	2.1645	2

- **Step 7:** Now, writing the mating pool based upon the actual count as shown in Table.
 - The actual count of string no 1 is 1, hence it occurs once in the mating pool.
 - The actual count of string no 2 is 2, hence it occurs twice in the mating pool.
 - Since the actual count of string no 3 is 0, it does not occur in the mating pool.
 - Similarly, the actual count of string no 4 being 1, it occurs once in the mating pool.
 - Based on this, the mating pool is formed.
-

String No.	Mating pool
1	0 1 1 0 0
2	1 1 0 0 1
2	1 1 0 0 1
4	1 0 0 1 1

- **Step 8:** Crossover operation is performed to produce new offspring (children).
- The crossover point is specified and based on the crossover point, single point crossover is performed and new offspring is produced.
 - Parent 1 0 1 1 0 0
 - Parent 2 1 1 0 0 1
 - The offspring is produced as,
 - Offspring 1 0 1 1 0 1
 - Offspring 2 1 1 0 0 0

Note: The crossover probability was assumed to 1.0

String No.	Mating pool	Crossover point	Offspring after crossover	x value	Fitness value $f(x) = x^2$
1	0 1 1 0 0	4	0 1 1 0 1	13	169
2	1 1 0 0 1	4	1 1 0 0 0	24	576
2	1 1 0 0 1	2	1 1 0 1 1	27	729
4	1 0 0 1 1	2	1 0 0 0 1	17	289
Sum					1763
average					440.75
maximum					729

- **Step 9:** After crossover operations, new off springs are produced and ‘x’ values are decodes and fitness is calculated.
- **Step 10:** In this step, mutation operation is performed to produce new off springs after crossover operation.
 - In mutation-flipping operation is performed and new off springs are produced.

String No.	Offspring after crossover	Mutation chromosomes for flipping	Offspring after Mutation	X value	Fitness value $F(x) = x^2$
1	0 1 1 0 1	1 0 0 0 0	1 1 1 0 1	29	841
2	1 1 0 0 0	0 0 0 0 0	1 1 0 0 0	24	576
2	1 1 0 1 1	0 0 0 0 0	1 1 0 1 1	27	729
4	1 0 0 0 1	0 0 1 0 0	1 0 1 0 0	20	400
Sum					2546
average					636.5
maximum					841

Note: The mutation probability was assumed to 0.001

- Once the off springs are obtained after mutation, they are decoded to x value and find fitness values are computed.
-
- This completes one generation.
- The mutation is performed on a bit-bit by basis .

Backtracking

CSP

- Sometimes we can finish the constraint propagation process and still have variables with multiple possible values.
- In that case we have to search for a **solution**.
- For a CSP with n variables of domain size d we would end up with a search tree where all the complete assignments (and thus all the solutions) are leaf nodes at depth n . But notice that the branching factor at the top level would be nd because any of d values can be assigned to any of n variables.
- At the next level, the branching factor is $(n - 1)d$, and so on for n levels.
- So the tree has $n!.d^n$ leaves, even though there are only d^n possible complete assignments!

Backtracking search

Variable assignments are commutative, i.e.,

[$WA = \text{red}$ then $NT = \text{green}$] same as [$NT = \text{green}$ then $WA = \text{red}$]

Only need to consider assignments to a single variable at each node

$\Rightarrow b = d$ and there are d^n leaves

Depth-first search for CSPs with single-variable assignments
is called backtracking search

Backtracking search is the basic uninformed algorithm for CSPs

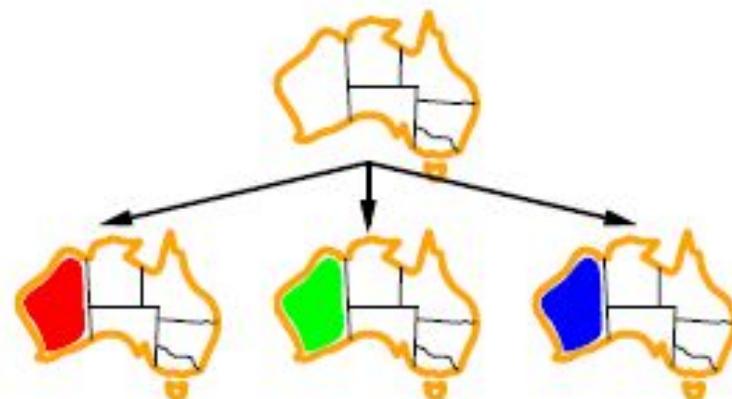
Backtracking search

```
function BACKTRACKING-SEARCH(csp) returns solution/failure
    return RECURSIVE-BACKTRACKING({ }, csp)
function RECURSIVE-BACKTRACKING(assignment, csp) returns soln/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 given 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
```

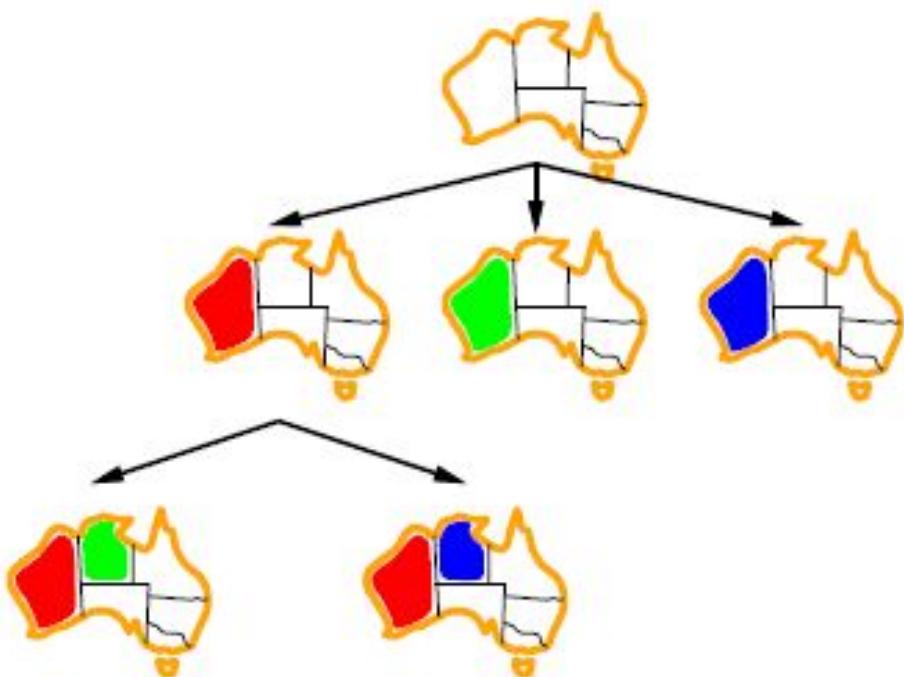
Backtracking example



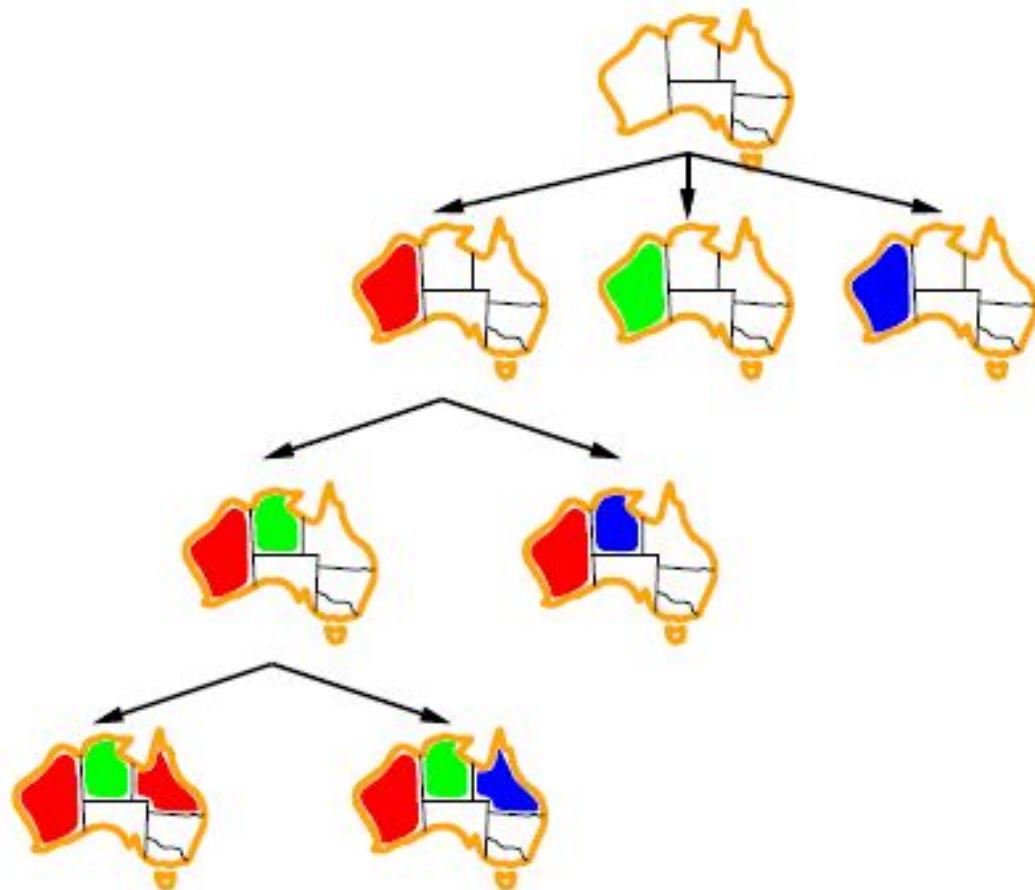
Backtracking example



Backtracking example



Backtracking example



Improving backtracking efficiency

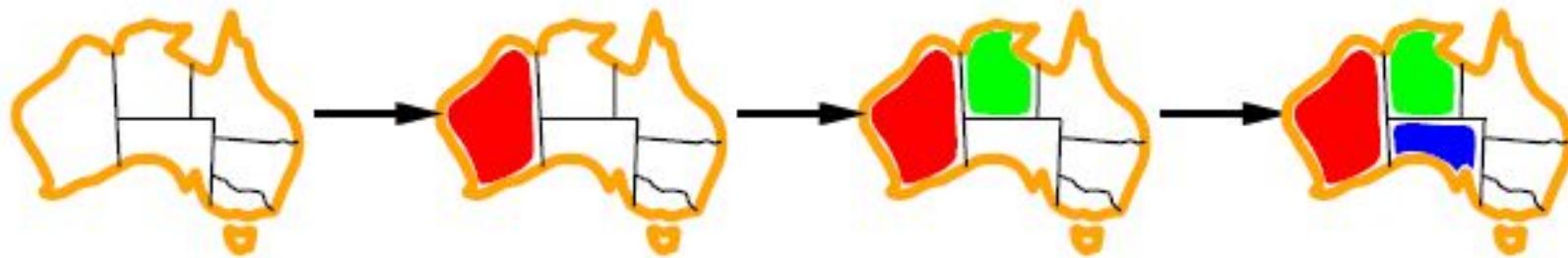
General-purpose methods can give huge gains in speed:

1. Which variable should be assigned next?
2. In what order should its values be tried?
3. Can we detect inevitable failure early?
4. Can we take advantage of problem structure?

Minimum remaining values

Minimum remaining values (MRV):

choose the variable with the fewest legal values

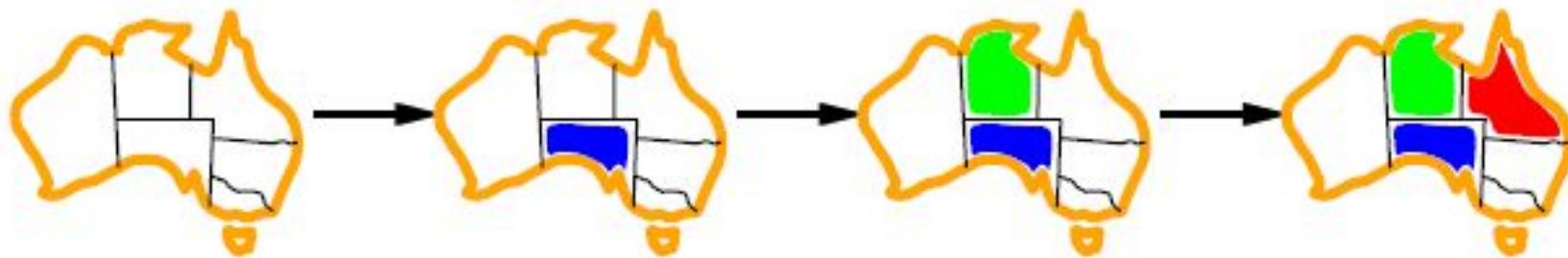


Degree heuristic

Tie-breaker among MRV variables

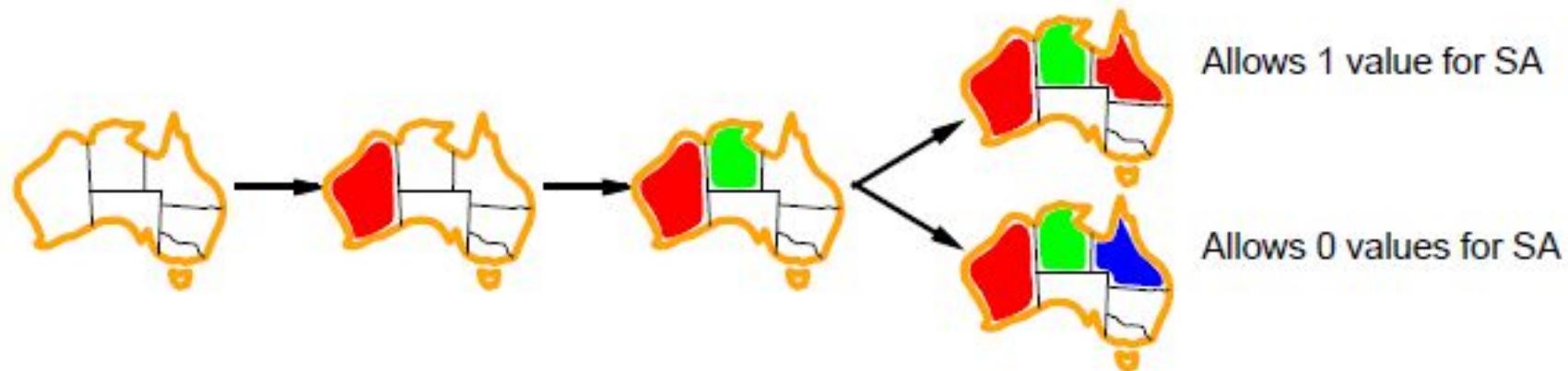
Degree heuristic:

choose the variable with the most constraints on remaining variables



Least constraining value

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



Combining these heuristics makes 1000 queens feasible

Forward checking

Idea: Keep track of remaining legal values for unassigned variables

Terminate search when any variable has no legal values



WA

NT

Q

NSW

V

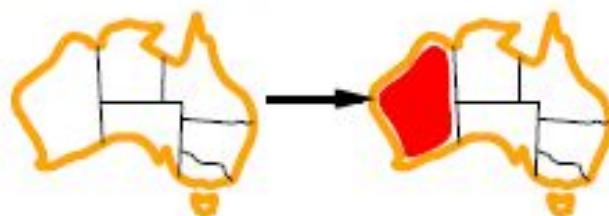
SA

T



Forward checking

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



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

Forward checking

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



WA

NT

Q

NSW

V

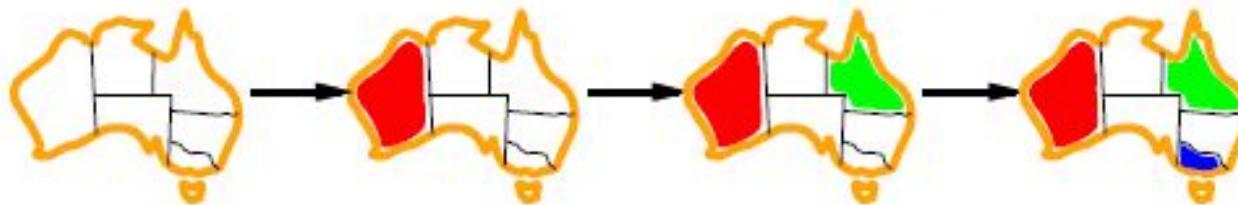
SA

T

[Red]	[Green]	[Blue]	[Red]	[Green]	[Blue]	[Red]	[Green]	[Blue]
[Red]		[Green]	[Blue]	[Red]	[Green]	[Blue]	[Red]	[Green]
[Red]		[Blue]	[Green]	[Red]	[Blue]		[Blue]	[Red]

Forward checking

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



WA

NT

Q

NSW

V

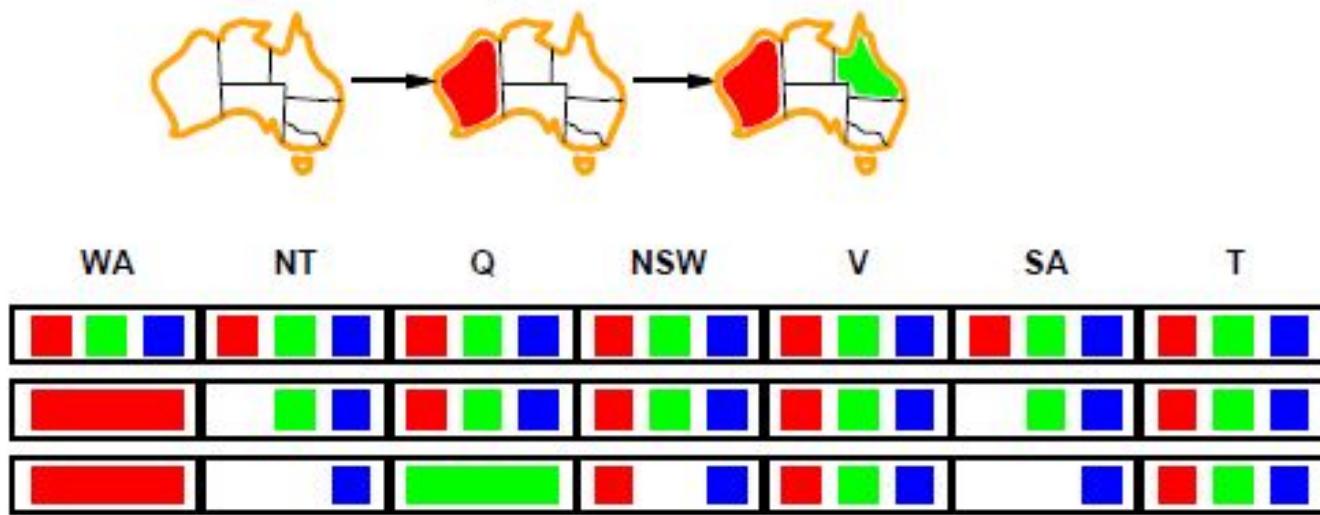
SA

T

█ Red	█ Green	█ Blue	█ Red	█ Green	█ Blue	█ Red	█ Green	█ Blue	█ Red	█ Green	█ Blue	█ Red	█ Green	█ Blue		
█ Red			█ Red	█ Green	█ Blue	█ Red	█ Green	█ Blue	█ Red	█ Green	█ Blue		█ Green	█ Blue		
█ Red				█ Green		█ Red		█ Blue	█ Red		█ Green	█ Blue		█ Red	█ Green	█ Blue
█ Red						█ Red			█ Red		█ Blue			█ Red	█ Green	█ Blue

Constraint propagation

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



NT and *SA* cannot both be blue!

Constraint propagation repeatedly enforces constraints locally

Arc-consistency

- A variable in a CSP is arc-consistent (edge-consistent) if every value in its domain satisfies the variable's binary constraints.
- X_i is arc-consistent with respect to another variable X_j ; if for every value in the current domain D_i there is some value in the domain D_j that satisfies the binary constraint on the arc (X_i, X_j) .
- A graph is arc-consistent if every variable is arc-consistent with every other variable.

Example

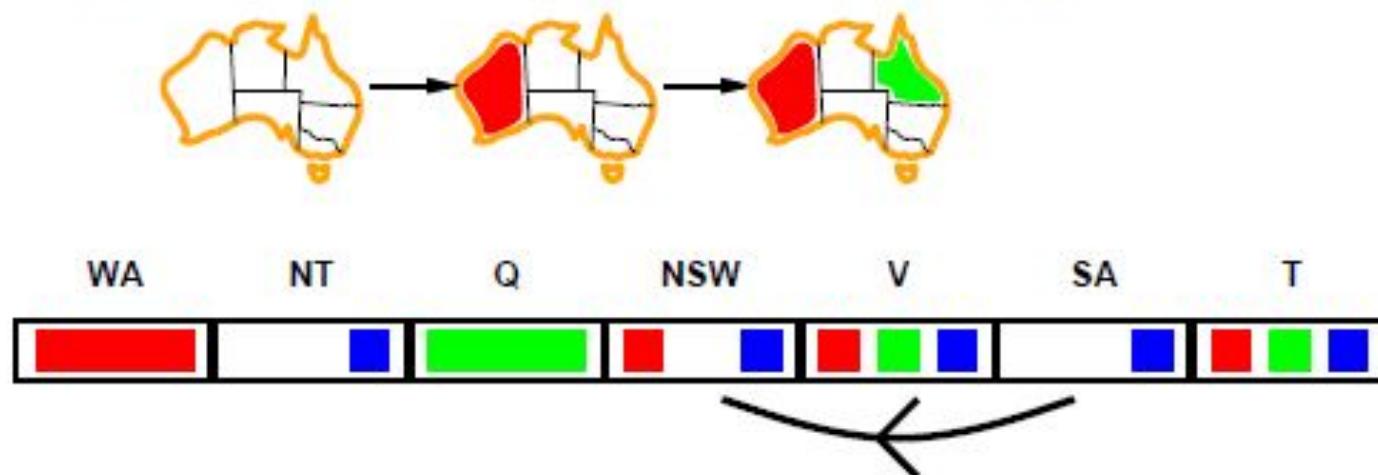
- Consider the constraint $Y = X^2$
 - where the domain of both X and Y is the set of decimal digits.
- We can write this constraint explicitly as
 $((X,Y), \{(0,0), (1,1), (2,4), (3,9)\})$
- To make X arc-consistent with respect to Y ,
 - we reduce X 's domain to $\{0, 1, 2, 3\}$.
- If we also make Y arc-consistent with respect to X ,
 - then Y 's domain becomes $\{0, 1, 4, 9\}$, and the whole CSP is arc-consistent.

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 y

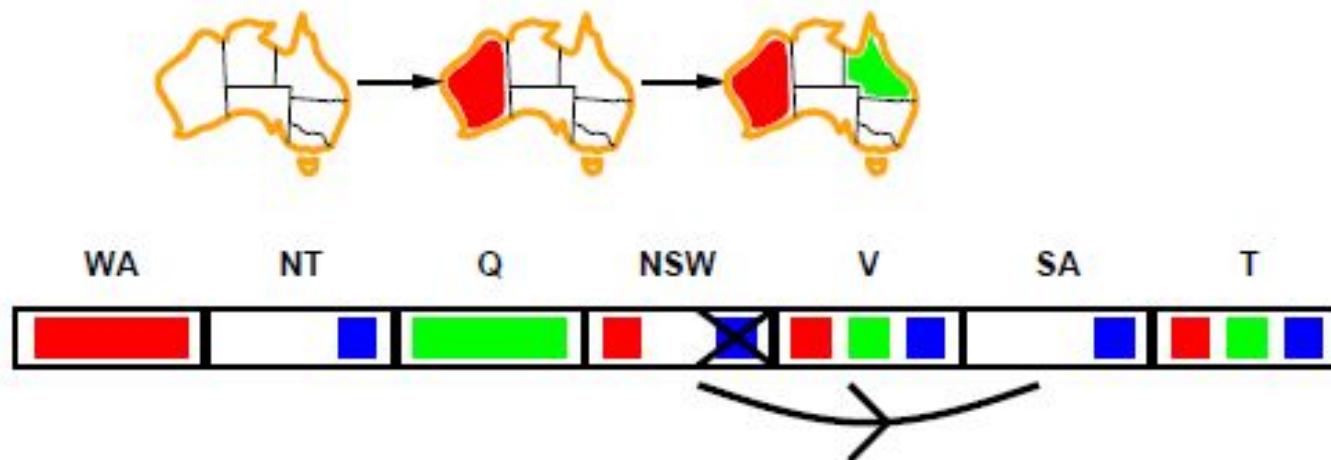


Arc consistency

Simplest form of propagation makes each arc consistent

$X \rightarrow Y$ is consistent iff

for every value x of X there is some allowed y

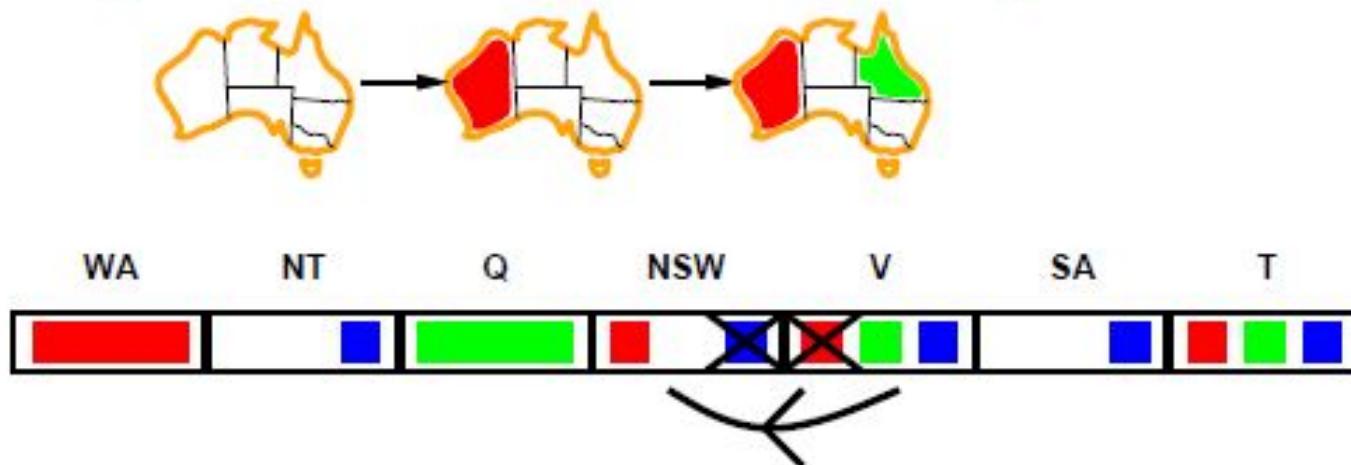


Arc consistency

Simplest form of propagation makes each arc consistent

$X \rightarrow Y$ is consistent iff

for every value x of X there is some allowed y



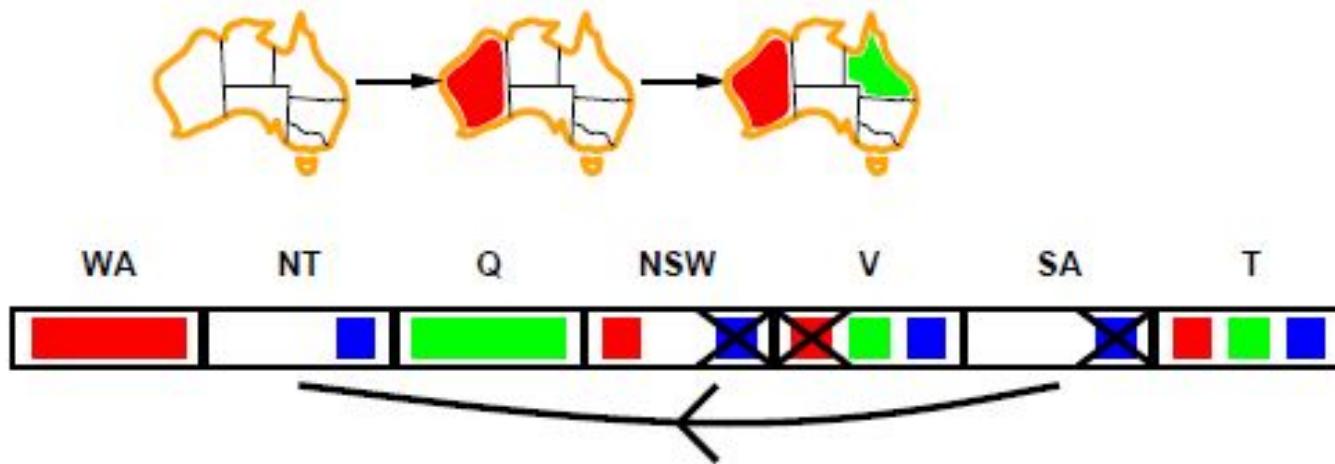
If X loses a value, neighbors of X need to be rechecked

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 y



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

Arc consistency detects failure earlier than forward checking

Can be run as a preprocessor or after each assignment

Arc consistency algorithm

```
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 REMOVE-INCONSISTENT-VALUES( $X_i, X_j$ ) then
      for each  $X_k$  in NEIGHBORS[ $X_i$ ] do
        add  $(X_k, X_i)$  to queue



---


function REMOVE-INCONSISTENT-VALUES( $X_i, X_j$ ) returns true iff succeeds
  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 the constraint  $X_i \leftrightarrow X_j$ 
      then delete  $x$  from DOMAIN[ $X_i$ ]; removed  $\leftarrow \text{true}$ 
  return removed
```

$O(n^2d^3)$, can be reduced to $O(n^2d^2)$ (but detecting all is NP-hard)

AC-3 algorithm

We can represent the AC-3 algorithm in 3 steps:

1. Get all the constraints and turn each one into two arcs. For example:
 $A > B$ becomes $A > B$ and $B < A$.
2. Add all the arcs to a queue.
3. Repeat until the queue is empty:
 - 3.1. Take the first arc (x, y) , off the queue (**dequeue**).
 - 3.2. For **every** value in the x domain, there must be some value of the y domain.
 - 3.3. Make x arc consistent with y . To do so, remove values from x domain for which there is no possible corresponding value for y domain.
 - 3.4. If the x domain has changed, add all arcs of the form (k, x) to the queue (**enqueue**). Here k is another variable different from y that has a relation to x .

Example

- Let's take this example, where we have three variables A , B , and C and the constraints: $A > B$ and $B = C$.
- $A=\{1,2,3\}$
- $B=\{1,2,3\}$
- $C=\{1,2,3\}$

- **Step 1: Generate All Arcs**

A>B

B<A

B=C

C=B

- Step 2: Create the Queue

Queue:

A>B
B<A
B=C
C=B

Arcs

A>B
B<A
B=C
C=B

- Step 3: Iterate Over the Queue

-

Queue:

A>B
B<A
B=C
C=B

A={1,2,3}
B={1,2,3}
C={1,2,3}

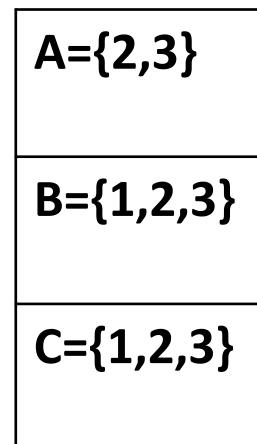
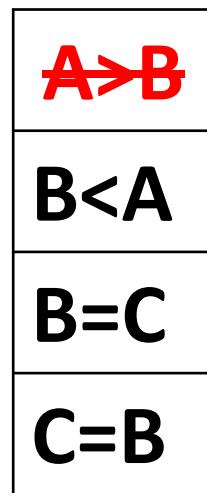
A>B
B<A
B=C
C=B

Arcs

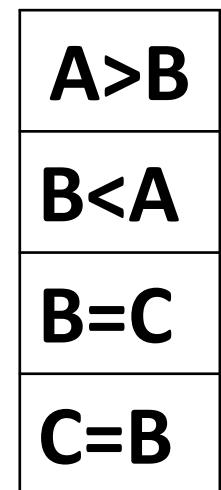
- Step 3: Iterate Over the Queue

-

Queue:



Arcs



- Step 3: Iterate Over the Queue

-

Queue:

A>B
B<A
B=C
C=B

A={2,3}
B={1,2,3}
C={1,2,3}

A>B
B<A
B=C
C=B

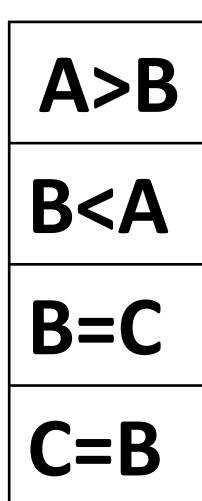
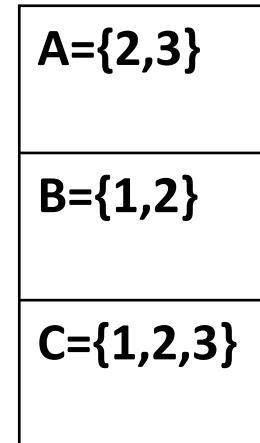
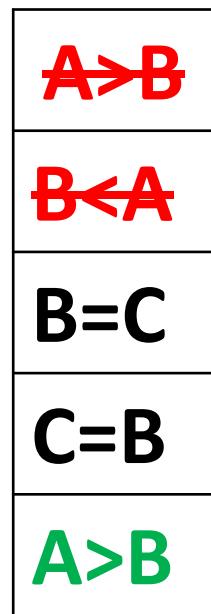
Arcs

- Step 3: Iterate Over the Queue

-

Arcs

Queue:

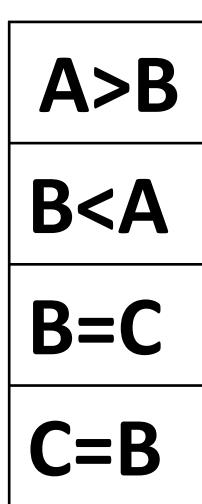
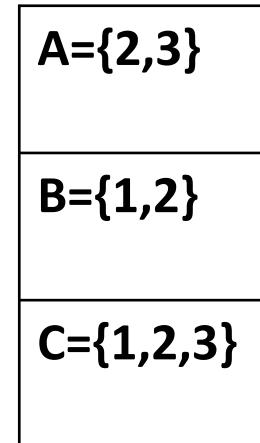
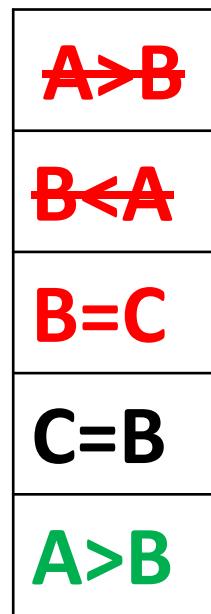


- Step 3: Iterate Over the Queue

-

Arcs

Queue:

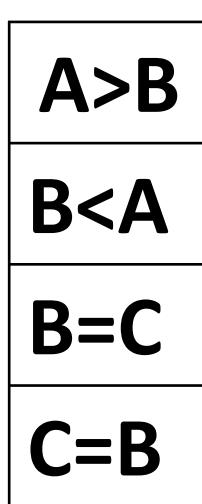
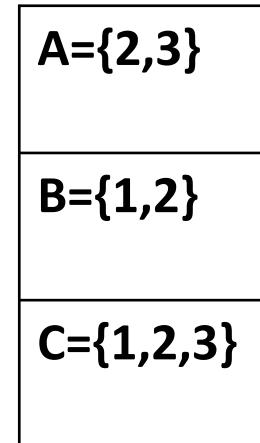
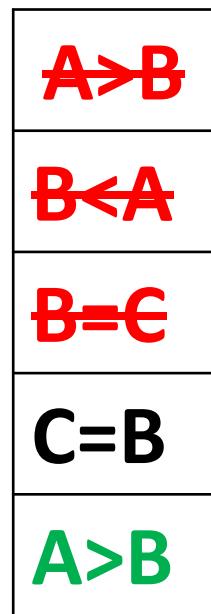


- Step 3: Iterate Over the Queue

-

Arcs

Queue:

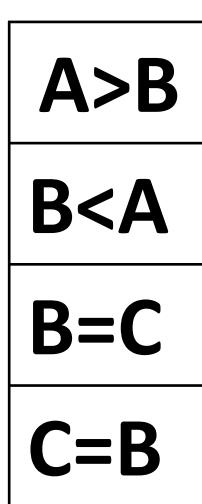
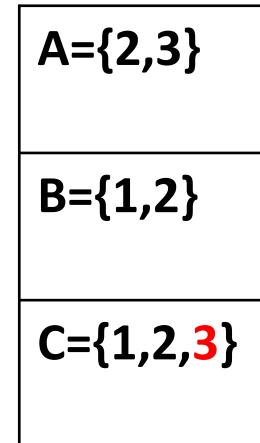
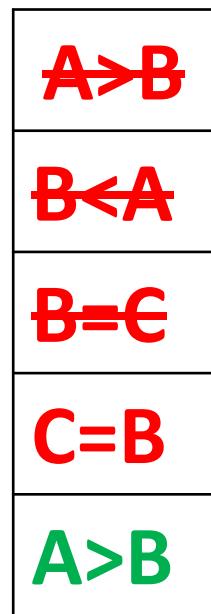


- Step 3: Iterate Over the Queue

-

Arcs

Queue:

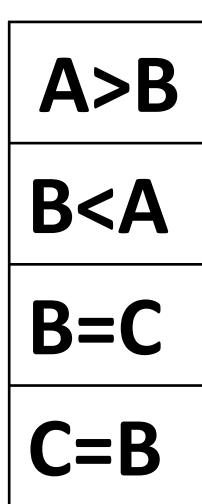
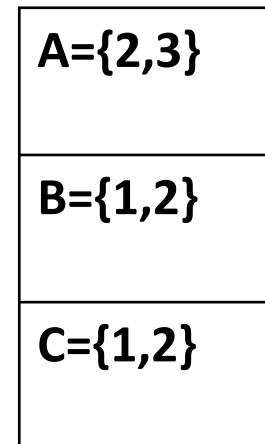
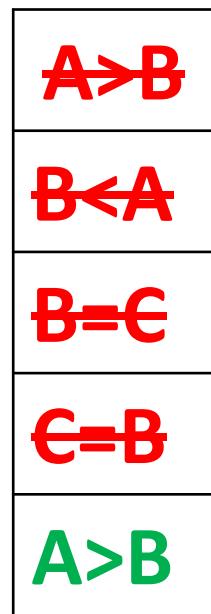


- Step 3: Iterate Over the Queue

-

Arcs

Queue:



- Step 3: Iterate Over the Queue

-

Queue:

A>B
B<A
B=C
C=B
A>B
B=C

A={2,3}

B={1,2}

C={1,2}

Arcs

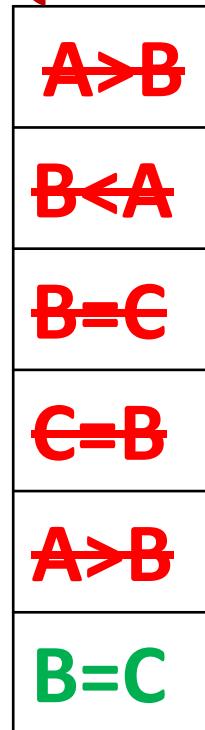
A>B
B<A
B=C
C=B

- Step 3: Iterate Over the Queue

-

Arcs

Queue:



A={2,3}

B={1,2}

C={1,2}

A>B

B<A

B=C

C=B

- Step 3: Iterate Over the Queue

-

Arcs

Queue:

A>B
B<A
B=C
C=B
A>B
B=C

A={2,3}

B={1,2}

C={1,2}

A>B
B<A
B=C
C=B

- Step 4: Stop if no more arc in the queue
-

Queue:

A>B
B<A
B=C
C=B
A>B
B=C

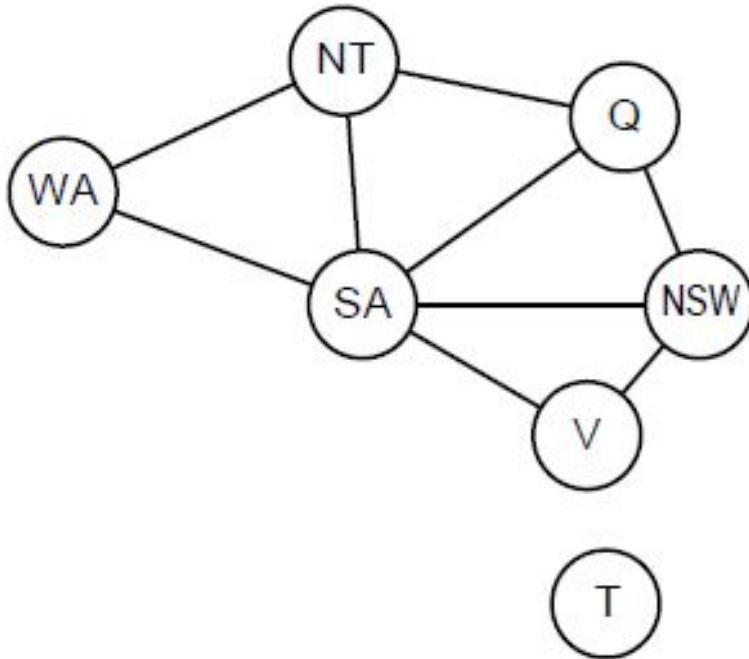
Arcs
A>B
B<A
B=C
C=B

A={2,3}

B={1,2}

C={1,2}

Problem structure



Tasmania and mainland are independent subproblems

Identifiable as connected components of constraint graph

Problem structure contd.

Suppose each subproblem has c variables out of n total

Worst-case solution cost is $n/c \cdot d^c$, linear in n

E.g., $n = 80$, $d = 2$, $c = 20$

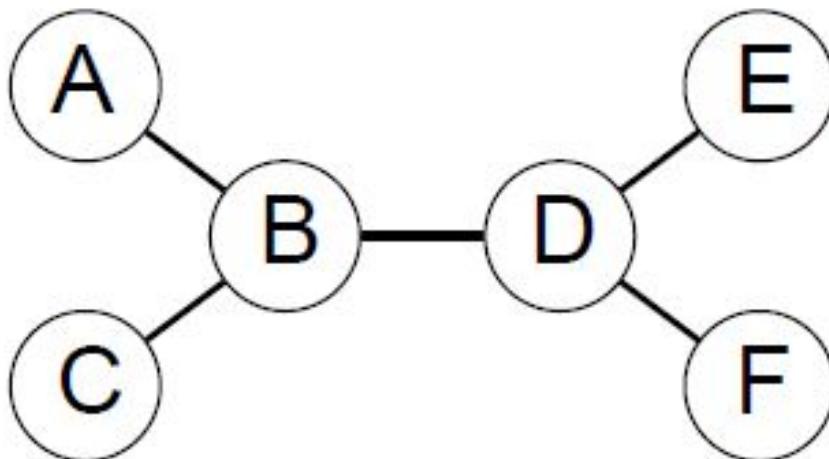
$2^{80} = 4$ billion years at 10 million nodes/sec

$4 \cdot 2^{20} = 0.4$ seconds at 10 million nodes/sec

- A Constraint graph is a tree when any two variables are connected by only one path.
- We will show that any tree-structure CSP can be solved in time linear in the number of variables.
- The key is a new notion of consistency, called directional arc consistency or DAC.
- A CSP is defined to be directional arc-consistent under an ordering of variables X_1, X_2, \dots, X_n , if and only if every X_i is arc consistent with each X_j for $j > i$.

- To solve a tree-structured CSP, first pick any variable to be the root of the tree, and choose an ordering of the variables such that each variable appears after its parent in the tree.
- Such an ordering is called a **topological sort**.

Tree-structured CSPs



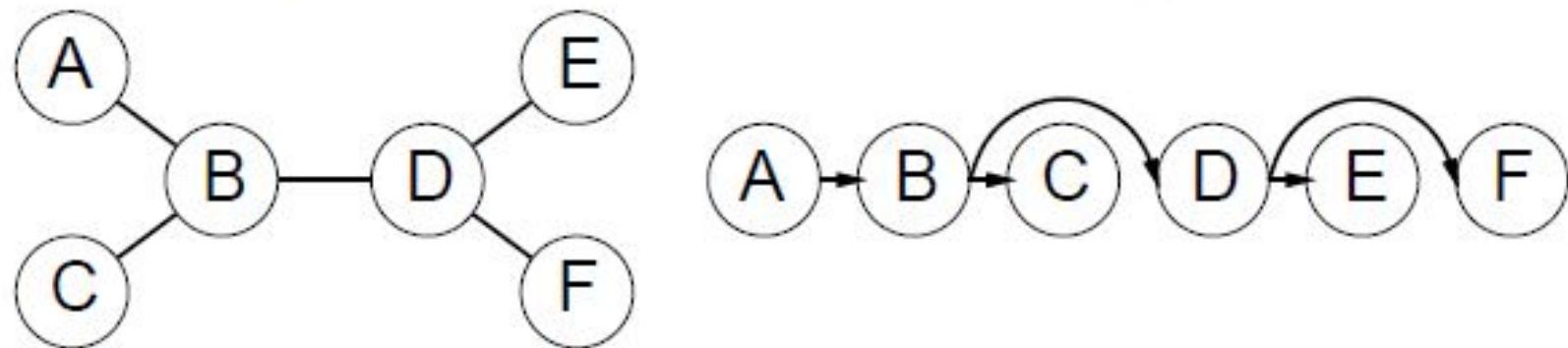
Theorem: if the constraint graph has no loops, the CSP can be solved in $O(nd^2)$ time

Compare to general CSPs, where worst-case time is $O(d^n)$

This property also applies to logical and probabilistic reasoning:
an important example of the relation between syntactic restrictions
and the complexity of reasoning.

Algorithm for tree-structured CSPs

1. Choose a variable as root, order variables from root to leaves such that every node's parent precedes it in the ordering

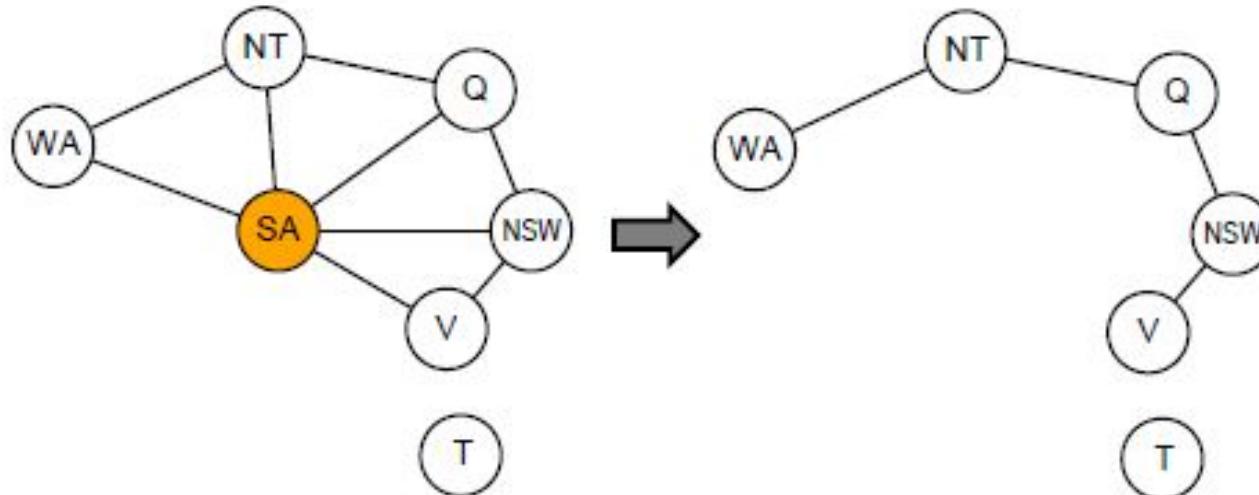


2. For j from n down to 2, apply $\text{REMOVEINCONSISTENT}(\text{Parent}(X_j), X_j)$
3. For j from 1 to n , assign X_j consistently with $\text{Parent}(X_j)$

- Once we have a **directed arc-consistent graph**, we can just march down the list of variables and choose any remaining value.
- Since each edge from a parent to its child is **arc-consistent**, we know that for any value we choose for the parent, there will be a valid value left to choose for the child.
- That means we won't have to backtrack; we can move linearly through the variables.

Nearly tree-structured CSPs

Conditioning: instantiate a variable, prune its neighbors' domains



Cutset conditioning: instantiate (in all ways) a set of variables such that the remaining constraint graph is a tree

Cutset size c \Rightarrow runtime $O(d^c \cdot (n - c)d^2)$, very fast for small c

Cycle Cutset Algorithm

1. Choose a subset S of the CSP's variables such that the constraint graph becomes a tree after removal of S . is called a cycle cutset.
2. For each possible assignment to the variables in S that satisfies all constraints on S ,
 - (a) remove from the domains of the remaining variables any values that are inconsistent with the assignment for S , and
 - (b) if the remaining CSP has a solution, return it together with the assignment for S

Iterative algorithms 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., hillclimb with $h(n)$ = total number of violated constraints

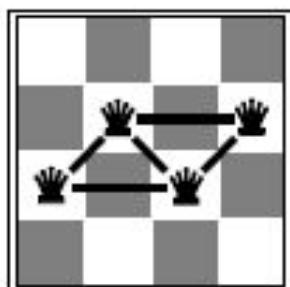
Example: 4-Queens

States: 4 queens in 4 columns ($4^4 = 256$ states)

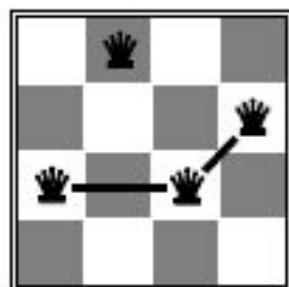
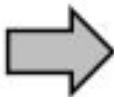
Operators: move queen in column

Goal test: no attacks

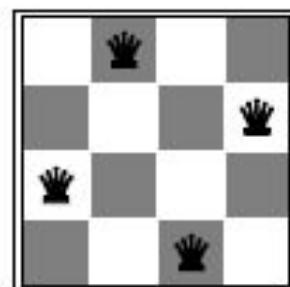
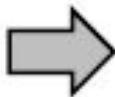
Evaluation: $h(n)$ = number of attacks



$h = 5$



$h = 2$



$h = 0$

Summary

CSPs are a special kind of problem:

states defined by values of a fixed set of variables
goal test defined by constraints on variable values

Backtracking = depth-first search with one variable assigned per node

Variable ordering and value selection heuristics help significantly

Forward checking prevents assignments that guarantee later failure

Constraint propagation (e.g., arc consistency) does additional work to constrain values and detect inconsistencies

The CSP representation allows analysis of problem structure

Tree-structured CSPs can be solved in linear time

Iterative min-conflicts is usually effective in practice

Constraint Satisfaction Problem

Constraint Satisfaction Problem (CSP)

- A problem is solved when each variable has a value that satisfies all the constraints on the variable. A problem described this way is called a **constraint satisfaction problem**, or CSP.
- The main idea is to eliminate large portions of the search space all at once by identifying variable/value combinations that violate the constraints.

Definition of CSP

- Constraint satisfaction problem consists of three components, X, D, and C:
 - X is a set of variables, $\{X_1, \dots, X_n\}$.
 - D is a set of domains, $\{D_1, \dots, D_n\}$, one for each variable.
 - C is a set of constraints that specify allowable combinations of values.
- A domain, D, consists of a set of allowable values, $\{v_1, \dots, v_n\}$, for variable X;
 - For example, a Boolean variable would have the domain {true, false}.
- Each constraint C_j consists of a pair (scope, rel),
 - where scope is a tuple of variables that participate in the constraint
 - rel is a relation that defines the values that those variables can take on.

- A relation can be represented as an explicit set of all tuples of values that satisfy the constraint, or as a function that can compute whether a tuple is a member of the relation.
 - For example, if X_1 and X_2 both have the domain $\{1,2,3\}$, then the constraint saying that X_1 ; must be greater than X_2 can be written as,
 $((X_1, X_2), \{(3,1), (3,2), (2,1)\})$ or as $((X_1, X_2), X_1 > X_2)$.

- CSPs deal with assignments of values to variables, $\{X_i = v_i, X_j = v_j, \dots\}$.
- An assignment that does not violate any constraints is called a **consistent or legal assignment**.
- A **complete assignment** is one in which every variable is assigned a value, and a **solution** to a CSP is a consistent, complete assignment.
- A **partial assignment** is one that leaves some variables unassigned, and a partial solution is a partial assignment that is **consistent**.

Example problem: Map coloring

- A map of Australia showing each of its states and territories.
- We are given the task of coloring each region either red, green, or blue in such a way that no two neighboring regions have the same color.



Example problem: Map coloring

- We define the variables to be the regions:

$$X = \{WA, NT, Q, NSW, V, SA, T\}.$$

- The domain of every variable is the set

$$D_i = \{\text{red, green, blue}\}.$$

- The constraints require neighboring regions to have distinct colors.
- Since there are nine places where regions border, there are nine constraints:

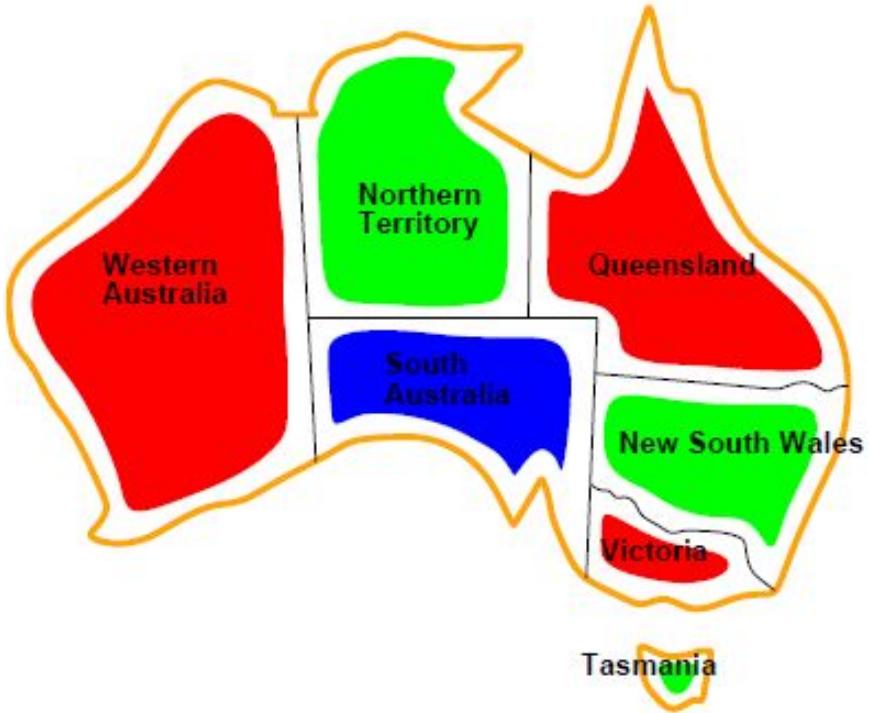
$$C = \{SA \neq WA, SA \neq NT, SA \neq Q, SA \neq NSW, SA \neq V, \\ WA \neq NT, NT \neq Q, Q \neq NSW, NSW \neq V\}.$$

Example problem: Map coloring

- Here we are using abbreviations; $SA \neq WA$ is a shortcut for $((SA, WA), SA \neq WA)$,
- Where $SA \neq WA$ can be fully enumerated in turn as
 $\{(red, green), (red, blue), (green, red), (green, blue),$
 $(blue, red), (blue, green)\}$.
- Solutions to this problem?

Example problem: Map coloring

- There are many possible solutions to this problem, such as
 $\{WA = \text{red}, NT = \text{green}, Q = \text{red}, NSW = \text{green}, V = \text{red}, SA = \text{blue}, T = \text{red}\}.$

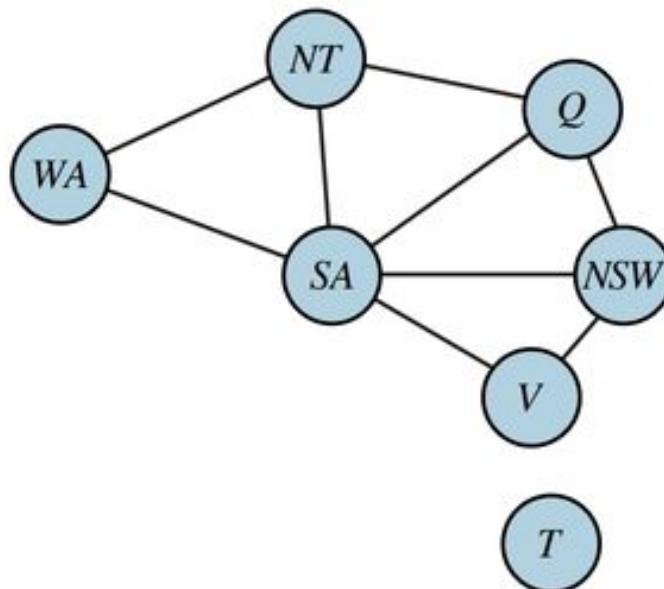


Solutions are assignments satisfying all constraints, e.g.,

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

Example problem: Map coloring

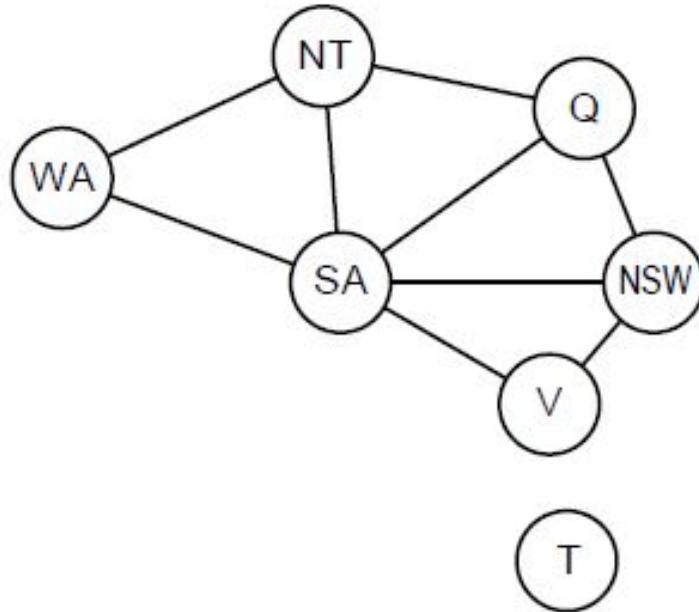
- It can be helpful to visualize a CSP as a constraint graph, as shown in Figure.
- The nodes of the graph correspond to variables of the problem, and an edge connects any two variables that participate in a constraint.



Constraint graph

Binary CSP: each constraint relates at most two variables

Constraint graph: nodes are variables, arcs show constraints



General-purpose CSP algorithms use the graph structure to speed up search. E.g., Tasmania is an independent subproblem!

Types of CSP

Discrete variables

finite domains; size $d \Rightarrow O(d^n)$ complete assignments

- ◊ e.g., Boolean CSPs, incl. Boolean satisfiability (NP-complete)

infinite domains (integers, strings, etc.)

- ◊ e.g., job scheduling, variables are start/end days for each job
- ◊ need a constraint language, e.g., $StartJob_1 + 5 \leq StartJob_3$
- ◊ linear constraints solvable, nonlinear undecidable

Continuous variables

- ◊ e.g., start/end times for Hubble Telescope observations
- ◊ linear constraints solvable in poly time by LP methods

Varieties of constraints

Unary constraints involve a single variable,

e.g., $SA \neq green$

Binary constraints involve pairs of variables,

e.g., $SA \neq WA$

Higher-order constraints involve 3 or more variables,

e.g., cryptarithmetic column constraints

Preferences (soft constraints), e.g., red is better than $green$

often representable by a cost for each variable assignment

→ constrained optimization problems

Cryptarithmetic Problem

- Cryptarithmetic Problem is a type of constraint satisfaction problem where the game is about digits and its unique replacement either with alphabets or other symbols.
- In **cryptarithmetic problem**, the digits (0-9) get substituted by some possible alphabets or symbols.
- The task in cryptarithmetic problem is to substitute each digit with an alphabet to get the result arithmetically correct.

Rules and constraints

- There should be a unique digit to be replaced with a unique alphabet.
- The result should satisfy the predefined arithmetic rules, i.e., $2+2=4$, nothing else.
- Digits should be from **0-9** only.
- The problem can be solved from both sides, i.e., **lefthand side (L.H.S)**, or **righthand side (R.H.S)**

Example-1

BASE

+BALL

GAMES

Example-1

$$\begin{array}{r} \text{B A S E} \\ + \text{B A L L} \\ \hline \text{G A M E S} \end{array}$$

Variables: {B,A,S,E,L,G,M}
Domain:{0,1,2,3,4,5,6,7,8,9}
Constraint: AllDiffz(B,A,S,E,L,G,M)

Example-1

BASE
+ **BALL** -

GAMES

B	5
B	5
G A	10
G	1
A	0

Example-1

B A S E
+ **B A L L** -
G A M E S

B	6
B	6
G A	12
G	1
A	2

Example-1

BASE
+ **BALL** -

GAMES

Cr	0	Cr	0
B	6	A	2
B	6	A	2
GA	12	M	4
G	1		
A	2		

Example-1

BASE
+ **BALL** -
GAMES

Cr	0	Cr	0	Cr	
B	6	A	2	S	3
B	6	A	2	L	5
GA	12	M	4	E	8
G	1				
A	2				

Example-1

BASE
+ **BALL** -
GAMES

Cr	0	Cr	0	C r		Cr	
B	6	A	2	S	3	E	8
B	6	A	2	L	5	L	5
GA	12	M	4	E	8	S	
G	1						
A	2						

Example-1 Solution

BASE

+BALL

GAMES

Cr	0	Cr	0	Cr		Cr	
B	7	A		S		E	
B	7	A		L		L	
GA	14	M		E		S	
G	1						
A	4						

Example-1 Solution

BASE

+BALL

GAMES

Cr	0	Cr	0	Cr		Cr	
B	7	A	4	S		E	
B	7	A	4	L		L	
GA	14	M	8	E		S	
G	1						
A	4						

Example-1 Solution

BASE

+BALL

GAMES

Cr	0	Cr	1	Cr		Cr	
B	7	A	4	S	8	E	
B	7	A	4	L	5	L	
GA	14	M	9	E	3	S	
G	1						
A	4						

Example-1 Solution

BASE

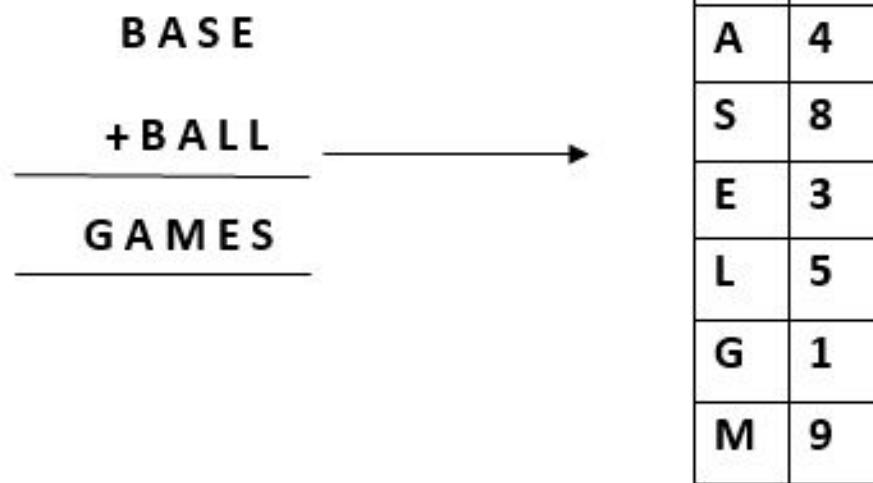
+BALL

GAMES

Cr	0	Cr	1	Cr	0	Cr	0
B	7	A	4	S	8	E	3
B	7	A	4	L	5	L	5
GA	14	M	9	E	3	S	8
G	1						
A	4						

BASE
+ **BALL**

GAMES



B	7
A	4
S	8
E	3
L	5
G	1
M	9

S E N D

+ M O R E

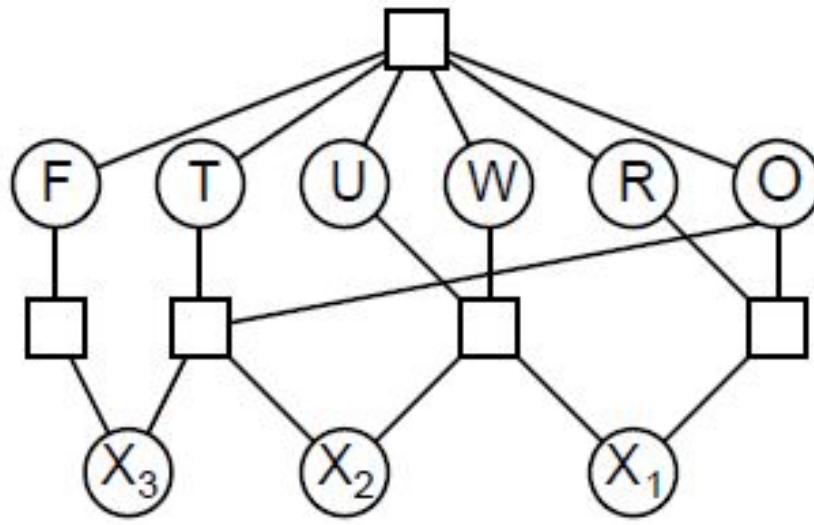
M O N E Y

S E N D
+ M O R E
—
M O N E Y
—

Cr		Cr		Cr		Cr	
S		E		N		D	
M		O		R		E	
MO		N		E		Y	
M							
O							

Example: Cryptarithmetic

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



Variables: $F \ T \ U \ W \ R \ O \ X_1 \ X_2 \ X_3$

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

Constraints

$\text{alldiff}(F, T, U, W, R, O)$

$O + O = R + 10 \cdot X_1$, etc.

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

<i>Cr</i>	0	<i>Cr</i>	0	<i>Cr</i>	0
T	7	W	3	O	4
T	7	W	3	O	4
FO	14	U	6	R	8
F	1				
O	4				

Real-world CSPs

Assignment problems

e.g., who teaches what class

Timetabling problems

e.g., which class is offered when and where?

Hardware configuration

Spreadsheets

Transportation scheduling

Factory scheduling

Floorplanning

Notice that many real-world problems involve real-valued variables

Standard search formulation (incremental)

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

States are defined by the values assigned so far

- ◊ **Initial state:** the empty assignment, {}
- ◊ **Successor function:** assign a value to an unassigned variable that does not conflict with current assignment.
⇒ fail if no legal assignments (not fixable!)
- ◊ **Goal test:** the current assignment is complete

- 1) This is the same for all CSPs! 😊
- 2) Every solution appears at depth n with n variables
⇒ use depth-first search
- 3) Path is irrelevant, so can also use complete-state formulation
- 4) $b = (n - \ell)d$ at depth ℓ , hence $n!d^n$ leaves!!!! 😞