

CSC384

Introduction to Artificial

Intelligence

CSC384: Intro to Artificial Intelligence

Instructor: Fahiem Bacchus

- Office D.L. Pratt, Room 398B
- Office Hours: Wednesday 3:30pm to 4:30, Thursday 11:00 am to 12 noon (or by appointment).

Lectures/Tutorials:

- Lectures
 - Tuesday and Thursday 1:00—2:00 pm in Room MP 102
- Tutorial
 - Thursday 2:00pm in Room MP 103

CSC384: Intro to Artificial Intelligence

Notes:

You are responsible for all material covered in either tutorials or lectures.

Sometimes Tutorials will cover new material, e.g., specific examples or elaborations of lecture material.

CSC384: Intro to Artificial Intelligence

Important Dates (more might be announced later).

Thursday 29th Oct. Midterm. Note $\frac{1}{2}$ of the class will write at 1:00 the other $\frac{1}{2}$ will write at 2:00.

Tuesday 10th Nov and Thursday 12th Nov. No classes as this is reading week.

CSC384: Reference Materials

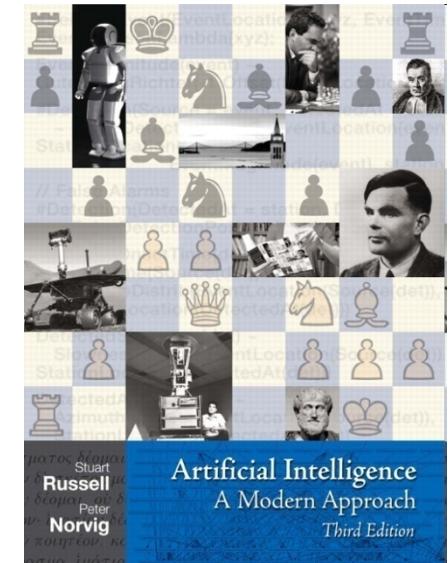
Recommended Textbook:

Artificial Intelligence: A Modern Approach

Stuart Russell and Peter Norvig

3rd Edition, 2009

- Recommended but not required.
- Older editions are also useable---but you will have to search the text for the relevant sections
- Sections most related to the lecture material will be indicated in the slides.
- <http://aima.cs.berkeley.edu/>



CSC384: Reference Materials

Alternate Book:

Computational Intelligence: A Logical Approach by David Poole and Alan Mackworth.

- Complete book is available on line!

<http://artint.info/>

Online Course:

- Various lectures are on line, e.g.,
<https://www.udacity.com/courses>
Introduction to Artificial Intelligence.

CSC384: Prerequisites

- Some probability (STA247H/STA255H/STA257H).
- Some knowledge of functional programming and logic programming is useful (CSC324H).
- This year the course will use **Python** in the assignments.
- If you don't have these prerequisites **you will be responsible** for learning any needed background material.
 - I will not have time to help you with that, and you will not be given any special consideration for not having had the proper background.

CSC384: Website

- **The course web site:**

<http://www.cs.toronto.edu/~fbacchus/csc384/>

- Primary source of more detailed information, announcements, etc.
- **Check the web site often.**
- Updates about assignments, clarifications etc. will be posted only on the web site.

- **The course bulletin board:**

<https://csc.cdf.toronto.edu>

Will not be monitored.

CSC384: How You Will be Graded

Course work:

1. 3 Assignments (mainly programming): **45%** in total equally divided.
2. Midterm Exam worth **15%**
3. A Final Exam (3hrs) worth **40%**

You need a minimum of 40% on the Final to pass the course

Please note. I do not accept late assignments.

You get zero for anything past the due date, unless you have a documented medical excuse (you must hand in an official **verification of student illness or injury** form

<http://www.illnessverification.utoronto.ca>

Plagiarism

- See
<http://www.cs.toronto.edu/~fpitt/documents/plagiarism.html>
for the meaning of plagiarism, how to avoid it, and the U of T policies about it.
- All assignments are to be done individually.
- You can discuss the assignments with other students, but you should not give your code (or parts of your code) to other students. You should not look at another student's code until after you have handed in your assignment (and the due date is past).
- Plagiarism has occurred in the past in this class and it has had very negative consequences for the students involved.
- Because 60% of the course mark is based on handed in work, we will be very diligent about detecting plagiarism.

CSC384: Email Policy

- I don't answer questions about course content or the assignments by email.
- I will read short and to the point email.
- Come to my office hours, talk to me before or after class
- If you have an unavoidable scheduling conflict we can arrange a mutually acceptable alternative meeting time.

CSC384

Intro to Artificial Intelligence

Artificial Intelligence

A branch of Computer Science.
Examines how we can achieve intelligent
behaviour through computation.

What is intelligence?



Are these Intelligent?

What is intelligence?



What about these?

What is Intelligence?

- Webster says:
 - The capacity to acquire and apply knowledge.
 - The faculty of thought and reason.
 - ...
- What features/abilities do humans (animals/animate objects) have that you think are indicative or characteristic of intelligence?
- *Abstract concepts, mathematics, language, problem solving, memory, logical reasoning, planning ahead, emotions, morality, ability to learn/adapt, etc...*

Artificial Intelligence

Studies how to achieve intelligent behavior through computational means.

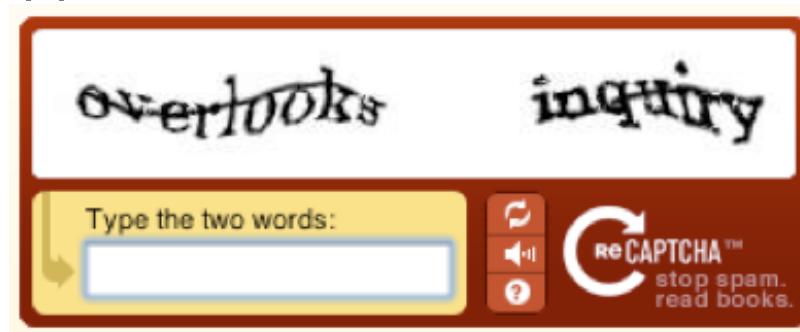
This makes AI a branch of Computer Science

Why do we think that intelligence can be captured through computation?

Modeling the processing that our brains do as computation has proved to be successful. Hence, human intelligence can arguably be best modeled as a computational process.

Classical Test of (Human) Intelligence

- The Turing Test:
 - A human interrogator. Communicates with a hidden subject that is either a computer system or a human.
If the human interrogator cannot reliably decide whether or not the subject is a computer, the computer is said to have passed the Turing test.
- Weak Turing type tests:



See Luis von Ahn, Manuel Blum, Nicholas Hopper, and John Langford.
CAPTCHA: Using Hard AI Problems for Security. In Eurocrypt.

Human Intelligence

- Turing provided some very persuasive arguments that a system passing the Turing test ***is intelligent.***
 - We can only really say it ***behaves like a human***
 - Nothing guarantees that it thinks like a human
- The Turing test does not provide much traction on the question of how to actually build an intelligent system.

Human Intelligence

- Recently some claims have been made of AI systems that can pass the Turing Test.
- However, these systems operate on subterfuge, and were able to convince a rather naïve jury that they were human like.
- The main technique used is obfuscation...rather than answering questions the system changed the topic!
- This is not what Turing described in his Turing Test

Human Intelligence

- In general there are various reasons why trying to mimic humans might **not** be the best approach to AI:
 - Computers and Humans have a very different architecture with quite different abilities.
 - Numerical computations
 - Visual and sensory processing
 - Massively and slow parallel vs. fast serial

	Computer	Human Brain
Computational Units	8 CPUs, 10^{10} gates	10^{11} neurons
Storage Units	10^{10} bits RAM 10^{13} bits disk	10^{11} neurons 10^{14} synapses
Cycle time	10^{-9} sec	10^{-3} sec
Bandwidth	10^{10} bits/sec	10^{14} bits/sec
Memory updates/sec	10^{10}	10^{14}

Human Intelligence

- But more importantly, we know very little about how the human brain performs its higher level processes. Hence, this point of view provides very little information from which a scientific understanding of these processes can be built.
- Nevertheless, Neuroscience has been very influential in some areas of AI. For example, in robotic sensing, vision processing, etc.
- Humans might not be best comparison?
 - Don't always make the best decisions
 - Computer intelligence can aid in our decision making

Rationality

- The alternative approach relies on the notion of **rationality**.
- Typically this is a precise formal notion of what it means to *do the right thing* in any particular circumstance. Provides
 - A precise mechanism for analyzing and understanding the properties of this ideal behavior we are trying to achieve.
 - A precise benchmark against which we can measure the behavior the systems we build.

Rationality

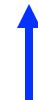
- Formal characterizations of rationality have come from diverse areas like logic (laws of thought) and economics (utility theory—how best to act under uncertainty, game theory how self-interested agents interact).
- There is no universal agreement about which notion of rationality is best, but since these notions are precise we can study them and give exact characterizations of their properties, good and bad.
- We'll focus on acting rationally
 - this has implications for thinking/reasoning

Computational Intelligence

- AI tries to understand and model intelligence as a computational process.
- Thus we try to construct systems whose computation achieves or approximates the desired notion of rationality.
- Hence AI is part of Computer Science.
 - Other areas interested in the study of intelligence lie in other areas or study, e.g., cognitive science which focuses on human intelligence. Such areas are very related, but their central focus tends to be different.

Four AI Definitions by Russell + Norvig

	Like humans	Not necessarily like humans
Think	Systems that think like humans	Systems that think rationally
Act	Systems that act like humans	Systems that act rationally <i>Our focus</i>



Cognitive Science

Subareas of AI

- Perception: vision, speech understanding, etc.
- Machine Learning, Neural networks
- Robotics
- Natural language processing
- Reasoning and decision making ← **OUR FOCUS**
 - **Knowledge representation**
 - **Reasoning** (logical, probabilistic)
 - **Decision making** (search, planning, decision theory)

Subareas of AI

- Many of the popular recent applications of AI in industry have been based on Machine Learning, e.g., voice recognition systems on your cell phone.
- We will not say much in this course about machine learning, although the last part of the course will introduce Bayes Nets a form of probabilistic graphical model.
- Probabilistic graphical models are fundamental in machine learning.

Subareas of AI

- Nor will we discuss Computer Vision nor Natural Language to any significant extent.
- All of these areas have developed a number of specialized theories and methods specific to the problems they study.
- The topics we will study here are fundamental techniques used in various AI systems, and often appear in advanced research in many other sub-areas of AI.
- In short, what we cover here is not sufficient for a deep understanding of AI, but it is a good start.

Further Courses in AI

- Perception: vision, speech understanding, etc.
 - CSC487H1 “Computational Vision”
 - CSC420H1 “Introduction to Image Understanding”
- Machine Learning, Neural networks
 - CSC321H “Introduction to Neural Networks and Machine Learning”
 - CSC411H “Machine Learning and Data Mining”
 - CSC412H1 “Uncertainty and Learning in Artificial Intelligence”
- Robotics
 - Engineering courses
- Natural language processing
 - CSC401H1 “Natural Language Computing”
 - CSC485H1 “Computational Linguistics”
- Reasoning and decision making
 - CSC486H1 “Knowledge Representation and Reasoning”
 - Builds on this course

What We Cover in CSC384

- Search (Chapter 3, 5, 6)

- Uninformed Search (3.4)
- Heuristic Search (3.5, 3.6)
- Game Tree Search (5)

- Knowledge Representation (Chapter 8, 9)

- First order logic for more general knowledge (8)
- Inference in First-Order Logic (9)

What We Cover in CSC384

- Classical Planning (Chapter 10)
 - Predicate representation of states
 - Planning Algorithms
- Quantifying Uncertainty and Probabilistic Reasoning (Chapter 13, 14, 16)
 - Uncertainties, Probabilities
 - Probabilistic Reasoning, Bayesian Networks

AI Successes

- **Games:** chess, checkers, poker, bridge, backgammon...
 - Search
- **Physical skills:** driving a car, flying a plane or helicopter, vacuuming...
 - Sensing, machine learning, planning, search, probabilistic reasoning
- **Language:** machine translation, speech recognition, character recognition, ...
 - Knowledge representation, machine learning, probabilistic reasoning
- **Vision:** face recognition, face detection, digital photographic processing, motion tracking, ...
- **Commerce and industry:** page rank for searching, fraud detection, trading on financial markets...
 - Search, machine learning, probabilistic reasoning

Recent AI Successes

- Darpa Grand Challenges
 - Goal: build a fully autonomous car that can drive a 240 km course in the Mojave desert
 - 2004: none went further than 12 km
 - 2005: 5 finished
 - 2007: Urban Challenge: 96 km urban course (former air force base) with obstacles, moving traffic, and traffic regulations: 6 finishers
 - 2011: Google testing its autonomous car for over 150,000 km on real roads
- 2011: IBM Watson competing successfully against two Jeopardy grand-champions

Degrees of Intelligence

- Building an intelligent system as capable as humans remains an elusive goal.
- However, systems have been built which exhibit various specialized degrees of intelligence.
- Formalisms and algorithmic ideas have been identified as being useful in the construction of these “intelligent” systems.
- Together these formalisms and algorithms form the foundation of our attempt to understand intelligence as a computational process.
- *In this course we will study some of these formalisms and see how they can be used to achieve various degrees of intelligence.*

Readings

- 1.1: What is AI?
- 2: Intelligent Agents
- Other interesting readings:
 - 1.2: Foundations
 - 1.3: History

CSC384

Intro to Artificial Intelligence

Artificial Intelligence

A branch of Computer Science.
Examines how we can achieve intelligent behaviour through computation.

What is intelligence?



Are these Intelligent?

What is intelligence?



What about these?

What is Intelligence?

- Webster says:
 - The capacity to acquire and apply knowledge.
 - The faculty of thought and reason.
 - ...
- What features/abilities do humans (animals/animate objects) have that you think are indicative or characteristic of intelligence?
- *Abstract concepts, mathematics, language, problem solving, memory, logical reasoning, planning ahead, emotions, morality, ability to learn/adapt, etc...*

CSC384, University of Toronto

5

Classical Test of (Human) Intelligence

- The Turing Test:
 - A human interrogator. Communicates with a hidden subject that is either a computer system or a human.
If the human interrogator cannot reliably decide whether or not the subject is a computer, the computer is said to have passed the Turing test.
- Weak Turing type tests:



See Luis von Ahn, Manuel Blum, Nicholas Hopper, and John Langford.
CAPTCHA: Using Hard AI Problems for Security. In Eurocrypt.

CSC384, University of Toronto

7

Artificial Intelligence

Studies how to achieve intelligent behavior through computational means.

This makes AI a branch of Computer Science

Why do we think that intelligence can be captured through computation?

Modeling the processing that our brains do as computation has proved to be successful. Hence, human intelligence can arguably be best modeled as a computational process.

CSC384, University of Toronto

6

Human Intelligence

- Turing provided some very persuasive arguments that a system passing the Turing test **is intelligent**.
 - We can only really say it **behaves like a human**
 - Nothing guarantees that it thinks like a human
- The Turing test does not provide much traction on the question of how to actually build an intelligent system.

CSC384, University of Toronto

8

Human Intelligence

- Recently some claims have been made of AI systems that can pass the Turing Test.
- However, these systems operate on subterfuge, and were able to convince a rather naïve jury that they were human like.
- The main technique used is obfuscation...rather than answering questions the system changed the topic!
- This is not what Turing described in his Turing Test

CSC384, University of Toronto

9

Human Intelligence

- But more importantly, we know very little about how the human brain performs its higher level processes. Hence, this point of view provides very little information from which a scientific understanding of these processes can be built.
- Nevertheless, Neuroscience has been very influential in some areas of AI. For example, in robotic sensing, vision processing, etc.
- Humans might not be best comparison?
 - Don't always make the best decisions
 - Computer intelligence can aid in our decision making

CSC384, University of Toronto

11

Human Intelligence

- In general there are various reasons why trying to mimic humans might **not** be the best approach to AI:
 - Computers and Humans have a very different architecture with quite different abilities.
 - Numerical computations
 - Visual and sensory processing
 - Massively and slow parallel vs. fast serial

	Computer	Human Brain
Computational Units	$8 \text{ CPUs}, 10^{10} \text{ gates}$	10^{11} neurons
Storage Units	10^{10} bits RAM $10^{13} \text{ bits disk}$	10^{11} neurons 10^{14} synapses
Cycle time	10^{-9} sec	10^{-3} sec
Bandwidth	10^{10} bits/sec	10^{14} bits/sec
Memory updates/sec	10^{10}	10^{14}

CSC384, University of Toronto

10

Rationality

- The alternative approach relies on the notion of **rationality**.
- Typically this is a precise formal notion of what it means to *do the right thing* in any particular circumstance. Provides
 - A precise mechanism for analyzing and understanding the properties of this ideal behavior we are trying to achieve.
 - A precise benchmark against which we can measure the behavior the systems we build.

CSC384, University of Toronto

12

Rationality

- Formal characterizations of rationality have come from diverse areas like logic (laws of thought) and economics (utility theory—how best to act under uncertainty, game theory how self-interested agents interact).
- There is no universal agreement about which notion of rationality is best, but since these notions are precise we can study them and give exact characterizations of their properties, good and bad.
- We'll focus on acting rationally
 - this has implications for thinking/reasoning

CSC384, University of Toronto

13

Computational Intelligence

- AI tries to understand and model intelligence as a computational process.
- Thus we try to construct systems whose computation achieves or approximates the desired notion of rationality.
- Hence AI is part of Computer Science.
 - Other areas interested in the study of intelligence lie in other areas or study, e.g., cognitive science which focuses on human intelligence. Such areas are very related, but their central focus tends to be different.

CSC384, University of Toronto

14

Four AI Definitions by Russell + Norvig

	Like humans	Not necessarily like humans
Think	Systems that think like humans	Systems that think rationally
Act	Systems that act like humans	Systems that act rationally


Cognitive Science

CSC384, University of Toronto

15

Subareas of AI

- Perception: vision, speech understanding, etc.
- Machine Learning, Neural networks
- Robotics
- Natural language processing
- Reasoning and decision making ← OUR FOCUS
 - Knowledge representation
 - Reasoning (logical, probabilistic)
 - Decision making (search, planning, decision theory)

CSC384, University of Toronto

16

Subareas of AI

- Many of the popular recent applications of AI in industry have been based on Machine Learning, e.g., voice recognition systems on your cell phone.
- We will not say much in this course about machine learning, although the last part of the course will introduce Bayes Nets a form of probabilistic graphical model.
- Probabilistic graphical models are fundamental in machine learning.

CSC384, University of Toronto

17

Subareas of AI

- Nor will we discuss Computer Vision nor Natural Language to any significant extent.
- All of these areas have developed a number of specialized theories and methods specific to the problems they study.
- The topics we will study here are fundamental techniques used in various AI systems, and often appear in advanced research in many other sub-areas of AI.
- In short, what we cover here is not sufficient for a deep understanding of AI, but it is a good start.

CSC384, University of Toronto

18

Further Courses in AI

- Perception: vision, speech understanding, etc.
 - CSC487H1 “Computational Vision”
 - CSC420H1 “Introduction to Image Understanding”
- Machine Learning, Neural networks
 - CSC321H “Introduction to Neural Networks and Machine Learning”
 - CSC411H “Machine Learning and Data Mining”
 - CSC412H1 “Uncertainty and Learning in Artificial Intelligence”
- Robotics
 - Engineering courses
- Natural language processing
 - CSC401H1 “Natural Language Computing”
 - CSC485H1 “Computational Linguistics”
- Reasoning and decision making
 - CSC486H1 “Knowledge Representation and Reasoning”
 - Builds on this course

CSC384, University of Toronto

19

What We Cover in CSC384

- Search ([Chapter 3, 5, 6](#))
 - Uninformed Search (3.4)
 - Heuristic Search (3.5, 3.6)
 - Game Tree Search (5)
- Knowledge Representation ([Chapter 8, 9](#))
 - First order logic for more general knowledge (8)
 - Inference in First-Order Logic (9)

CSC384, University of Toronto

20

What We Cover in CSC384

• Classical Planning ([Chapter 10](#))

- Predicate representation of states
- Planning Algorithms

• Quantifying Uncertainty and Probabilistic Reasoning ([Chapter 13, 14, 16](#))

- Uncertainties, Probabilities
- Probabilistic Reasoning, Bayesian Networks

Recent AI Successes

- Darpa Grand Challenges
 - Goal: build a fully autonomous car that can drive a 240 km course in the Mojave desert
 - 2004: none went further than 12 km
 - 2005: 5 finished
 - 2007: Urban Challenge: 96 km urban course (former air force base) with obstacles, moving traffic, and traffic regulations: 6 finishers
 - 2011: Google testing its autonomous car for over 150,000 km on real roads
- 2011: IBM Watson competing successfully against two Jeopardy grand-champions

AI Successes

- **Games:** chess, checkers, poker, bridge, backgammon...

- Search

- **Physical skills:** driving a car, flying a plane or helicopter, vacuuming...

- Sensing, machine learning, planning, search, probabilistic reasoning

- **Language:** machine translation, speech recognition, character recognition, ...

- Knowledge representation, machine learning, probabilistic reasoning

- **Vision:** face recognition, face detection, digital photographic processing, motion tracking, ...

- **Commerce and industry:** page rank for searching, fraud detection, trading on financial markets...

- Search, machine learning, probabilistic reasoning

Degrees of Intelligence

- Building an intelligent system as capable as humans remains an elusive goal.
- However, systems have been built which exhibit various specialized degrees of intelligence.
- Formalisms and algorithmic ideas have been identified as being useful in the construction of these “intelligent” systems.
- Together these formalisms and algorithms form the foundation of our attempt to understand intelligence as a computational process.
- *In this course we will study some of these formalisms and see how they can be used to achieve various degrees of intelligence.*

Readings

- 1.1: What is AI?
 - 2: Intelligent Agents
- Other interesting readings:
 - 1.2: Foundations
 - 1.3: History

Search

- One of the most basic techniques in AI
 - Underlying sub-module in most AI systems
- Can solve many problems that humans are not good at (achieving super-human performance)
- Very useful as a general algorithmic technique for solving many non-AI problems.

How do we plan our holiday?

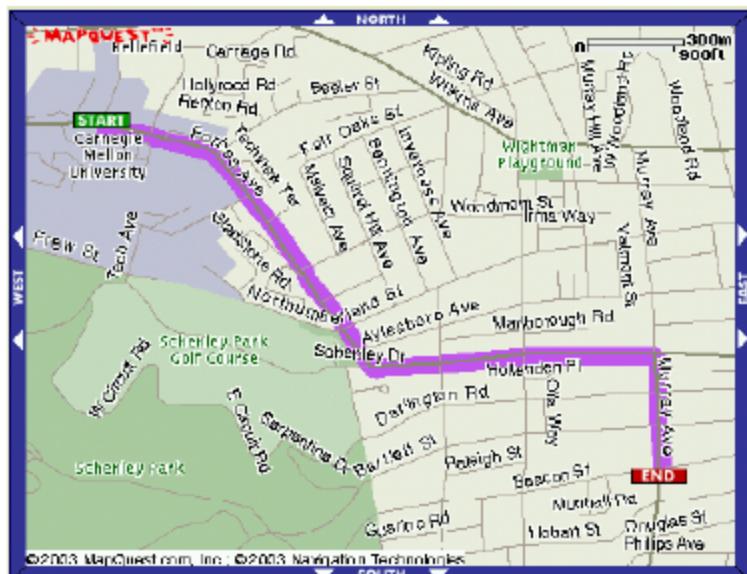
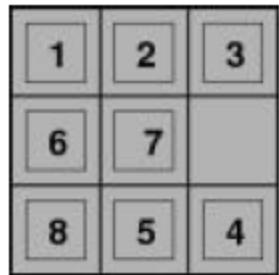
- We must take into account various preferences and constraints to develop a schedule.
- An important technique in developing such a schedule is “**hypothetical**” reasoning.
- Example: On holiday in England
 - Currently in Edinburgh
 - Flight leaves tomorrow from London
 - Need plan to get to your plane
 - If I take a 6 am train where will I be at 2 pm? Will I be still able to get to the airport on time?

How do we plan our holiday?

- This kind of hypothetical reasoning involves asking
 - what state will I be in after taking certain actions, or after certain sequences of events?
- From this we can reason about particular sequences of events or actions one should try to bring about to achieve a desirable state.
- Search is a computational method for capturing a particular version of this kind of reasoning.

Many problems can be solved by search:

Search Problems



Slide 7

Why Search?

- Successful
 - Success in game playing programs based on search.
 - Many other AI problems can be successfully solved by search.
- Practical
 - Many problems don't have specific algorithms for solving them. Casting as search problems is often the easiest way of solving them.
 - Search can also be useful in approximation (e.g., local search in optimization problems).
 - Problem specific heuristics provides search with a way of exploiting extra knowledge.
- Some critical aspects of intelligent behaviour, e.g., planning, can be naturally cast as search.

Limitations of Search

- There are many difficult questions that are not resolved by search. In particular, the whole question of how does an intelligent system formulate the problem it wants to solve as a search problem is not addressed by search.
- Search only shows how to solve the problem once we have it correctly formulated.

Search

- Formulating a problem as search problem (representation)
- Heuristic Search
- Game-Tree-Search
- Readings
 - Introduction: Chapter 3.1 – 3.3
 - Uninformed Search: Chapter 3.4
 - Heuristic Search: Chapters 3.5, 3.6

Representing a problem: The Formalism

To formulate a problem as a search problem we need the following components:

1. **STATE SPACE:** Formulate a **state space** over which we perform search. The state space is a way of representing in a computer the states of the real problem.
2. **ACTIONS or STATE SPACE Transitions:** Formulate **actions** that allow one to move between different states. The actions reflect the actions one can take in the real problem but operate on the state space instead.

Representing a problem: The Formalism

To formulate a problem as a search problem we need the following components:

3. **INITIAL or START STATE and GOAL:** Identify the **initial state** that best represents the starting conditions, and the goal or condition one wants to achieve.
4. **Heuristics:** Formulate various **heuristics** to help guide the search process.

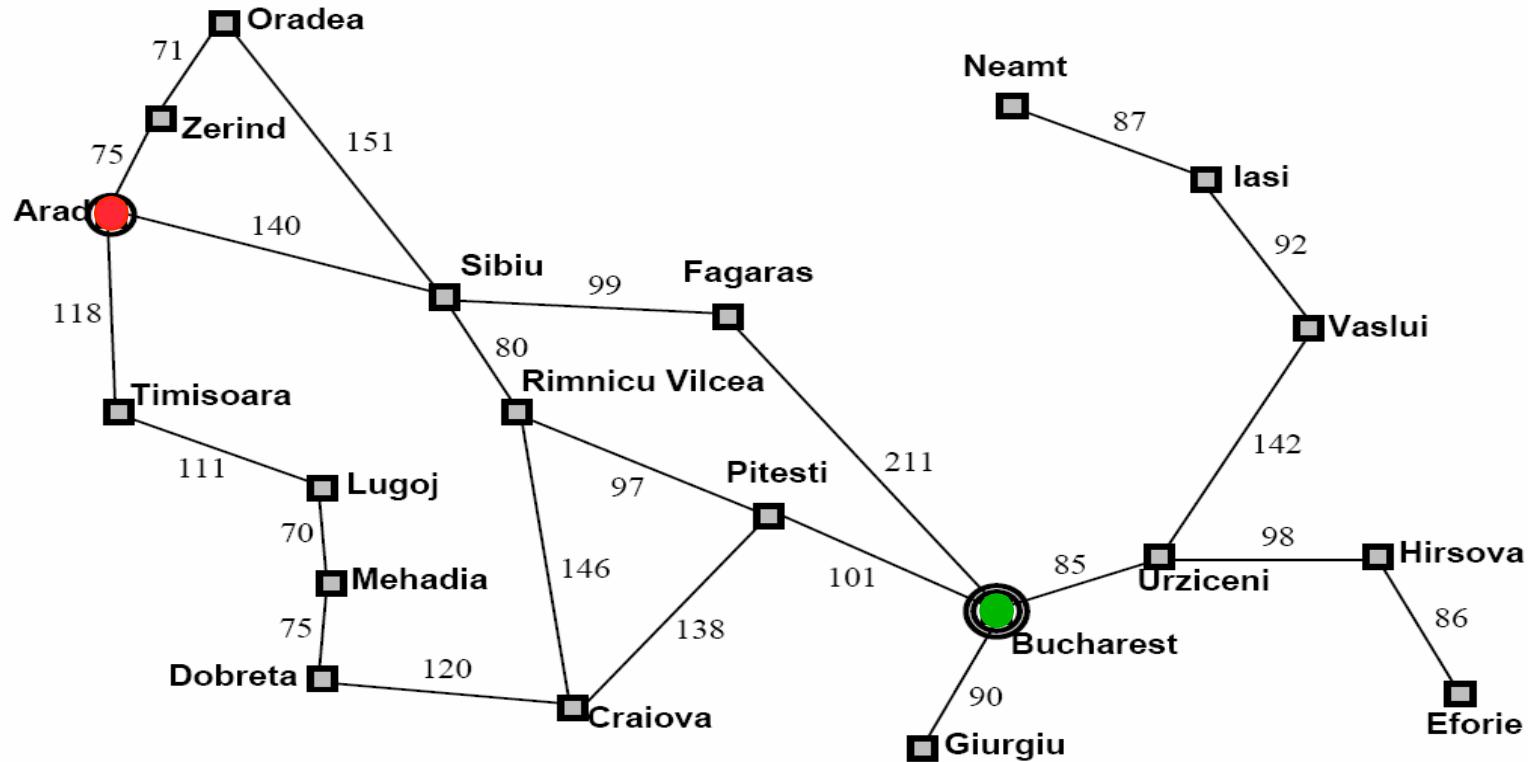
The Formalism

Once the problem has been formulated as a state space search, various algorithms can be utilized to solve the problem.

- A solution to the problem will be a sequence of actions/moves that can transform your current state into a state where your desired condition holds.

Example 1: Romania Travel.

Currently in **Arad**, need to get to **Bucharest** by tomorrow to catch a flight. What is the **State Space**?



Example 1.

- State space.
 - States: the various cities you could be located in.
 - Our abstraction: we are ignoring the low level details of driving, states where you are on the road between cities, etc.
 - Actions: drive between neighboring cities.
 - Initial state: in Arad
 - Desired condition (Goal): be in a state where you are in Bucharest. (How many states satisfy this condition?)
- Solution will be the route, the sequence of cities to travel through to get to Bucharest.

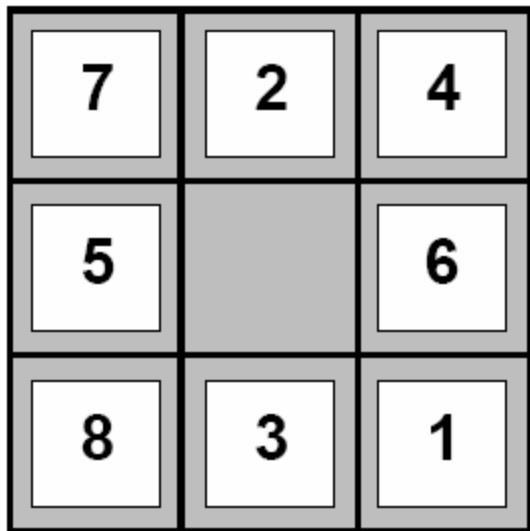
Example 2.

- Water Jugs
 - We have a 3 gallon (liter) jug and a 4 gallon jug. We can fill either jug to the top from a tap, we can empty either jug, or we can pour one jug into the other (at least until the other jug is full).
 - **States**: pairs of numbers (gal3 , gal4) where gal3 is the number of gallons in the 3 gallon jug, and gal4 is the number of gallons in the 4 gallon jug.
 - **Actions**: Empty-3-Gallon, Empty-4-Gallon, Fill-3-Gallon, Fill-4-Gallon, Pour-3-into-4, Pour 4-into-3.
 - **Initial state**: Various, e.g., $(0,0)$
 - **Desired condition (Goal)**: Various, e.g., $(0,2)$ or $(*, 3)$ where $*$ means we don't care.

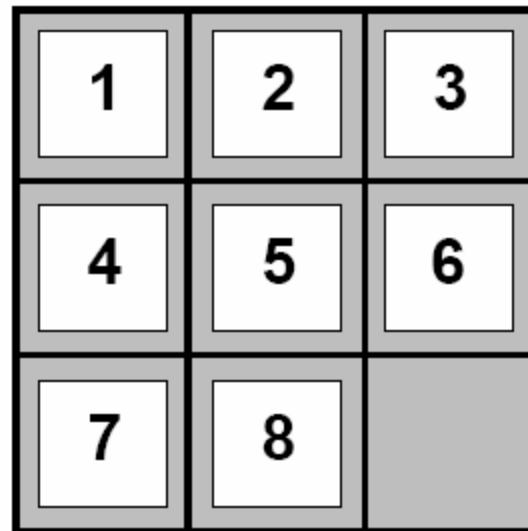
Example 2.

- Water Jugs
 - If we start off with gal3 and gal4 as integer, can only reach integer values.
 - Some values, e.g., $(1,2)$ are not reachable from some initial state, e.g., $(0,0)$.
 - Some actions are no-ops. They do not change the state, e.g.,
 - $(0,0) \rightarrow \text{Empty-3-Gallon} \rightarrow (0,0)$

Example 3. The 8-Puzzle



Start State



Goal State

Rule: Can slide a tile into the blank spot.

Alternative view: move the blank spot around.

Example 3. The 8-Puzzle

- State space.
 - **States**: The different configurations of the tiles. How many different states?
 - **Actions**: Moving the blank up, down, left, right. Can every action be performed in every state?
 - **Initial state**: e.g., state shown on previous slide.
 - **Desired condition (Goal)**: be in a state where the tiles are all in the positions shown on the previous slide.
- Solution will be a sequence of moves of the blank that transform the initial state to a goal state.

Example 3. The 8-Puzzle

- Although there are $9!$ different configurations of the tiles (362,880) in fact the state space is divided into two disjoint parts.
- Only when the blank is in the middle are all four actions possible.
- Our goal condition is satisfied by only a single state. But one could easily have a goal condition like
 - The 8 is in the upper left hand corner.
 - How many different states satisfy this goal?

More complex situations

- Perhaps actions lead to multiple states, e.g., flip a coin to obtain heads OR tails. Or we don't know for sure what the initial state is (prize is behind door 1, 2, or 3). Now we might want to consider how **likely** different states and action outcomes are.
- This leads to probabilistic models of the search space and different algorithms for solving the problem.
- Later we will see some techniques for reasoning under uncertainty.

Algorithms for Search

Inputs:

- a specified **initial state** (a specific world state)
- a **successor function** $S(x) = \{\text{set of states that can be reached from state } x \text{ via a single action}\}$.
- a **goal test** a function that can be applied to a state and returns true if the state satisfies the goal condition.
- An **action cost** function $C(x,a,y)$ which determines the cost of moving from state x to state y using action a . ($C(x,a,y) = \infty$ if a does not yield y from x). Note that different actions might generate the same move of $x \rightarrow y$.

Algorithms for Search

Output:

- a sequence of states leading from the initial state to a state satisfying the goal test.
- The sequence might be, optimal in cost for some algorithms, optimal in length for some algorithms, come with no optimality guarantees from other algorithms.

Algorithms for Search

Obtaining the action sequence.

- The set of successors of a state x might arise from different actions, e.g.,
 - $x \rightarrow a \rightarrow y$
 - $x \rightarrow b \rightarrow z$
- Successor function $S(x)$ yields a set of states that can be reached from x via **any** single action.
 - Rather than just return a set of states, we annotate these states by the action used to obtain them:
 - $S(x) = \{<y, a>, <z, b>\}$
 y via action a , z via action b .
 - $S(x) = \{<y, a>, <y, b>\}$
 y via action a , also y via alternative action b .

Template Search Algorithms

- The search space consists of **states** and actions that move between states.
- A **path** in the search space is a **sequence** of states connected by actions, $\langle s_0, s_1, s_2, \dots, s_k \rangle$, for every s_i and its successor s_{i+1} there must exist an action a_i that transitions s_i to s_{i+1} .
 - Alternately a path can be specified by
 - (a) an initial state s_0 , and
 - (b) a sequence of actions that are applied in turn starting from s_0 .
- The search algorithms perform search by examining alternate paths of the search space. The objects used in the algorithm are called **nodes**—each node contains a path.

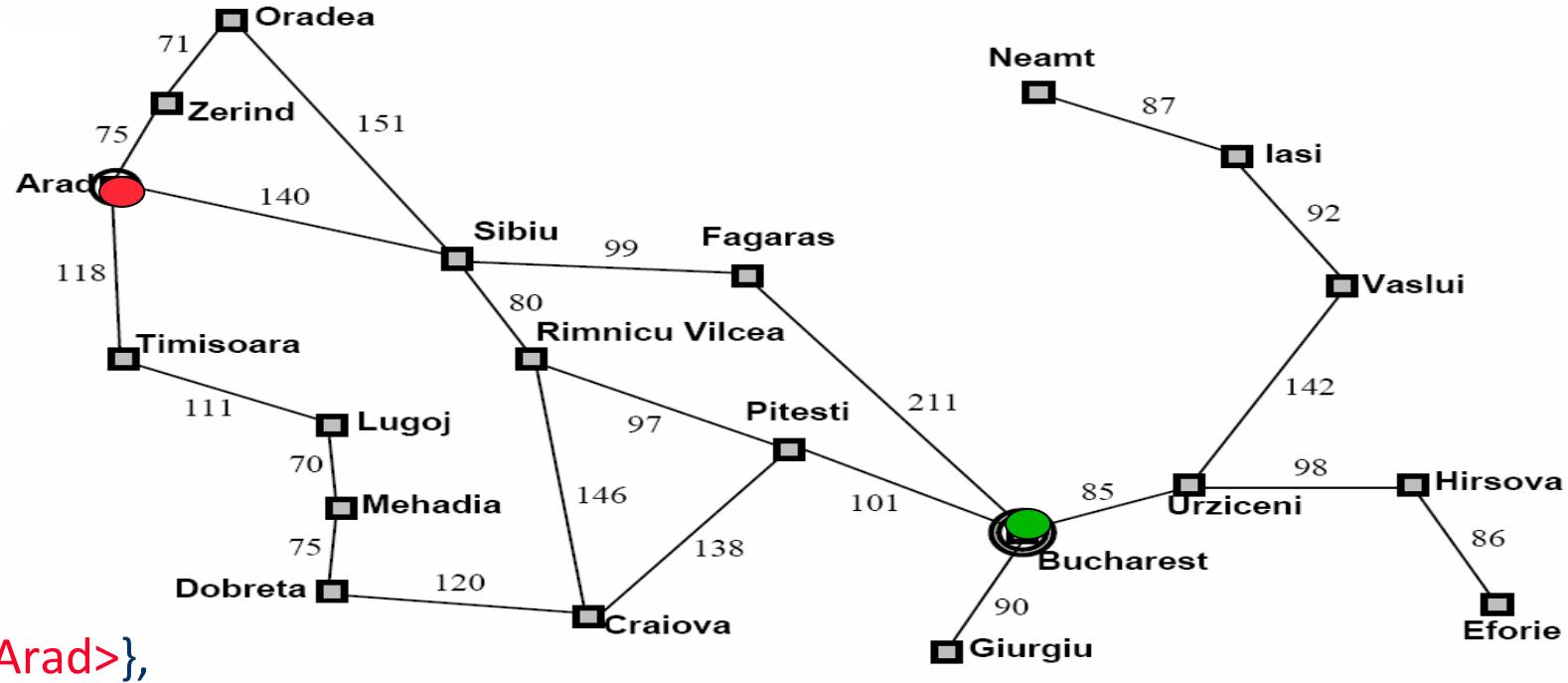
Template Algorithm for Search

- We maintain a set of **Frontier** nodes also called the **OPEN** set.
 - These nodes are paths in the search space that all start at the initial state.
- Initially we set $\text{OPEN} = \{\langle \text{Start State} \rangle\}$
 - The path (node) that starts and terminates at the start state.
- At each step we select a node n from OPEN . Let x be the state n terminates at. We check if x satisfies the goal, if not we add all extensions of n to OPEN (by finding all states in $S(x)$).

Template Algorithm for Search

```
Search(OPEN, Successors, Goal? )
  While(Open not EMPTY) {
    n = select node from OPEN
    Curr = terminal state of n
    If (Goal?(Curr)) return n.
    OPEN = (OPEN- {n}) Us ∈ Successors(Curr) <n,s>
      /* Note OPEN could grow or shrink */
  }
  return FAIL
```

When does OPEN get smaller in size?



{<Arad>},

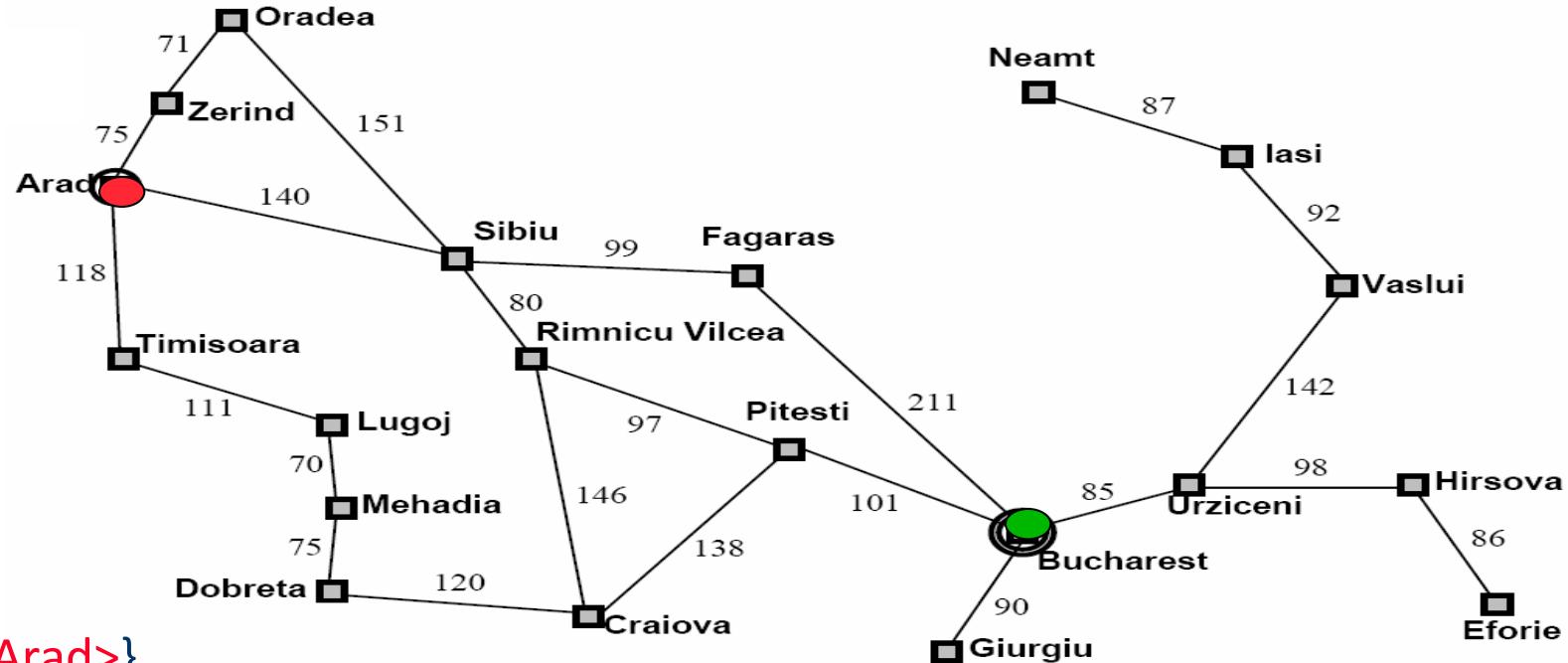
{<A,Z>, <A,T>, <A,S>},

{<A,Z>, <A,T>, <A,S,A>, <A,S,O>, <A,S,F>, <A,S,R>}

{<A,Z>, <A,T>, <A,S,A>, <A,S,O>, <A,S,R>, <A,S,F,S>, <A,S,F,B>}

Solution: Arad -> Sibiu -> Fagaras -> Bucharest

Cost: 140 + 99 + 211 = 450



{<Arad>},

{<A,Z>, <A,T>, <A, S>},

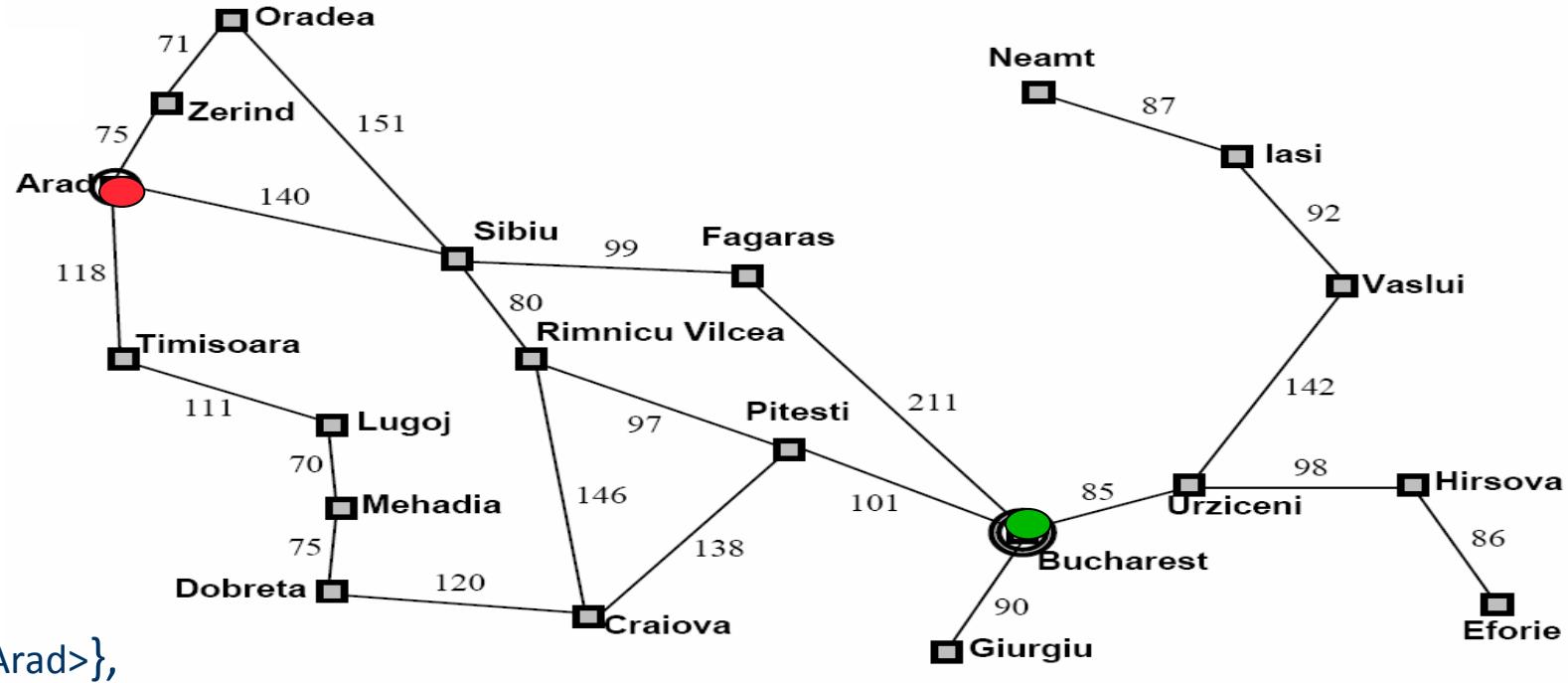
{<A,Z>, <A,T>, <A,S,A>, <A,S,O>, <A,S,F>, <A,S,R>}

{<A,Z>, <A,T>, <A,S,A>, <A,S,O>, <A,S,F>, <A,S,R,S>, <A,S,R,C>, <A,S,R,P>}

{<A,Z>, <A,T>, <A,S,A>, <A,S,O>, <A,S,F>, <A,S,R,S>, <A,S,R,C>, <A,S,R,P,R>, <A,S,R,P,C>, <A,S,R,P,B>}

Solution: Arad -> Sibiu -> Rimnicu Vilcea -> Pitesti -> Bucharest

Cost: 140 + 80 + 97 + 101 = 418



{<Arad>},

{<A,Z>, <A,T>, <A, S>},

{<A,Z>, <A,T>, **<A,S,A>**, <A,S,O>, <A,S,F>, <A,S,R>}

{<A,Z>, <A,T>, **<A,S,A>**, <A,S,O>, <A,S,R>, **<A,S,F,S>**, <A,S,F,B>}

.....

cycles can cause non-termination!

... we deal with this issue later

Selection Rule

The order paths are selected from OPEN has a critical effect on the operation of the search:

- Whether or not a solution is found
- The cost of the solution found.
- The time and space required by the search.

How to select the next path from OPEN?

All search techniques keep OPEN as an ordered set (e.g., a priority queue) and repeatedly execute:

- If OPEN is empty, terminate with failure.
 - Get the **next** path from OPEN.
 - If the path leads to a goal state, terminate with success.
 - Extend the path (i.e. generate the successor states of the terminal state of the path) and put the new paths in OPEN.
-
- How do we order the paths on OPEN?

Critical Properties of Search

- **Completeness:** will the search always find a solution if a solution exists?
- **Optimality:** will the search always find the least cost solution? (when actions have costs)
- **Time complexity:** what is the maximum number of nodes (paths) than can be expanded or generated?
- **Space complexity:** what is the maximum number of nodes (paths) that have to be stored in memory?

Uninformed Search Strategies

- These are strategies that adopt a fixed rule for selecting the next state to be expanded.
- The rule does not change irrespective of the search problem being solved.
- These strategies do not take into account any domain specific information about the particular search problem.
- Uninformed search techniques:
 - Breadth-First, Uniform-Cost, Depth-First, Depth-Limited, and Iterative-Deepening search

Breadth-First Search

Breadth-First Search

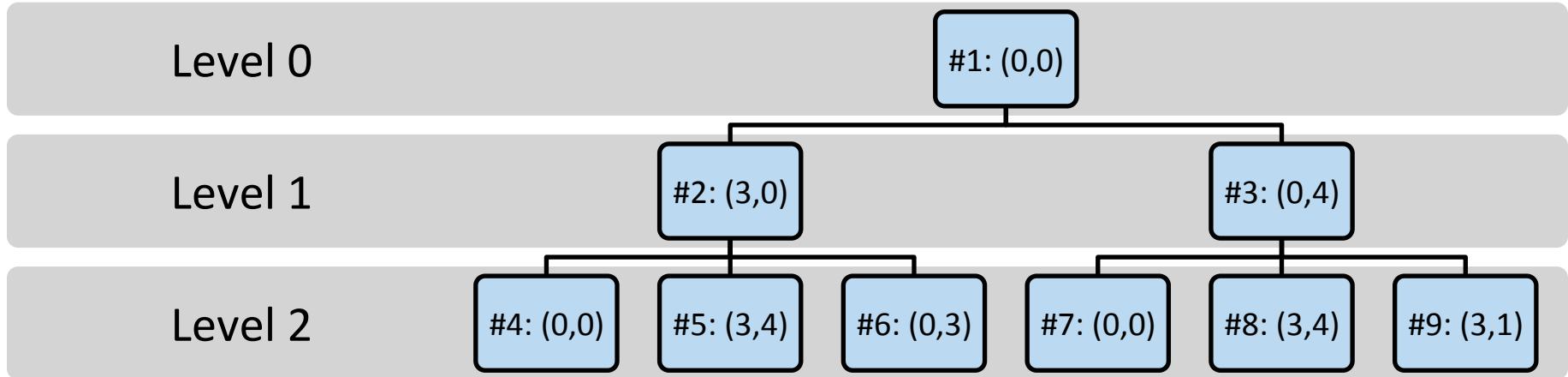
- Place the new paths that extend the current path at the **end** of OPEN.

WaterJugs. Start = $(0,0)$, Goal = $(* , 2)$

Green = Newly Added.

1. OPEN = { $\langle(0,0)\rangle$ }
2. OPEN = { $\langle(0,0),(3,0)\rangle, \langle(0,0),(0,4)\rangle$ }
3. OPEN = { $\langle(0,0),(0,4)\rangle, \langle(0,0),(3,0),(0,0)\rangle,$
 $\langle(0,0),(3,0),(3,4)\rangle, \langle(0,0),(3,0),(0,3)\rangle$ }
4. OPEN = { $\langle(0,0),(3,0),(0,0)\rangle, \langle(0,0),(3,0),(3,4)\rangle,$
 $\langle(0,0),(3,0),(0,3)\rangle, \langle(0,0),(0,4),(0,0)\rangle,$
 $\langle(0,0),(0,4),(3,4)\rangle, \langle(0,0),(0,4),(3,1)\rangle$ }

Breadth-First Search



- Above we indicate only the state that each nodes terminates at. The path represented by each node is the path from the root to that node.
- Breadth-First explores the search space level by level.

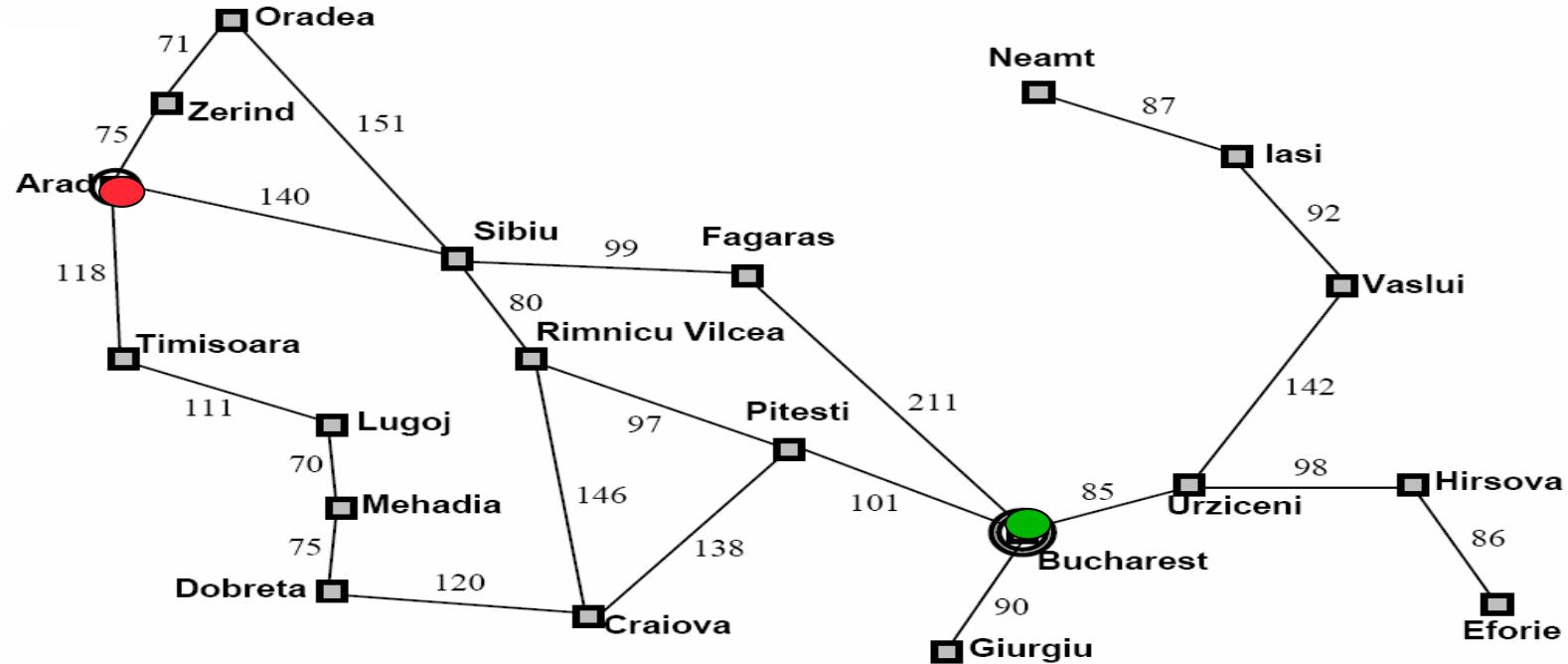
Breadth-First Properties

Completeness?

- The length of the path removed from OPEN is non-decreasing.
 - we replace each expanded node n with an extension of n .
 - **All** shorter paths are expanded prior before any longer path.
- Hence, eventually we must examine all paths of length d , and thus find a solution if one exists.

Optimality?

- By the above will find shortest length solution
 - least cost solution?
 - Not necessarily: shortest solution not always cheapest solution if actions have varying costs



Breadth first Solution: Arad \rightarrow Sibiu \rightarrow Fagaras \rightarrow Bucharest

Cost: 140 + 99 + 211 = 450

Lowest cost Solution: Arad \rightarrow Sibiu \rightarrow Rimnicu Vilcea \rightarrow Pitesti \rightarrow Bucharest

Cost: 140 + 80 + 97 + 101 = 418

Breadth-First Properties

Measuring time and space complexity.

- let b be the maximum number of successors of any node (maximal branching factor).
- let d be the depth of the shortest solution.
 - Root at depth 0 is a path of length 1
 - So $d = \text{length of path} - 1$

Time Complexity?

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

Breadth-First Properties

Space Complexity?

- $O(b^{d+1})$: If goal node is last node at level d , all of the successors of the other nodes will be on OPEN when the goal node is expanded $b(b^d - 1)$

Breadth-First Properties

Space complexity is a real problem.

- E.g., let $b = 10$, and say 100,000 nodes can be expanded per second and each node requires 100 bytes of storage:

Depth	Nodes	Time	Memory
1	1	0.01 millisec.	100 bytes
6	10^6	10 sec.	100 MB
8	10^8	17 min.	10 GB
9	10^9	3 hrs.	100 GB

- Typically run out of space before we run out of time in most applications.

Uniform-Cost Search

Uniform-Cost Search

- Keep OPEN ordered by increasing cost of the path.
- Always expand the least cost path.
- Identical to Breadth first if each action has the same cost.

Uniform-Cost Properties

Completeness?

- If each transition has costs $\geq \varepsilon > 0$.
- The previous argument used for breadth first search holds: the cost of the path represented by each node n chosen to be expanded must be non-decreasing.

Optimality?

- Finds optimal solution if each transition has cost $\geq \varepsilon > 0$.
 - Explores paths in the search space in increasing order of cost. So must find minimum cost path to a goal before finding any higher costs paths.

Uniform-Cost Search. Proof of Optimality

Let us prove Optimality more formally. We will reuse this argument later on when we examine Heuristic Search

Uniform-Cost Search. Proof of Optimality

Lemma 1.

Let $c(n)$ be the cost of node n on OPEN (cost of the path represented by n). If n_2 is expanded IMMEDIATELY after n_1 then

$$c(n_1) \leq c(n_2).$$

Proof: there are 2 cases:

- a. n_2 was on OPEN when n_1 was expanded:

We must have $c(n_1) \leq c(n_2)$ otherwise n_2 would have been selected for expansion rather than n_1

- b. n_2 was added to OPEN when n_1 was expanded

Now $c(n_1) < c(n_2)$ since the path represented by n_2 extends the path represented by n_1 and thus cost at least ϵ more.

Uniform-Cost Search. Proof of Optimality

Lemma 2.

When node n is expanded every path in the search space with cost strictly less than $c(n)$ has already been expanded.

Proof:

- Let $n_k = \langle \text{Start}, s_1, \dots, s_k \rangle$ be a path with cost less than $c(n)$. Let $n_0 = \langle \text{Start} \rangle$, $n_1 = \langle \text{Start}, s_1 \rangle$, $n_2 = \langle \text{Start}, s_1, s_2 \rangle$, ..., $n_i = \langle \text{Start}, s_1, \dots, s_i \rangle$, ..., $n_k = \langle \text{Start}, s_1, \dots, s_k \rangle$. Let n_i be *the last node in this sequence that has already been expanded by search*.
- So, n_{i+1} must still be on OPEN: it was added to open when n_i was expanded. Also $c(n_{i+1}) \leq c(n_k) < c(n)$: $c(n_{i+1})$ is a subpath of n_k we have assumed that $c(n_k)$ is $< c(n)$.
- But then uniform-cost would have expanded n_{i+1} not n .
- So every node n_i including n_k must already be expanded, i.e., this lower cost path has already been expanded.

Uniform-Cost Search. Proof of Optimality

Lemma 3.

The first time uniform-cost expands a node n terminating at state S , it has found the minimal cost path to S (it might later find other paths to S but none of them can be cheaper).

Proof:

- All cheaper paths have already been expanded, none of them terminated at S .
- All paths expanded after n will be at least as expensive, so no cheaper path to S can be found later.

So, when a path to a goal state is expanded the path must be optimal (lowest cost).

Uniform-Cost Properties

Time and Space Complexity?

- $O(b^{C^*/\varepsilon})$ where C^* is the cost of the optimal solution.
- There may be many paths with cost $\leq C^*$: there can be as many as b^d paths of length d in the worst case.

Paths with cost lower than C^* can be as long as C^*/ε (why no longer?), so might have $b^{C^*/\varepsilon}$ paths to explore before finding an optimal cost path.

Depth-First Search

Depth-First Search

- Place the new paths that extend the current path at the **front** of OPEN.

WaterJugs. Start = (0,0), Goal = (*,2)

Green = Newly Added.

1. OPEN = {<(0,0)>}
2. OPEN = {**<(0,0), (3,0)>**, **<(0,0), (0,4)>**}
3. OPEN = {**<(0,0),(3,0),(0,0)>**, **<(0,0),(3,0),(3,4)>**,
<(0,0),(3,0),(0,3)>, **<(0,4),(0,0)>**}
4. OPEN = {**<(0,0),(3,0),(0,0),(3,0)>**, **<(0,0),(3,0),(0,0),(0,4)>**
<(0,0), (3,0), (3,4)>, **<(0,0),(3,0),(0,3)>**,
<(0,0),(0,4)>}

Depth-First Search

Level 0

#1: (0,0)

Level 1

#2: (3,0)

(0,4)

Level 2

#3: (0,0)

(3,4)

(0,3)

Level 3

#4: (3, 0)

(0,4)

- Red nodes are backtrack points (these nodes remain on open).

Depth-First Properties

Completeness?

- Infinite paths? Cause incompleteness!
- Prune paths with cycles (duplicate states)
We get completeness if state space is finite

Optimality?

No!

Depth-First Properties

Time Complexity?

- $O(b^m)$ where m is the length of the longest path in the state space.
- Very bad if m is much larger than d (shortest path to a goal state), but if there are many solution paths it can be much faster than breadth first. (Can by good luck bump into a solution quickly).

Depth-First Properties

- Depth-First Backtrack Points = unexplored siblings of nodes along current path.
 - These are the nodes that remain on open after we extract a node to expand.

Space Complexity?

- $O(bm)$, linear space!
 - Only explore a single path at a time.
 - OPEN only contains the deepest node on the current path along with the **backtrack** points.
- A significant advantage of DFS

Depth-Limited Search

Depth Limited Search

- Breadth first has space problems. Depth first can run off down a very long (or infinite) path.

Depth limited search

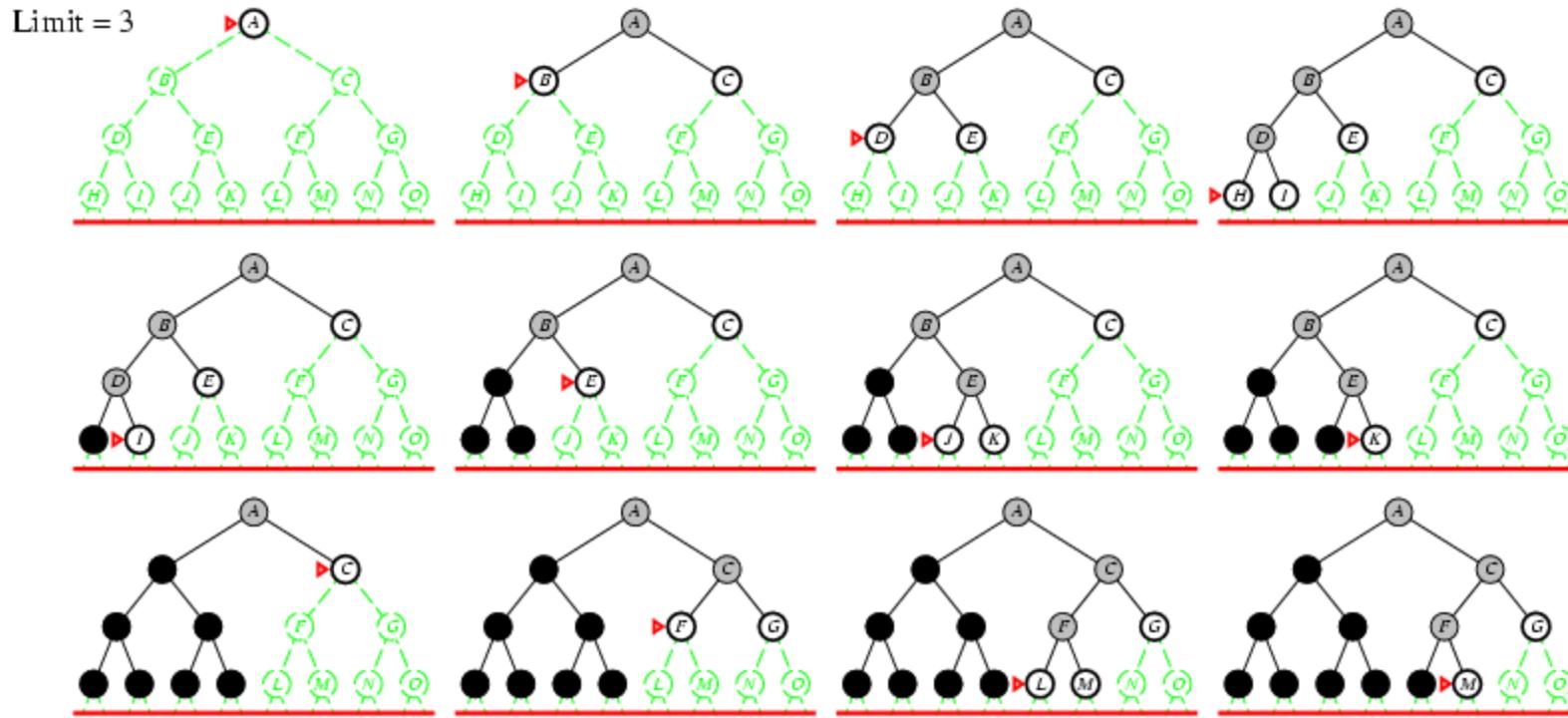
- Perform depth first search but only to a pre-specified depth limit D .
 - THE ROOT is at DEPTH 0. ROOT is a path of length 1.
 - No node representing a path of length more than $D+1$ is placed on OPEN.
 - We “truncate” the search by looking only at paths of length $D+1$ or less.
- Now infinite length paths are not a problem.
- But will only find a solution if a solution of $\text{DEPTH} \leq D$ exists.

Depth Limited Search

```
DLS(OPEN, Successors, Goal?) /* Call with OPEN = {<START>} */  
WHILE(OPEN not EMPTY) {  
    n= select first node from OPEN  
    Curr = terminal state of n  
    If(Goal?(Curr)) return n  
  
    If Depth(n) < D //Don't add successors if Depth(n) = D  
        OPEN = (OPEN- {n}) Us ∈ Successors(Curr) <n,s>  
    Else  
        OPEN = OPEN - {n}  
        CutOffOccured = TRUE.  
    }  
return FAIL
```

We will use CutOffOccured later.

Depth Limited Search Example



Iterative Deepening Search

Iterative Deepening Search

- Solve the problems of depth-first and breadth-first by extending depth limited search
- Starting at depth limit $L = 0$, we iteratively increase the depth limit, performing a depth limited search for each depth limit.
- Stop if a solution is found, or if the depth limited search failed without cutting off any nodes because of the depth limit.
 - If no nodes were cut off, the search examined all paths in the state space and found no solution → no solution exists.

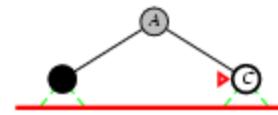
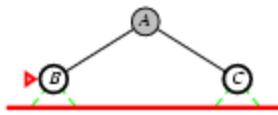
Iterative Deepening Search Example

Limit = 0



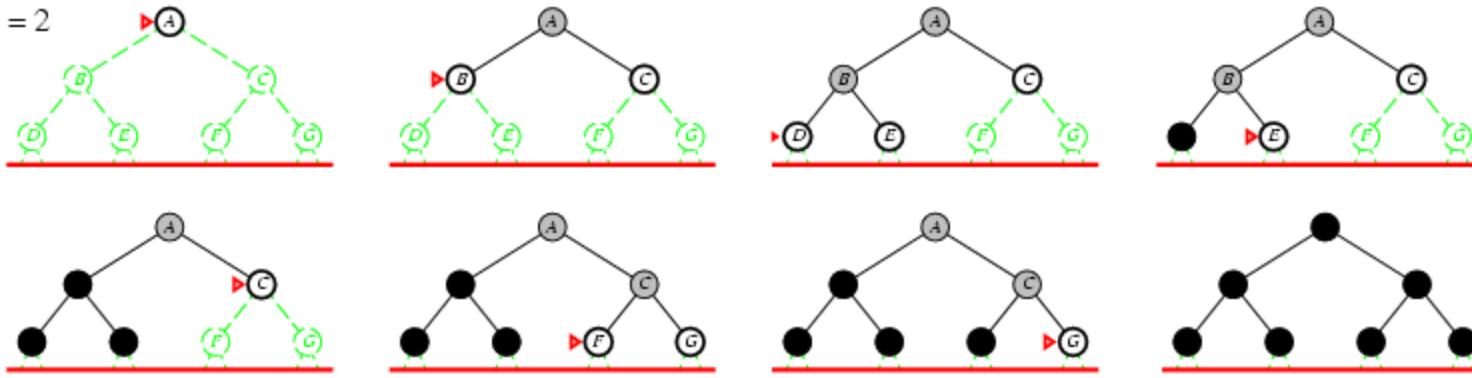
Iterative Deepening Search Example

Limit = 1

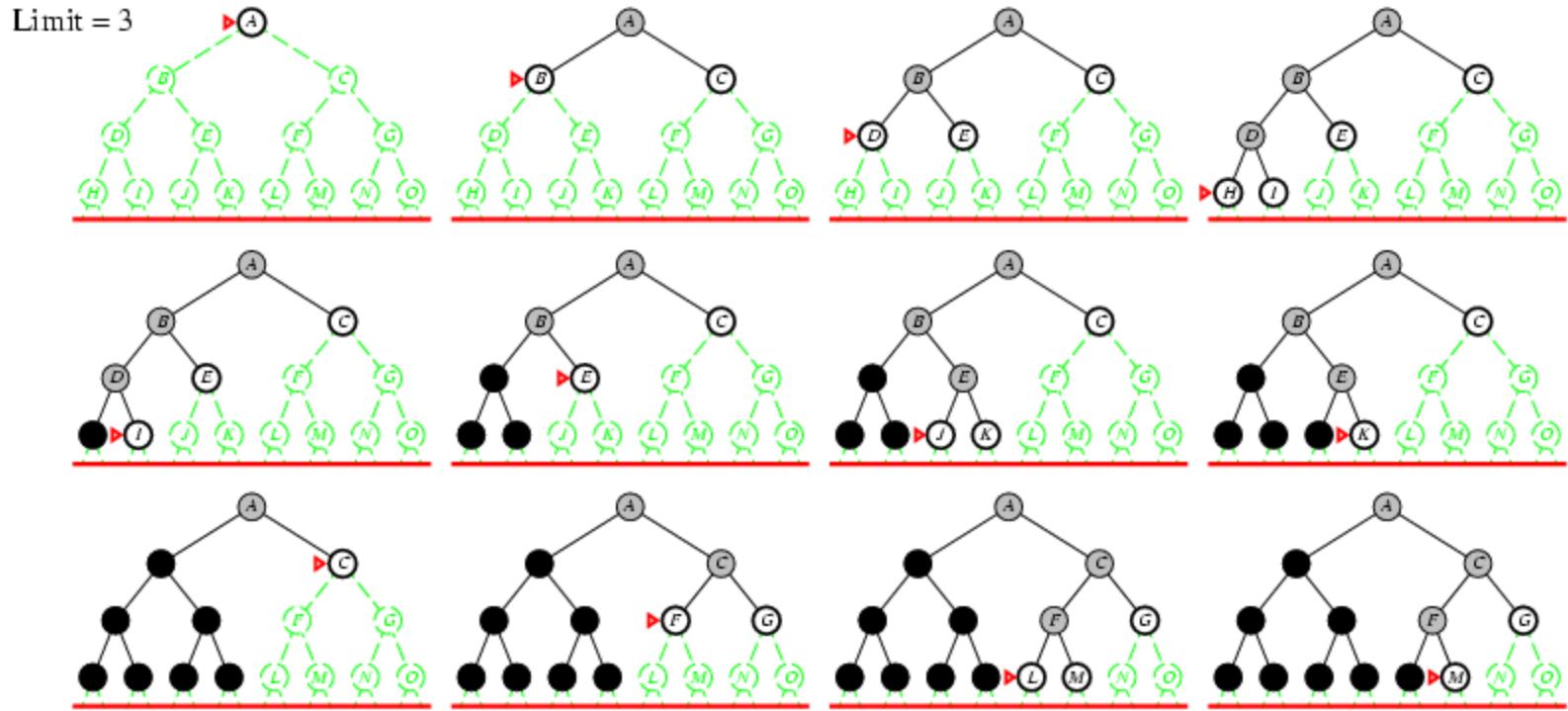


Iterative Deepening Search Example

Limit = 2



Iterative Deepening Search Example



Iterative Deepening Search Properties

Completeness?

- Yes if a minimal depth solution of depth d exists.
 - What happens when the depth limit $L=d$?
 - What happens when the depth limit $L < d$?

Time Complexity?

Iterative Deepening Search Properties

Time Complexity

- $(d+1)b^0 + db^1 + (d-1)b^2 + \dots + b^d = O(b^d)$
- E.g. $b=4, d=10$
 - $(11)*4^0 + 10*4^1 + 9*4^2 + \dots + 4^{10} = 1,864,131$
 - $4^{10} = 1,048,576$
 - Most nodes lie on bottom layer.

BFS can explore more states than IDS!

- For IDS, the time complexity is
 - $(d+1)b^0 + db^1 + (d-1)b^2 + \dots + b^d = O(b^d)$
- For BFS, the time complexity is
 - $1 + b + b^2 + b^3 + \dots + b^d + b(b^d - 1) = O(b^{d+1})$

E.g. $b=4$, $d=10$

- For IDS
 - $(11)*4^0 + 10*4^1 + 9*4^2 + \dots + 4^{10} = 1,864,131$ (states generated)
- For BFS
 - $1 + 4 + 4^2 + \dots + 4^{10} + 4(4^{10} - 1) = 5,592,401$ (states generated)
 - In fact IDS can be more efficient than breadth first search: nodes at limit are not expanded. BFS must expand all nodes until it expands a goal node. So at the bottom layer it will add many nodes to OPEN before finding the goal node.

Iterative Deepening Search Properties

Space Complexity

- $O(bd)$ Still linear!

Optimal?

- Will find shortest length solution which is optimal if costs are uniform.
- If costs are not uniform, we can use a “cost” bound instead.
 - Only expand paths of cost less than the cost bound.
 - Keep track of the minimum cost unexpanded path in each depth first iteration, increase the cost bound to this on the next iteration.
 - This can be more expensive. Need as many iterations of the search as there are distinct path costs.

Path/Cycle Checking

Path Checking

If n_k represents the path $\langle s_0, s_1, \dots, s_k \rangle$ and we expand s_k to obtain child c , we have

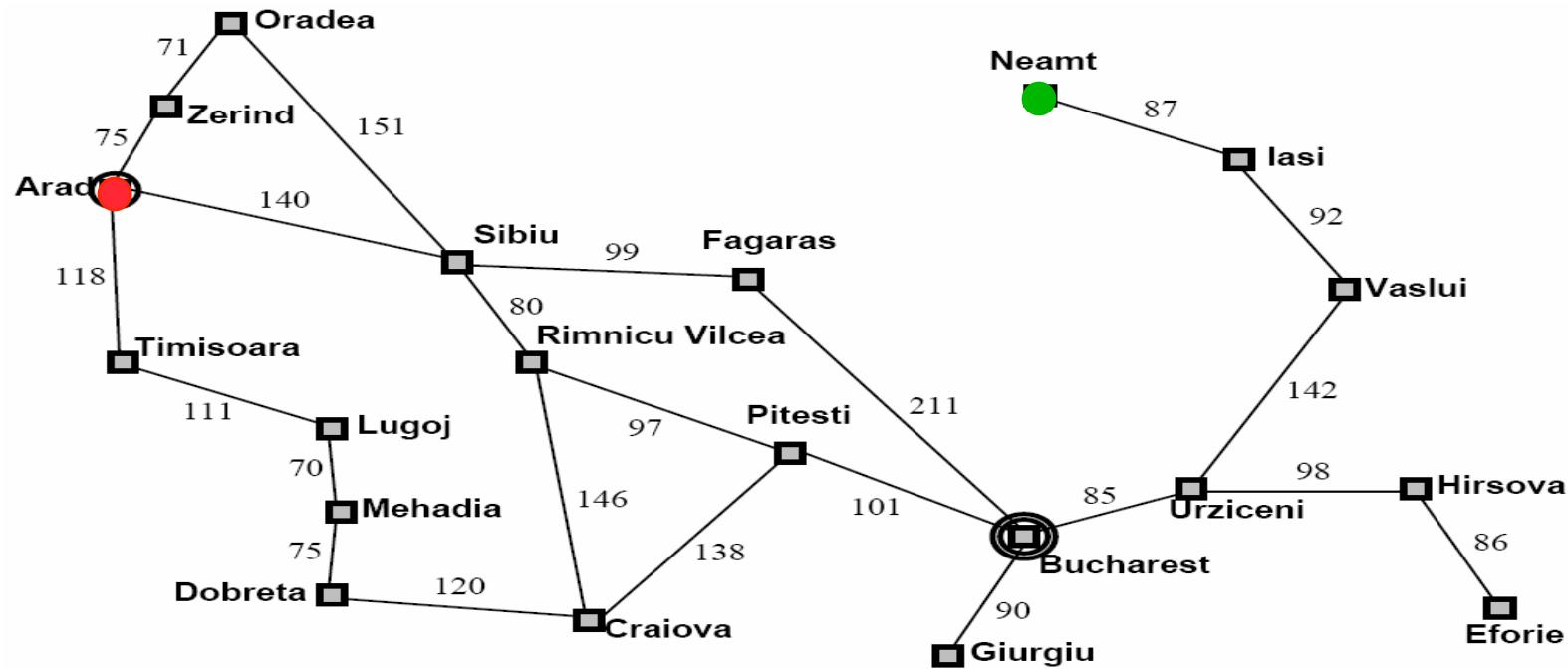
$$\langle S, s_1, \dots, s_k, c \rangle$$

As the path to “c”.

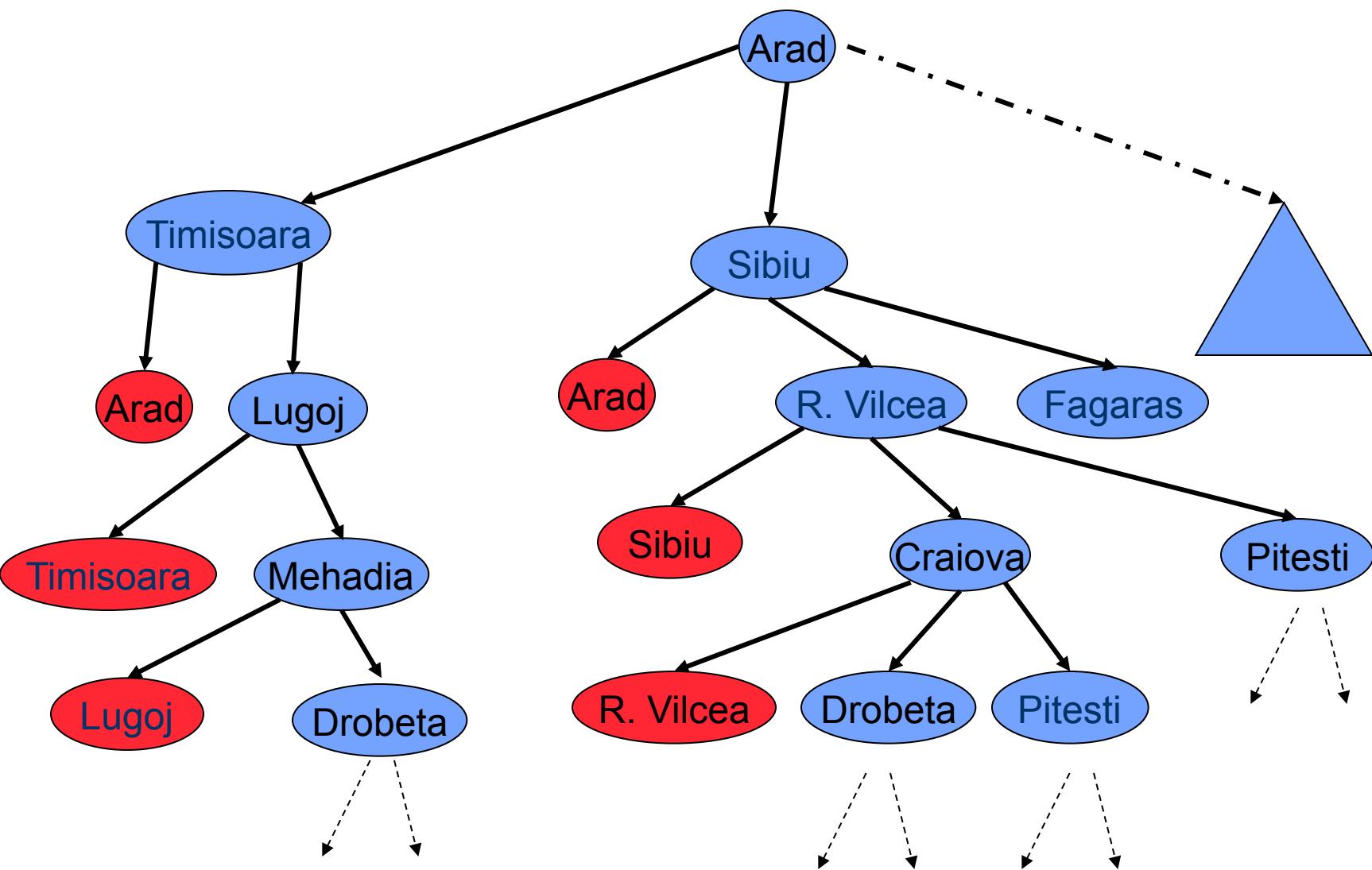
Path checking:

- Ensure that the state c is not equal to the state reached by any ancestor of c along this path.
- Paths are checked in isolation!

Example: Arad to Neamt



Path Checking Example

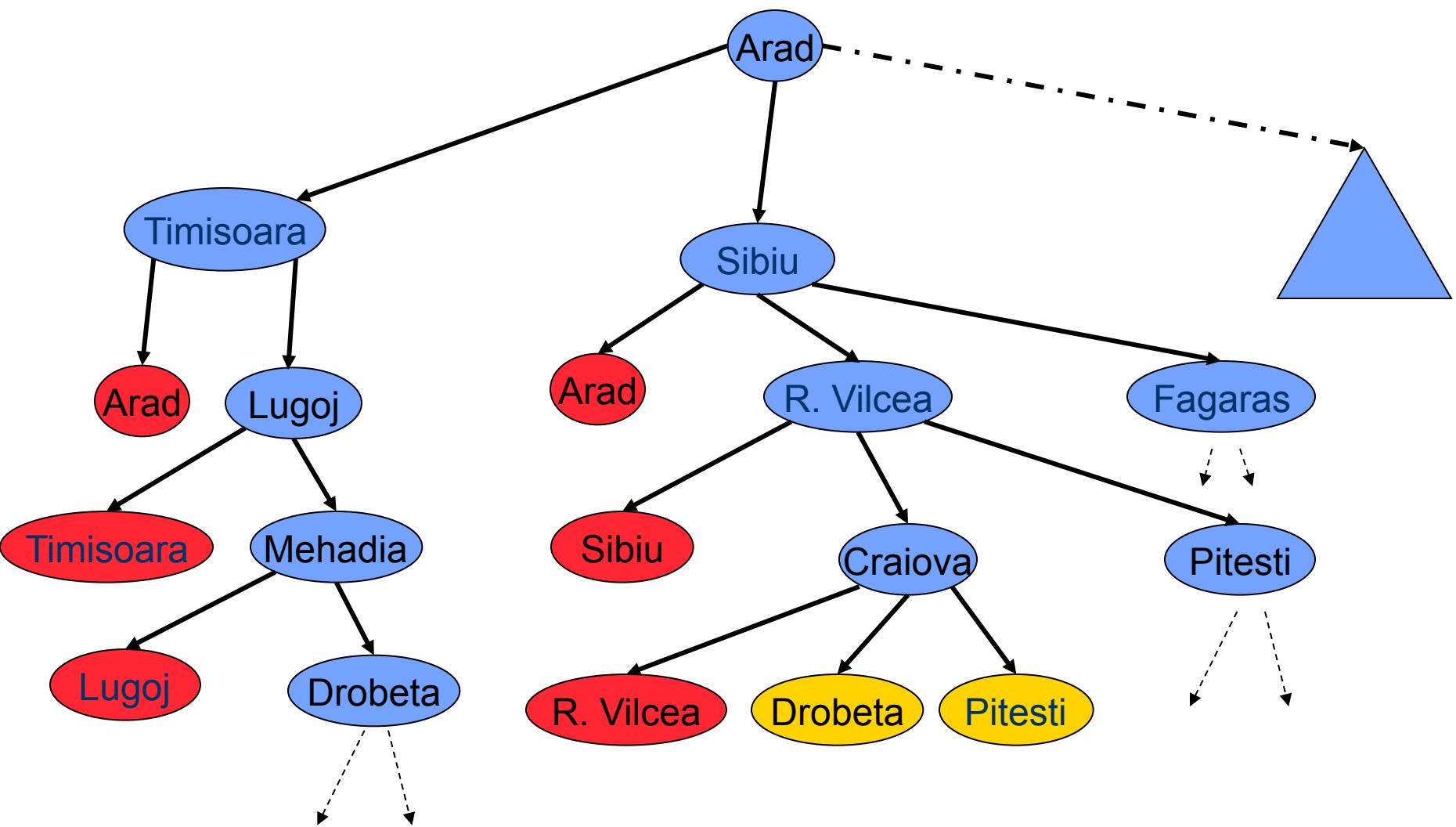


Cycle Checking

Cycle Checking

- Keep track of **all states** previously expanded during the search.
- When we expand n_k to obtain child c
 - Ensure that c is not equal to **any** previously expanded state.
- This is called **cycle checking**, or **multiple path checking**.
- What happens when we utilize this technique with depth-first search?
 - **What happens to space complexity?**

Cycle Checking Example (BFS)



Cycle Checking

- Higher space complexity (equal to the space complexity of breadth-first search).
- There is an additional issue when we are looking for an optimal solution
 - With uniform-cost search, we still find an optimal solution
 - The first time uniform-cost expands a state it has found the minimal cost path to it.
 - This means that the nodes rejected subsequently by cycle checking can't have better paths.
 - We will see later that we don't always have this property when we do heuristic search.

Heuristic Search (Informed Search)

Heuristic Search

- In **uninformed search**, we don't try to evaluate which of the nodes on OPEN are most promising. We never “look-ahead” to the goal.
E.g., in uniform cost search we always expand the cheapest path. We don't consider the cost of getting to the goal from the end of the current path.
- Often we have some other knowledge about the merit of nodes, e.g., going the wrong direction in Romania.

Heuristic Search

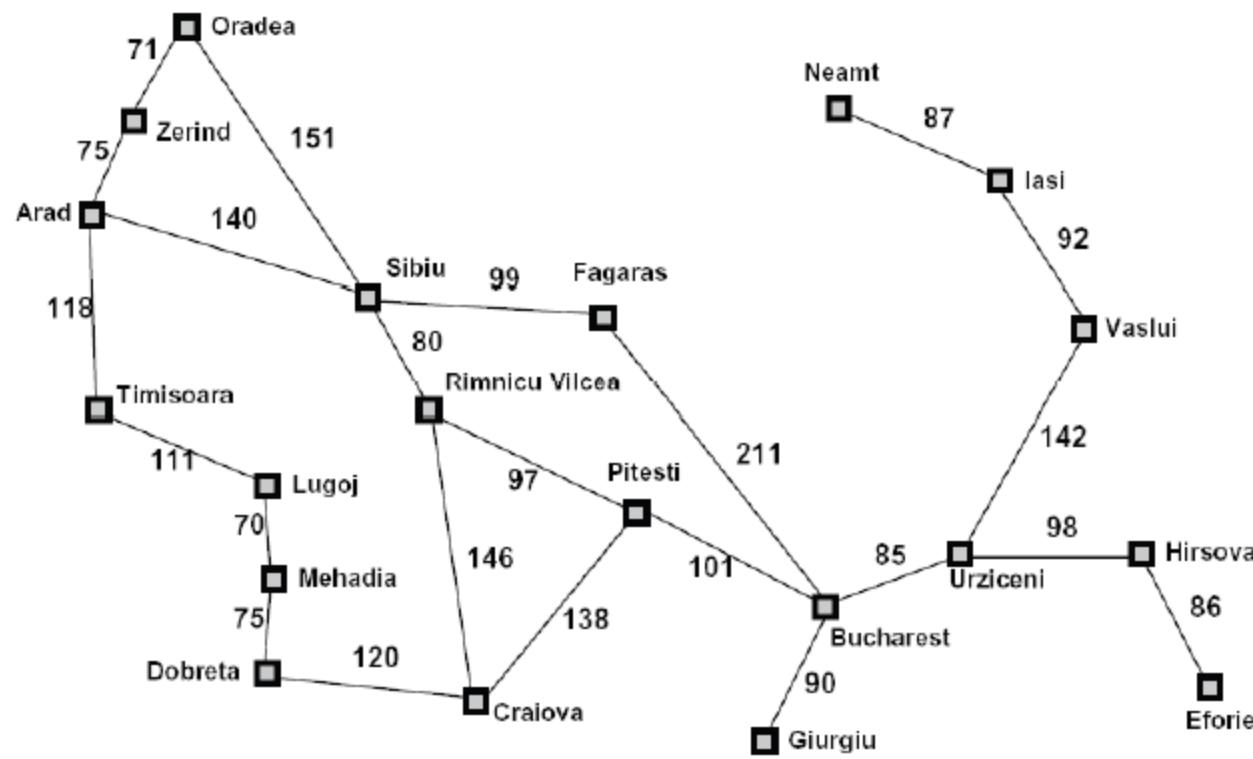
Merit of an OPEN node: different notions of merit.

- If we are concerned about the **cost of the solution**, we might want to consider how costly it is to get to the goal from the terminal state of that node.
- If we are concerned about **minimizing computation** in search we might want to consider how easy it is to find the goal from the terminal state of that node.
- We will focus on the “**cost of solution**” notion of merit.

Heuristic Search

- The idea is to develop a domain specific heuristic function $h(n)$.
- $h(n)$ **guesses** the cost of getting to the goal from node n (i.e., from the terminal state of the path represented by n).
- There are different ways of guessing this cost in different domains.
 - heuristics are **domain specific**.

Example: Euclidean distance



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

Planning a path from Arad to Bucharest, we can utilize the **straight line distance from each city to our goal**. This lets us plan our trip by picking cities at each time point that minimize the distance to our goal.

Heuristic Search

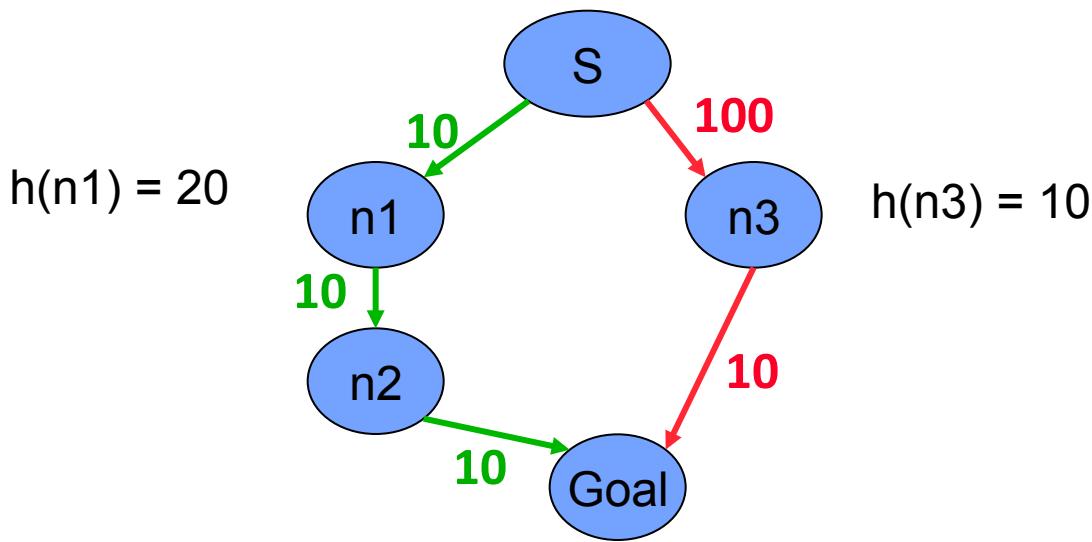
- If $h(n_1) < h(n_2)$ this means that we guess that it is cheaper to get to the goal from n_1 than from n_2 .
- We require that
 - $h(n) = 0$ for every node n whose terminal state satisfies the goal.
 - Zero cost of achieving the goal from node that already satisfies the goal.

Using only $h(n)$: Greedy best-first search

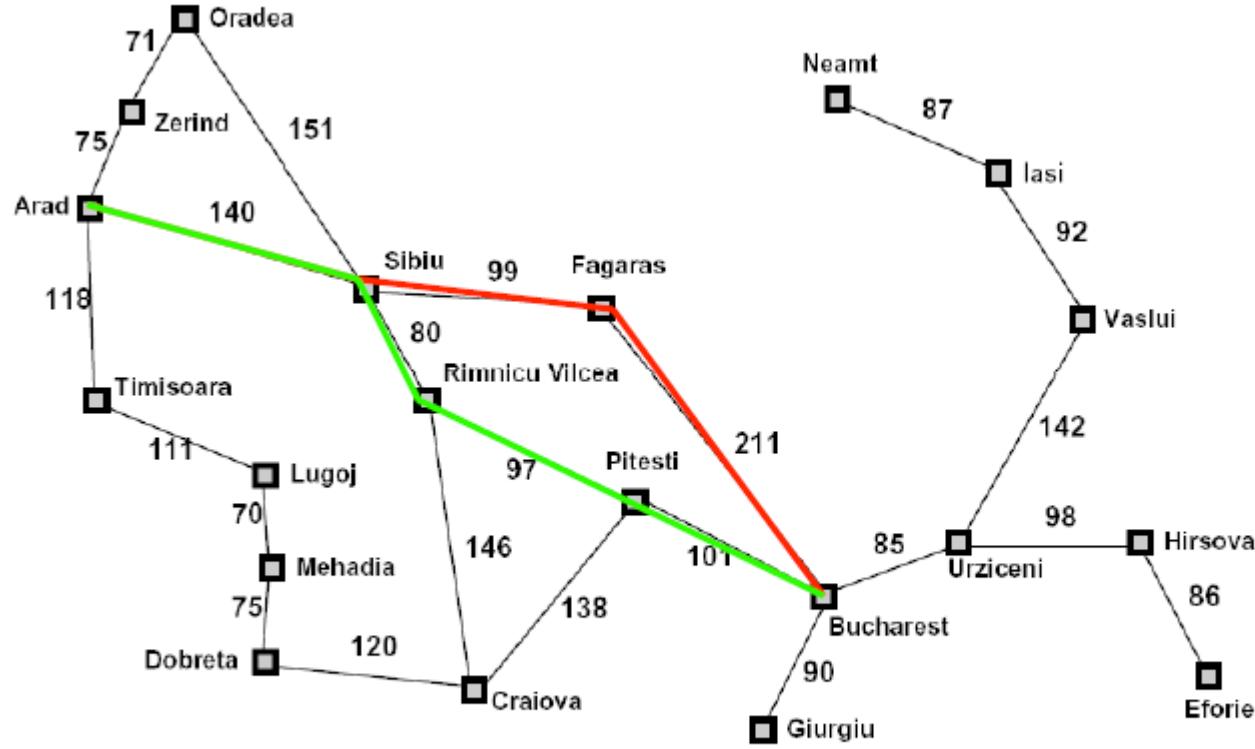
- We use $h(n)$ to rank the nodes on OPEN
 - Always expand node with lowest h -value.
- We are greedily trying to achieve a low cost solution.
- However, this method **ignores the cost of getting to n** , so it can be lead astray exploring nodes that cost a lot but seem to be close to the goal:

→ step cost = 10

→ step cost = 100



Greedy best-first search example

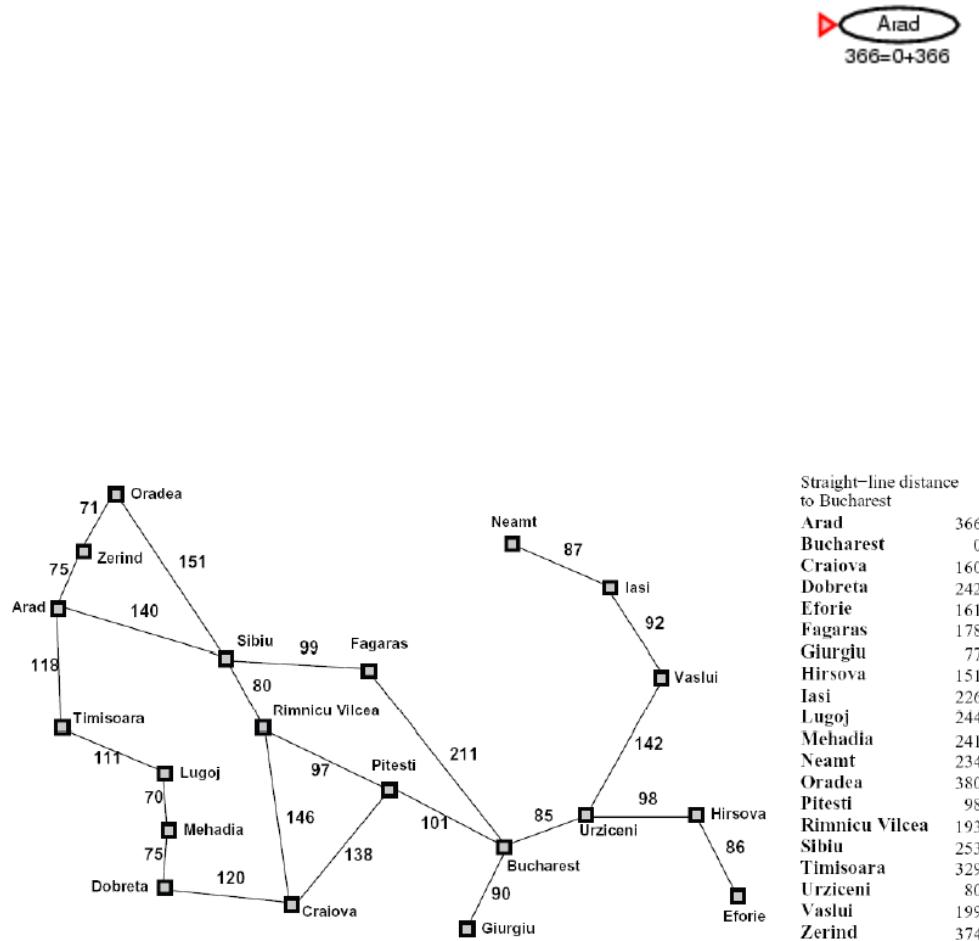


Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

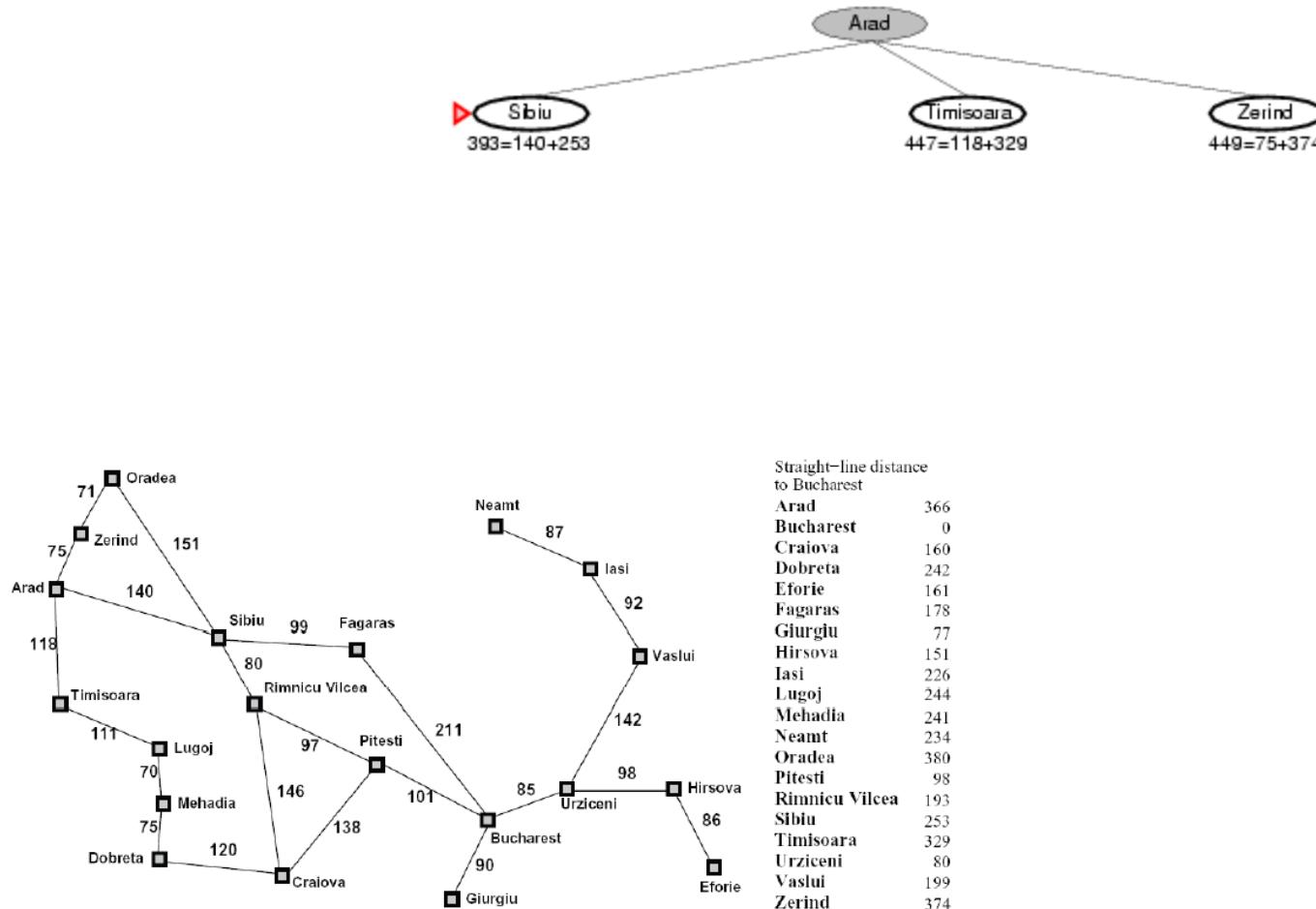
A* search

- Take into account the cost of getting to the node as well as our estimate of the cost of getting to the goal from the node.
- Define an evaluation function $f(n)$
$$f(n) = g(n) + h(n)$$
 - $g(n)$ is the cost of the path represented by node n
 - $h(n)$ is the heuristic estimate of the cost of achieving the goal from n .
- Always expand the node with lowest f-value on OPEN.
- The f-value, $f(n)$ is an estimate of the cost of getting to the goal via the node (path) n .
 - I.e., we first follow the path n then we try to get to the goal. $f(n)$ estimates the total cost of such a solution.

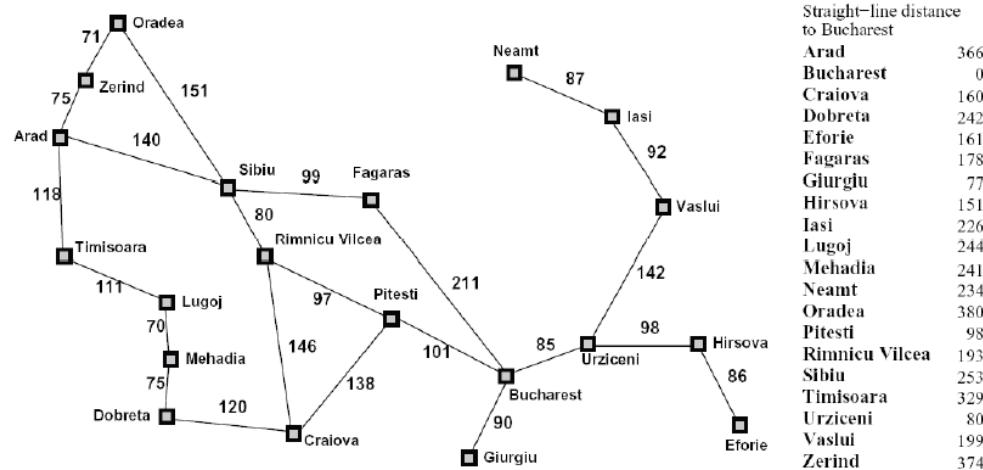
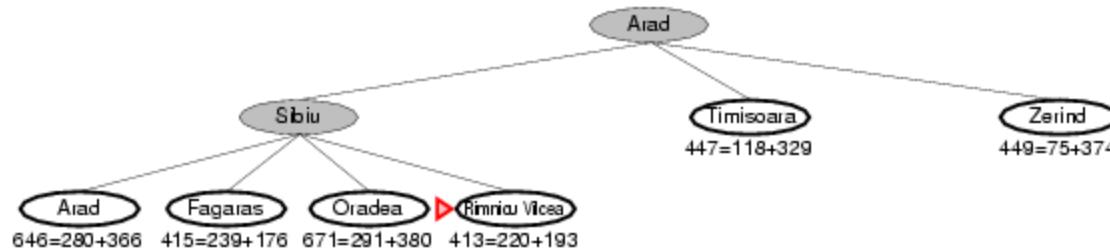
A* example



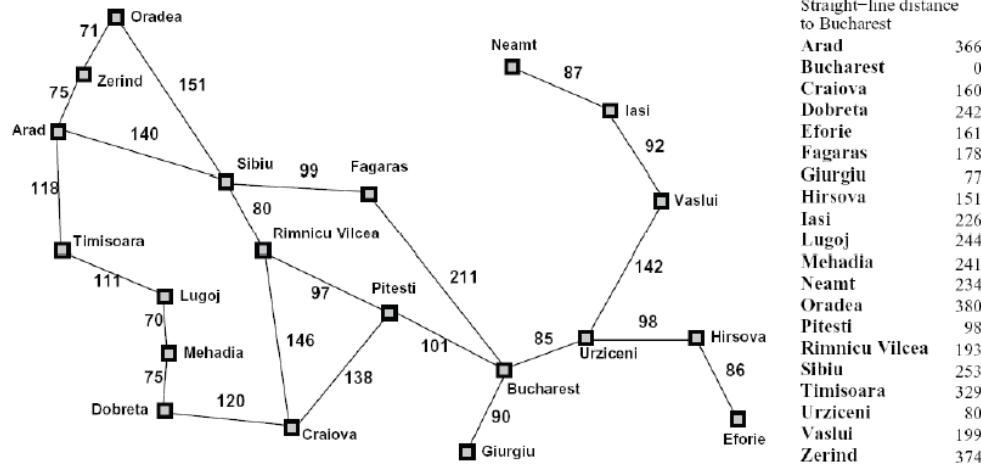
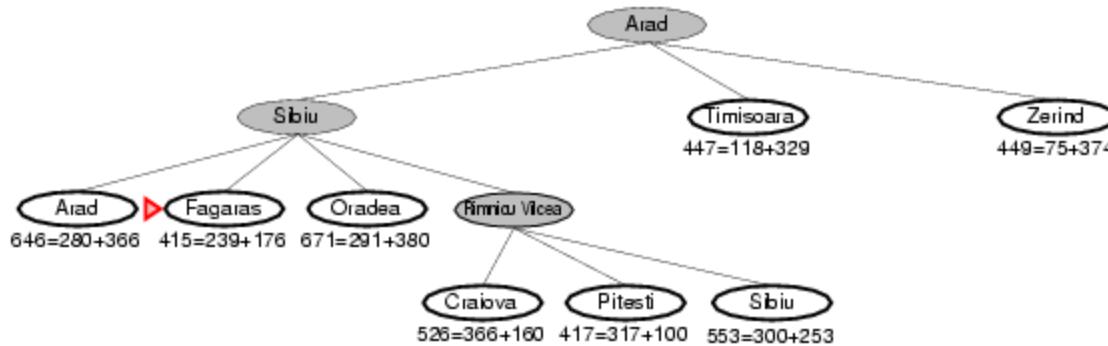
A* example



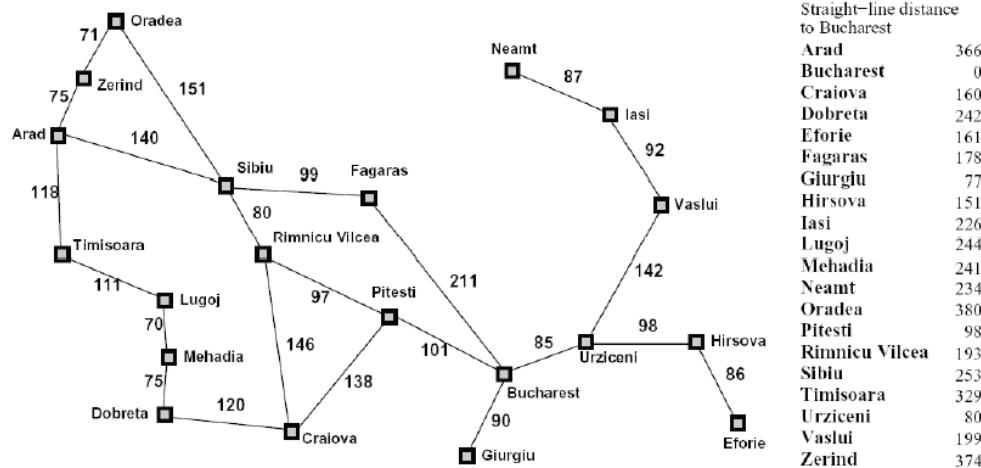
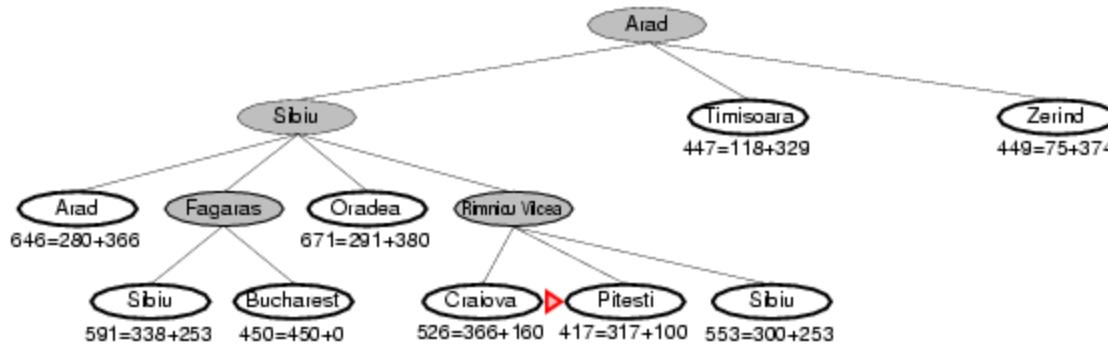
A* example



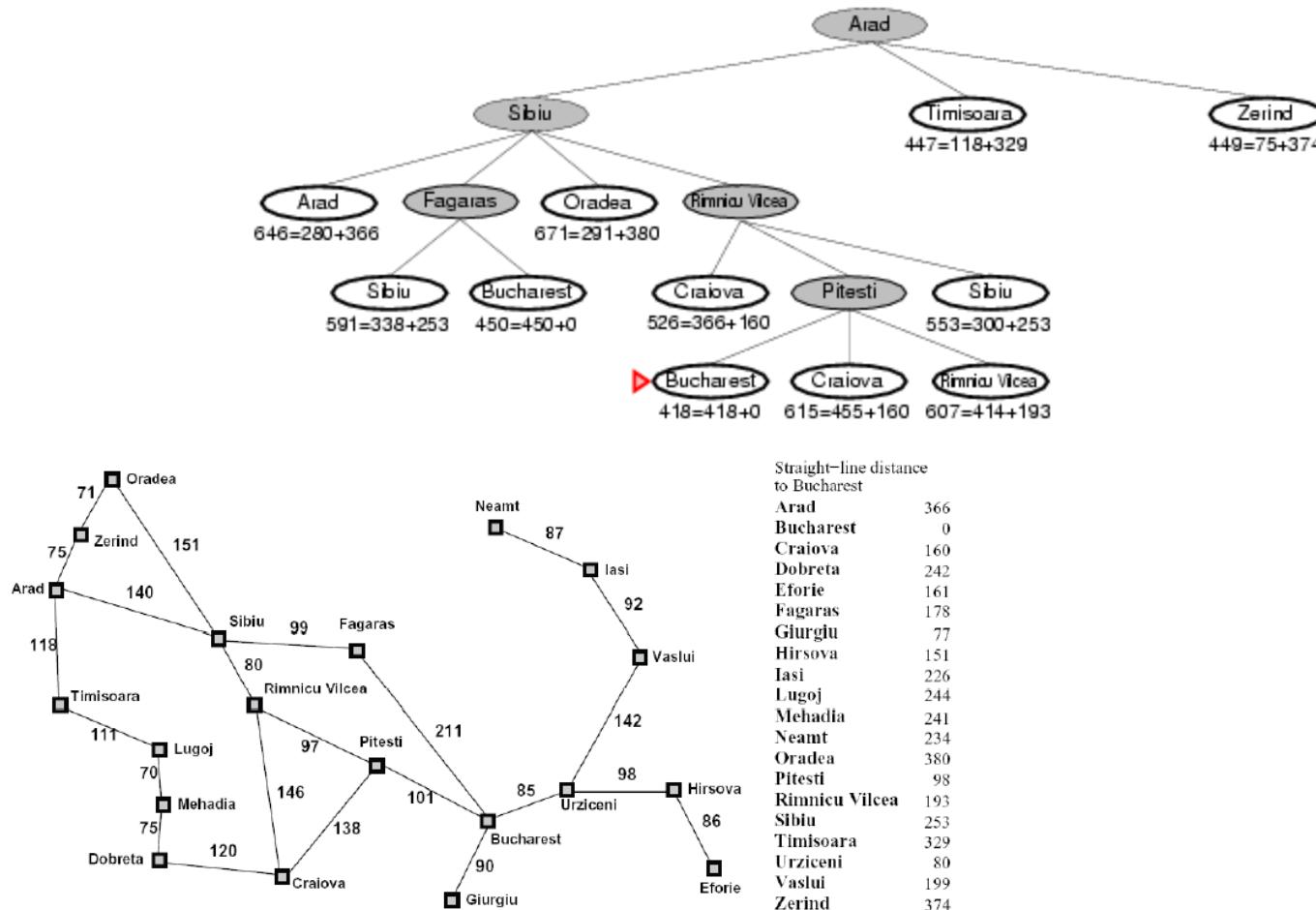
A* example



A* example



A* example



Properties of A* depend on conditions on $h(n)$

- We want to analyze the behavior of the resultant search.
 - Completeness, time and space, optimality?
- To obtain such results we must put some further conditions on the heuristic function $h(n)$ and the search space.

Conditions on $h(n)$: Admissible

- We always assume that $c(s_1, a, s_2) \geq \varepsilon > 0$ for any two states s_1 and s_2 and any action a : the cost of any transition is greater than zero and can't be arbitrarily small.

Let $h^*(n)$ be the **cost of an optimal path** from n to a goal node (∞ if there is no path). Then an **admissible** heuristic satisfies the condition

$$h(n) \leq h^*(n)$$

- an admissible heuristic **never over-estimates** the cost to reach the goal, i.e., it is **optimistic**
- Hence $h(g) = 0$, for any goal node g
- Also $h^*(n) = \infty$ if there is no path from n to a goal node

Consistency (aka monotonicity)

- A stronger condition than $h(n) \leq h^*(n)$.
- A **monotone/consistent** heuristic satisfies the triangle inequality: for all nodes n_1, n_2 and for all actions a

$$h(n_1) \leq C(n_1, a, n_2) + h(n_2)$$

Where $C(n_1, a, n_2)$ means the cost of getting from the terminal state of n_1 to the terminal state of n_2 via action a .

- Note that there might be more than one transition (action) between n_1 and n_2 , the inequality must hold for all of them.
- Monotonicity implies admissibility.
 - $(\text{forall } n_1, n_2, a) \ h(n_1) \leq C(n_1, a, n_2) + h(n_2) \rightarrow (\text{forall } n) \ h(n) \leq h^*(n)$

Consistency \rightarrow Admissible

- Assume consistency: $h(n) \leq c(n,a,n_2) + h(n_2)$
- Prove admissible: $h(n) \leq h^*(n)$

Proof:

Let $n \rightarrow n_1 \rightarrow \dots \rightarrow n^*$ be an OPTIMAL path from n to a goal (with actions a_1, a_2). Note the cost of this path is $h^*(n)$, and each subpath ($n_i \rightarrow \dots \rightarrow n^*$) has cost equal to $h^*(n_i)$.

If no path exists from n to a goal then $h^*(n) = \infty$ and $h(n) \leq h^*(n)$

Otherwise prove $h(n) \leq h^*(n)$ by induction on the length of this optimal path.

Base Case: $n = n^*$

By our conditions on h , $h(n) = 0 \leq h(n)^* = 0$

Induction Hypothesis: $h(n_1) \leq h^*(n_1)$

$$h(n) \leq c(n,a_1,n_1) + h(n_1) \leq c(n,a_1,n_1) + h^*(n_1) = h^*(n)$$

Intuition behind admissibility

$h(n) \leq h^*(n)$ means that the search won't miss any promising paths.

- If it really is cheap to get to a goal via n (i.e., both $g(n)$ and $h^*(n)$ are low), then $f(n) = g(n) + h(n)$ will also be low, and the search won't ignore n in favor of more expensive options.
- This can be formalized to show that admissibility implies optimality.
- Monotonicity gives some additional properties when it comes to cycle checking.

Consequences of monotonicity

1. The f-values of nodes along a path must be non-decreasing.

Let $\text{Start} \rightarrow s_1 \rightarrow s_2 \rightarrow \dots \rightarrow s_k$ be a path. Let n_i be the subpath $\text{Start} \rightarrow s_1 \rightarrow \dots \rightarrow s_i$:

We claim that: $f(n_i) \leq f(n_{i+1})$

Proof

$$\begin{aligned}f(n_i) &= c(\text{Start} \rightarrow \dots \rightarrow n_i) + h(n_i) \\&\leq c(\text{Start} \rightarrow \dots \rightarrow n_i) + c(n_i \rightarrow n_{i+1}) + h(n_{i+1}) \\&\leq c(\text{Start} \rightarrow \dots \rightarrow n_i \rightarrow n_{i+1}) + h(n_{i+1}) \\&\leq g(n_{i+1}) + h(n_{i+1}) = f(n_{i+1})\end{aligned}$$

Consequences of monotonicity

2. If n_2 is expanded immediately after n_1 , then
 $f(n_1) \leq f(n_2)$

(the f-value of expanded nodes is **monotonic** non-decreasing)

Proof:

- If n_2 was on OPEN when n_1 was expanded, then $f(n_1) \leq f(n_2)$ otherwise we would have expanded n_2 .
- If n_2 was added to OPEN after n_1 's expansion, then n_2 extends n_1 's path. That is, the path represented by n_1 is a prefix of the path represented by n_2 . By property (1) we have $f(n_1) \leq f(n_2)$ as the f-values along a path are non-decreasing.

Consequences of monotonicity

3. Corollary: the sequence of f-values of the nodes expanded by A* is non-decreasing. I.e, If n_2 is expanded **after** (not necessarily immediately after) n_1 , then $f(n_1) \leq f(n_2)$
(the f-value of expanded nodes is **monotonic** non-decreasing)

Proof:

- If n_2 was on OPEN when n_1 was expanded, then $f(n_1) \leq f(n_2)$ otherwise we would have expanded n_2 .
- If n_2 was added to OPEN after n_1 's expansion, then let n be an ancestor of n_2 that was present when n_1 was being expanded (this could be n_1 itself). We have $f(n_1) \leq f(n)$ since A* chose n_1 while n was present on OPEN. Also, since n is along the path to n_2 , by property (1) we have $f(n) \leq f(n_2)$. So, we have $f(n_1) \leq f(n_2)$.

Consequences of monotonicity

4. When n is expanded every path with lower f -value has already been expanded.

- **Proof:** Assume by contradiction that there exists a path $\langle \text{Start}, n_0, n_1, n_{i-1}, n_i, n_{i+1}, \dots, n_k \rangle$ with $f(n_k) < f(n)$ and n_i is its last expanded node.
 - n_{i+1} must be on OPEN while n is expanded, so
 - a) by (1) $f(n_{i+1}) \leq f(n_k)$ since they lie along the same path.
 - b) since $f(n_k) < f(n)$ so we have $f(n_{i+1}) < f(n)$
 - c) by (2) $f(n) \leq f(n_{i+1})$ because n is expanded before n_{i+1} .
 - Contradiction from b&c!

Consequences of monotonicity

5. With a monotone heuristic, the first time A* expands a state, it has found the minimum cost path to that state.

Proof:

- Let **PATH1** = <Start, $s_0, s_1, \dots, s_k, s>$ be **the first** path to a state s found. We have $f(\text{path1}) = c(\text{PATH1}) + h(s)$.
- Let **PATH2** = <Start, $t_0, t_1, \dots, t_j, s>$ be another path to s found later. we have $f(\text{path2}) = c(\text{PATH2}) + h(s)$.
- Note $h(s)$ is dependent only on the state s (terminal state of the path) it does not depend on how we got to s .
- By property (3), $f(\text{path1}) \leq f(\text{path2})$
- hence: $c(\text{PATH1}) \leq c(\text{PATH2})$

Consequences of monotonicity

Complete.

- Yes, consider a least cost path to a goal node
 - $\text{SolutionPath} = \langle \text{Start} \rightarrow n_1 \rightarrow \dots \rightarrow G \rangle$ with cost $c(\text{SolutionPath})$. Since $h(G) = 0$, this means that $f(\text{SolutionPath}) = \text{cost}(\text{SolutionPath})$
 - Since each action has a cost $\geq \epsilon > 0$, there are only a finite number of paths that have f -value $< c(\text{SolutionPath})$. None of these paths lead to a goal node since SolutionPath is a least cost path to the goal.
 - So eventually SolutionPath , or some equal cost path to a goal must be expanded.

Time and Space complexity.

- When $h(n) = 0$, for all n h is monotone.
 - A* becomes uniform-cost search!
- It can be shown that when $h(n) > 0$ for some n and still admissible, the number of nodes expanded can be no larger than uniform-cost.
- Hence the same bounds as uniform-cost apply. (These are worst case bounds). Still exponential unless we have a very good h !
- In real world problems, we sometimes run out of time and memory. IDA* can sometimes be used to address memory issues, but IDA* isn't very good when many cycles are present.

Consequences of monotonicity

Optimality

- Yes, by (5) the first path to a goal node must be optimal.
 5. With a monotone heuristic, the first time A* expands a state, it has found the minimum cost path to that state.

Cycle Checking

- We can use a simple implementation of cycle checking (multiple path checking)---just reject all search nodes visiting a state already visited by a previously expanded node. By property (5) we need keep only the first path to a state, rejecting all subsequent paths.

Admissibility without monotonicity

When “ h ” is admissible but not monotonic.

- Time and Space complexity remain the same. Completeness holds.
- Optimality still holds (without cycle checking), but need a different argument: don't know that paths are explored in order of cost.

Admissibility without monotonicity

What about Cycle Checking?

- No longer guaranteed we have found an optimal path to a node *the first time* we visit it.
- So, cycle checking might not preserve optimality.
 - To fix this: for previously visited nodes, must remember cost of previous path. If new path is cheaper must explore again.

Space Problems with A*

- A* has the same potential space problems as BFS or UCS
- IDA* - Iterative Deepening A* is similar to Iterative Deepening Search and similarly addresses space issues.

IDA* - Iterative Deepening A*

Objective: reduce memory requirements for A*

- Like iterative deepening, but now the “cutoff” is the f-value ($g+h$) rather than the depth
- At each iteration, the cutoff value is the smallest f-value of any node that exceeded the cutoff on the previous iteration
- Avoids overhead associated with keeping a sorted queue of nodes, and the open list occupies only linear space.
- Two new parameters:
 - curBound (any node with a bigger f-value is discarded)
 - smallestNotExplored (the smallest f-value for discarded nodes in a round) when OPEN becomes empty, the search starts a new round with this bound.
 - Easier to expand all nodes with f-value EQUAL to the f-limit. This way we can compute “smallestNotExplored” more easily.

Constructing Heuristics

Building Heuristics: Relaxed Problem

- One useful technique is to consider an easier problem, and let $h(n)$ be the cost of reaching the goal in the easier problem.
- 8-Puzzle moves.
 - Can move a tile from square A to B if
 - A is adjacent (left, right, above, below) to B
 - and B is blank
- Can relax some of these conditions
 1. can move from A to B if A is adjacent to B (ignore whether or not position is blank)
 2. can move from A to B if B is blank (ignore adjacency)
 3. can move from A to B (ignore both conditions).

Building Heuristics: Relaxed Problem

- #3 “*can move from A to B (ignore both conditions)* ”.
leads to the misplaced tiles heuristic.
 - To solve the puzzle, we need to move each tile into its final position.
 - Number of moves = number of misplaced tiles.
 - Clearly $h(n) = \text{number of misplaced tiles} \leq h^*(n)$ the cost of an optimal sequence of moves from n.
- #1 “*can move from A to B if A is adjacent to B (ignore whether or not position is blank)* ”
leads to the manhattan distance heuristic.
 - To solve the puzzle we need to slide each tile into its final position.
 - We can move vertically or horizontally.
 - Number of moves = sum over all of the tiles of the number of vertical and horizontal slides we need to move that tile into place.
 - Again $h(n) = \text{sum of the manhattan distances} \leq h^*(n)$
 - in a real solution we need to move each tile at least that far and we can only move one tile at a time.

Building Heuristics: Relaxed Problem

Comparison of IDS and A* (average total nodes expanded):

Depth	IDS	A*(Misplaced) h1	A*(Manhattan) h2
10	47,127	93	39
14	3,473,941	539	113
24	---	39,135	1,641

Let **h1**=Misplaced, **h2**=Manhattan

- Does h2 **always** expand fewer nodes than h1?
 - Yes! Note that **h2 dominates h1**, i.e. for all n: $h1(n) \leq h2(n)$. From this you can prove h2 is faster than h1 (once both are admissible).
 - Therefore, among several admissible heuristic the one with highest value is the fastest.

Building Heuristics: Relaxed Problem

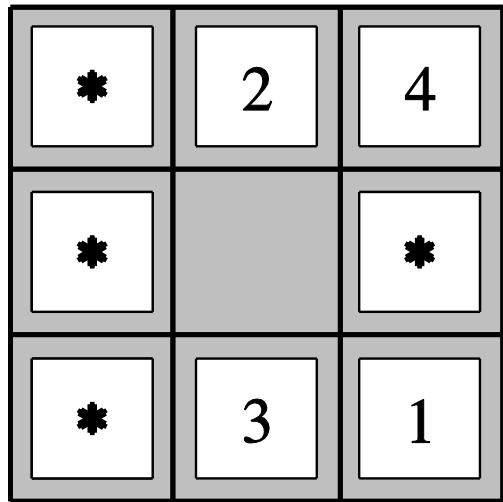
The **optimal** cost to nodes in the relaxed problem is an **admissible heuristic** for the original problem!

Proof Idea: the optimal solution in the original problem is a solution for relaxed problem, therefore it must be at least as expensive as the optimal solution in the relaxed problem.

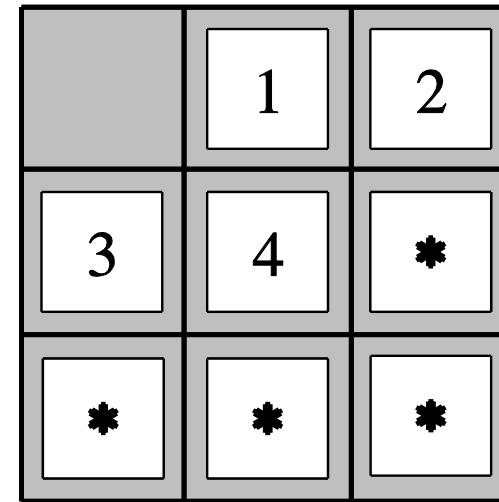
So admissible heuristics can sometimes be constructed by finding a relaxation whose optimal solution can be easily computed.

Building Heuristics: Pattern databases

- Try to generate admissible heuristics by solving a subproblem and storing the exact solution cost for that subproblem
- See Chapter 3.6.3 if you are interested.



Start State



Goal State

Search

- One of the most basic techniques in AI
 - Underlying sub-module in most AI systems
- Can solve many problems that humans are not good at (achieving super-human performance)
- Very useful as a general algorithmic technique for solving many non-AI problems.

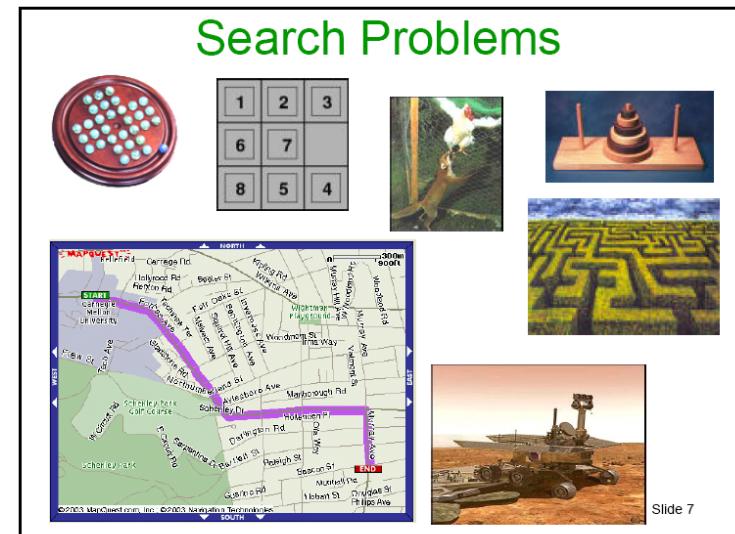
How do we plan our holiday?

- We must take into account various preferences and constraints to develop a schedule.
- An important technique in developing such a schedule is “**hypothetical**” reasoning.
- Example: On holiday in England
 - Currently in Edinburgh
 - Flight leaves tomorrow from London
 - Need plan to get to your plane
 - If I take a 6 am train where will I be at 2 pm? Will I be still able to get to the airport on time?

How do we plan our holiday?

- This kind of hypothetical reasoning involves asking
 - what state will I be in after taking certain actions, or after certain sequences of events?
- From this we can reason about particular sequences of events or actions one should try to bring about to achieve a desirable state.
- Search is a computational method for capturing a particular version of this kind of reasoning.

Many problems can be solved by search:



Why Search?

- Successful
 - Success in game playing programs based on search.
 - Many other AI problems can be successfully solved by search.
- Practical
 - Many problems don't have specific algorithms for solving them. Casting as search problems is often the easiest way of solving them.
 - Search can also be useful in approximation (e.g., local search in optimization problems).
 - Problem specific heuristics provides search with a way of exploiting extra knowledge.
- Some critical aspects of intelligent behaviour, e.g., planning, can be naturally cast as search.

Limitations of Search

- There are many difficult questions that are not resolved by search. In particular, the whole question of how does an intelligent system formulate the problem it wants to solve as a search problem is not addressed by search.
- Search only shows how to solve the problem once we have it correctly formulated.

Search

Search

- Formulating a problem as search problem (representation)
- Heuristic Search
- Game-Tree-Search
- Readings
 - Introduction: Chapter 3.1 – 3.3
 - Uninformed Search: Chapter 3.4
 - Heuristic Search: Chapters 3.5, 3.6

Representing a problem: The Formalism

To formulate a problem as a search problem we need the following components:

1. **STATE SPACE:** Formulate a **state space** over which we perform search. The state space is a way of representing in a computer the states of the real problem.
2. **ACTIONS or STATE SPACE Transitions:** Formulate **actions** that allow one to move between different states. The actions reflect the actions one can take in the real problem but operate on the state space instead.

Representing a problem: The Formalism

To formulate a problem as a search problem we need the following components:

3. **INITIAL or START STATE and GOAL:** Identify the **initial state** that best represents the starting conditions, and the goal or condition one wants to achieve.
4. **Heuristics:** Formulate various **heuristics** to help guide the search process.

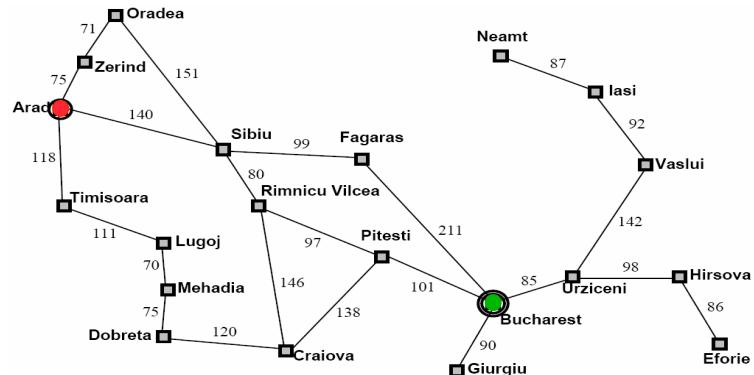
The Formalism

Once the problem has been formulated as a state space search, various algorithms can be utilized to solve the problem.

- A solution to the problem will be a sequence of actions/moves that can transform your current state into a state where your desired condition holds.

Example 1: Romania Travel.

Currently in **Arad**, need to get to **Bucharest** by tomorrow to catch a flight. What is the **State Space**?



Example 1.

- State space.
- **States:** the various cities you could be located in.
 - Our abstraction: we are ignoring the low level details of driving, states where you are on the road between cities, etc.
- **Actions:** drive between neighboring cities.
- **Initial state:** in Arad
- **Desired condition (Goal):** be in a state where you are in Bucharest. (How many states satisfy this condition?)
- Solution will be the route, the sequence of cities to travel through to get to Bucharest.

Example 2.

- Water Jugs

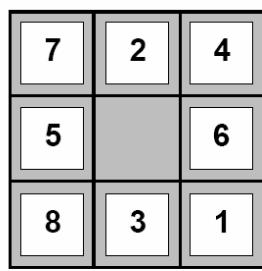
- We have a 3 gallon (liter) jug and a 4 gallon jug. We can fill either jug to the top from a tap, we can empty either jug, or we can pour one jug into the other (at least until the other jug is full).
- **States:** pairs of numbers (gal3, gal4) where gal3 is the number of gallons in the 3 gallon jug, and gal4 is the number of gallons in the 4 gallon jug.
- **Actions:** Empty-3-Gallon, Empty-4-Gallon, Fill-3-Gallon, Fill-4-Gallon, Pour-3-into-4, Pour 4-into-3.
- **Initial state:** Various, e.g., (0,0)
- **Desired condition (Goal):** Various, e.g., (0,2) or (*, 3) where * means we don't care.

Example 2.

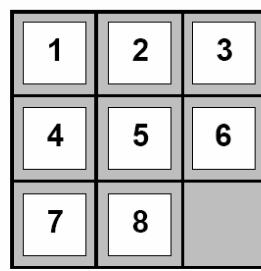
- Water Jugs

- If we start off with gal3 and gal4 as integer, can only reach integer values.
- Some values, e.g., (1,2) are not reachable from some initial state, e.g., (0,0).
- Some actions are no-ops. They do not change the state, e.g.,
 - (0,0) → Empty-3-Gallon → (0,0)

Example 3. The 8-Puzzle



Start State



Goal State

Rule: Can slide a tile into the blank spot.

Alternative view: move the blank spot around.

Example 3. The 8-Puzzle

- State space.

- **States:** The different configurations of the tiles. How many different states?
- **Actions:** Moving the blank up, down, left, right. Can every action be performed in every state?
- **Initial state:** e.g., state shown on previous slide.
- **Desired condition (Goal):** be in a state where the tiles are all in the positions shown on the previous slide.

- Solution will be a sequence of moves of the blank that transform the initial state to a goal state.

Example 3. The 8-Puzzle

- Although there are $9!$ different configurations of the tiles (362,880) in fact the state space is divided into two disjoint parts.
- Only when the blank is in the middle are all four actions possible.
- Our goal condition is satisfied by only a single state. But one could easily have a goal condition like
 - The 8 is in the upper left hand corner.
 - How many different states satisfy this goal?

More complex situations

- Perhaps actions lead to multiple states, e.g., flip a coin to obtain heads OR tails. Or we don't know for sure what the initial state is (prize is behind door 1, 2, or 3). Now we might want to consider how **likely** different states and action outcomes are.
- This leads to probabilistic models of the search space and different algorithms for solving the problem.
- Later we will see some techniques for reasoning under uncertainty.

Algorithms for Search

Inputs:

- a specified **initial state** (a specific world state)
- a **successor function** $S(x) = \{\text{set of states that can be reached from state } x \text{ via a single action}\}$.
- a **goal test** a function that can be applied to a state and returns true if the state satisfies the goal condition.
- An **action cost** function $C(x,a,y)$ which determines the cost of moving from state x to state y using action a . ($C(x,a,y) = \infty$ if a does not yield y from x). Note that different actions might generate the same move of $x \rightarrow y$.

Algorithms for Search

Output:

- a sequence of states leading from the initial state to a state satisfying the goal test.
- The sequence might be, optimal in cost for some algorithms, optimal in length for some algorithms, come with no optimality guarantees from other algorithms.

Algorithms for Search

Obtaining the action sequence.

- The set of successors of a state x might arise from different actions, e.g.,
 - $x \rightarrow a \rightarrow y$
 - $x \rightarrow b \rightarrow z$
- Successor function $S(x)$ yields a set of states that can be reached from x via **any** single action.
 - Rather than just return a set of states, we annotate these states by the action used to obtain them:
 - $S(x) = \{<y, a>, <z, b>\}$
 y via action a , z via action b .
 - $S(x) = \{<y, a>, <y, b>\}$
 y via action a , also y via alternative action b .

Template Search Algorithms

- The search space consists of **states** and actions that move between states.
- A **path** in the search space is a **sequence** of states connected by actions, $\langle s_0, s_1, s_2, \dots, s_k \rangle$, for every s_i and its successor s_{i+1} there must exist an action a_i that transitions s_i to s_{i+1} .
 - Alternately a path can be specified by
 - (a) an initial state s_0 , and
 - (b) a sequence of actions that are applied in turn starting from s_0 .
- The search algorithms perform search by examining alternate paths of the search space. The objects used in the algorithm are called **nodes**—each node contains a path.

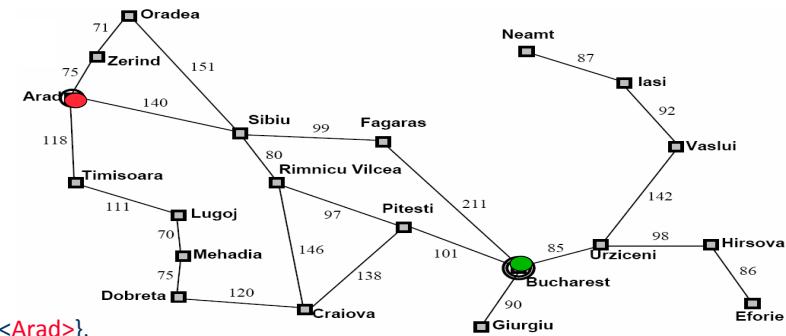
Template Algorithm for Search

- We maintain a set of **Frontier** nodes also called the **OPEN** set.
 - These nodes are paths in the search space that all start at the initial state.
- Initially we set $OPEN = \{\langle \text{Start State} \rangle\}$
 - The path (node) that starts and terminates at the start state.
- At each step we select a node n from $OPEN$. Let x be the state n terminates at. We check if x satisfies the goal, if not we add all extensions of n to $OPEN$ (by finding all states in $S(x)$).

Template Algorithm for Search

```
Search(OPEN, Successors, Goal? )  
  While(Open not EMPTY) {  
    n = select node from OPEN  
    Curr = terminal state of n  
    If (Goal?(Curr)) return n.  
    OPEN = (OPEN - {n}) Us ∈ Successors(Curr) <n,s>  
    /* Note OPEN could grow or shrink */  
  }  
  return FAIL
```

When does OPEN get smaller in size?



{<Arad>},

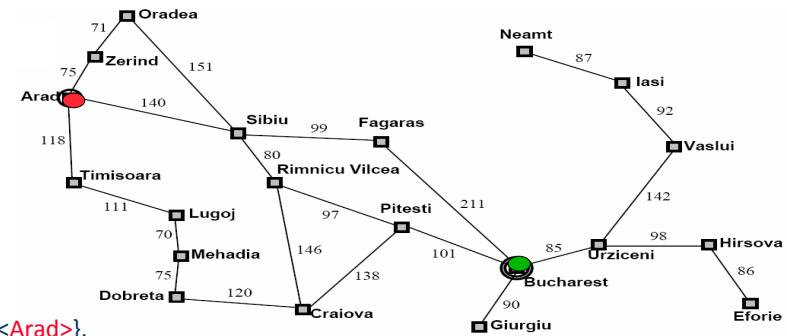
{<A,Z>, <A,T>, <A, S>},

{<A,Z>, <A,T>, <A,S,A>, <A,S,O>, <A,S,F>, <A,S,R>}

{<A,Z>, <A,T>, <A,S,A>, <A,S,O>, <A,S,R>, <A,S,F,S>, <A,S,F,B>}

Solution: Arad -> Sibiu -> Fagaras -> Bucharest

Cost: 140 + 99 + 211 = 450



{<Arad>},

{<A,Z>, <A,T>, <A, S>},

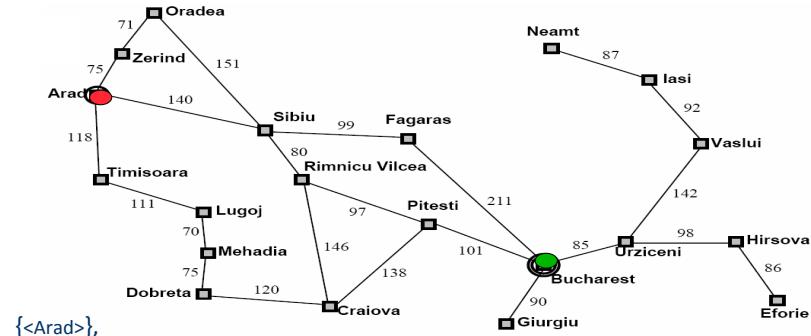
{<A,Z>, <A,T>, <A,S,A>, <A,S,O>, <A,S,F>, <A,S,R>}

{<A,Z>, <A,T>, <A,S,A>, <A,S,O>, <A,S,F>, <A,S,R,S>, <A,S,R,C>, <A,S,R,P>}

{<A,Z>, <A,T>, <A,S,A>, <A,S,O>, <A,S,F>, <A,S,R,S>, <A,S,R,C>, <A,S,R,P,R>, <A,S,R,P,C>, <A,S,R,P,B>}

Solution: Arad -> Sibiu -> Rimnicu Vilcea -> Pitesti -> Bucharest

Cost: 140 + 80 + 97 + 101 = 418



{<Arad>},

{<A,Z>, <A,T>, <A, S>},

{<A,Z>, <A,T>, <A,S,A>, <A,S,O>, <A,S,F>, <A,S,R>}

{<A,Z>, <A,T>, <A,S,A>, <A,S,O>, <A,S,R>, <A,S,F,S>, <A,S,F,B>}

....

cycles can cause non-termination!

... we deal with this issue later

Selection Rule

The order paths are selected from OPEN has a critical effect on the operation of the search:

- Whether or not a solution is found
- The cost of the solution found.
- The time and space required by the search.

How to select the next path from OPEN?

All search techniques keep OPEN as an ordered set (e.g., a priority queue) and repeatedly execute:

- If OPEN is empty, terminate with failure.
 - Get the **next** path from OPEN.
 - If the path leads to a goal state, terminate with success.
 - Extend the path (i.e. generate the successor states of the terminal state of the path) and put the new paths in OPEN.
- How do we order the paths on OPEN?

Critical Properties of Search

- **Completeness**: will the search always find a solution if a solution exists?
- **Optimality**: will the search always find the least cost solution? (when actions have costs)
- **Time complexity**: what is the maximum number of nodes (paths) than can be expanded or generated?
- **Space complexity**: what is the maximum number of nodes (paths) that have to be stored in memory?

Uninformed Search Strategies

- These are strategies that adopt a fixed rule for selecting the next state to be expanded.
- The rule does not change irrespective of the search problem being solved.
- These strategies do not take into account any domain specific information about the particular search problem.
- Uninformed search techniques:
 - Breadth-First, Uniform-Cost, Depth-First, Depth-Limited, and Iterative-Deepening search

Breadth-First Search

Breadth-First Search

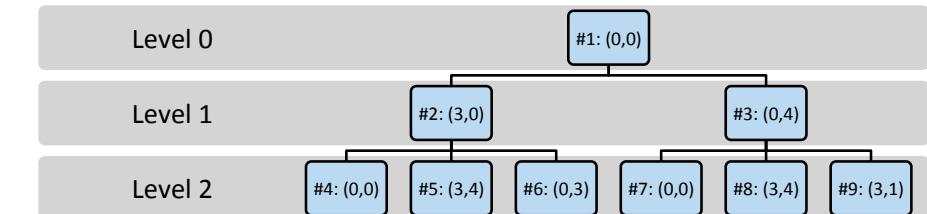
- Place the new paths that extend the current path at the **end** of OPEN.

WaterJugs. Start = (0,0), Goal = (*,2)

Green = Newly Added.

- OPEN = {<(0,0)>}
- OPEN = {<(0,0),(3,0)>, <(0,0),(0,4)>}
- OPEN = {<(0,0),(0,4)>, <(0,0),(3,0),(0,0)>, <(0,0),(3,0),(3,4)>, <(0,0),(3,0),(0,3)>}
- OPEN = {<(0,0),(3,0),(0,0)>, <(0,0),(3,0),(3,4)>, <(0,0),(3,0),(0,3)>, <(0,0),(0,4),(0,0)>, <(0,0),(0,4),(3,4)>, <(0,0),(0,4),(3,1)>}

Breadth-First Search



- Above we indicate only the state that each node terminates at. The path represented by each node is the path from the root to that node.
- Breadth-First explores the search space level by level.

Breadth-First Properties

Completeness?

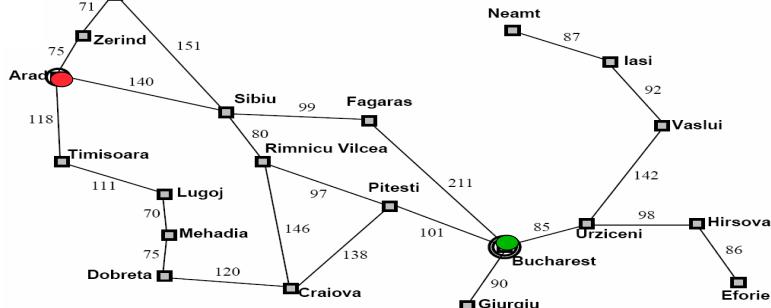
- The length of the path removed from OPEN is non-decreasing.
 - we replace each expanded node n with an extension of n .
 - All shorter paths are expanded prior before any longer path.
- Hence, eventually we must examine all paths of length d , and thus find a solution if one exists.

Optimality?

- By the above will find shortest length solution
 - least cost solution?
 - Not necessarily: shortest solution not always cheapest solution if actions have varying costs

Breadth first Solution: Arad -> Sibiu -> Fagaras -> Bucharest
Cost: $140 + 99 + 211 = 450$

Lowest cost Solution: Arad -> Sibiu -> Rimnicu Vilcea -> Pitesti -> Bucharest
Cost: $140 + 80 + 97 + 101 = 418$



Breadth-First Properties

Measuring time and space complexity.

- let b be the maximum number of successors of any node (maximal branching factor).
- let d be the depth of the shortest solution.
 - Root at depth 0 is a path of length 1
 - So $d = \text{length of path} - 1$

Time Complexity?

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

Breadth-First Properties

Space Complexity?

- $O(b^{d+1})$: If goal node is last node at level d , all of the successors of the other nodes will be on OPEN when the goal node is expanded $b(b^d - 1)$

Breadth-First Properties

Space complexity is a real problem.

- E.g., let $b = 10$, and say 100,000 nodes can be expanded per second and each node requires 100 bytes of storage:

Depth	Nodes	Time	Memory
1	1	0.01 millisec.	100 bytes
6	10^6	10 sec.	100 MB
8	10^8	17 min.	10 GB
9	10^9	3 hrs.	100 GB

- Typically run out of space before we run out of time in most applications.

Uniform-Cost Search

Uniform-Cost Search

- Keep OPEN ordered by increasing cost of the path.
- Always expand the least cost path.
- Identical to Breadth first if each action has the same cost.

Uniform-Cost Properties

Completeness?

- If each transition has costs $\geq \varepsilon > 0$.
- The previous argument used for breadth first search holds: the cost of the path represented by each node n chosen to be expanded must be non-decreasing.

Optimality?

- Finds optimal solution if each transition has cost $\geq \varepsilon > 0$.
 - Explores paths in the search space in increasing order of cost. So must find minimum cost path to a goal before finding any higher costs paths.

Uniform-Cost Search. Proof of Optimality

Let us prove Optimality more formally. We will reuse this argument later on when we examine Heuristic Search

Uniform-Cost Search. Proof of Optimality

Lemma 1.

Let $c(n)$ be the cost of node n on OPEN (cost of the path represented by n). If n_2 is expanded IMMEDIATELY after n_1 then
 $c(n_1) \leq c(n_2)$.

Proof: there are 2 cases:

- a. n_2 was on OPEN when n_1 was expanded:
We must have $c(n_1) \leq c(n_2)$ otherwise n_2 would have been selected for expansion rather than n_1
- b. n_2 was added to OPEN when n_1 was expanded
Now $c(n_1) < c(n_2)$ since the path represented by n_2 extends the path represented by n_1 and thus cost at least ε more.

Uniform-Cost Search. Proof of Optimality

Lemma 2.

When node n is expanded every path in the search space with cost strictly less than $c(n)$ has already been expanded.

Proof:

- Let $n_k = \langle \text{Start}, s_1, \dots, s_k \rangle$ be a path with cost less than $c(n)$. Let $n_0 = \langle \text{Start} \rangle$, $n_1 = \langle \text{Start}, s_1 \rangle$, $n_2 = \langle \text{Start}, s_1, s_2 \rangle, \dots, n_i = \langle \text{Start}, s_1, \dots, s_i \rangle, \dots, n_k = \langle \text{Start}, s_1, \dots, s_k \rangle$. Let n_i be the last node in this sequence that has already been expanded by search.
- So, n_{i+1} must still be on OPEN: it was added to open when n_i was expanded. Also $c(n_{i+1}) \leq c(n_k) < c(n)$: $c(n_{i+1})$ is a subpath of n_k we have assumed that $c(n_k) < c(n)$.
- But then uniform-cost would have expanded n_{i+1} not n .
- So every node n_i including n_k must already be expanded, i.e., this lower cost path has already been expanded.

Uniform-Cost Search. Proof of Optimality

Lemma 3.

The first time uniform-cost expands a node n terminating at state S , it has found the minimal cost path to S (it might later find other paths to S but none of them can be cheaper).

Proof:

- All cheaper paths have already been expanded, none of them terminated at S .
- All paths expanded after n will be at least as expensive, so no cheaper path to S can be found later.

So, when a path to a goal state is expanded the path must be optimal (lowest cost).

Uniform-Cost Properties

Time and Space Complexity?

- $O(b^{C^*/\epsilon})$ where C^* is the cost of the optimal solution.
- There may be many paths with cost $\leq C^*$: there can be as many as b^d paths of length d in the worst case.

Paths with cost lower than C^* can be as long as C^*/ϵ (why no longer?), so might have $b^{C^*/\epsilon}$ paths to explore before finding an optimal cost path.

Depth-First Search

Depth-First Search

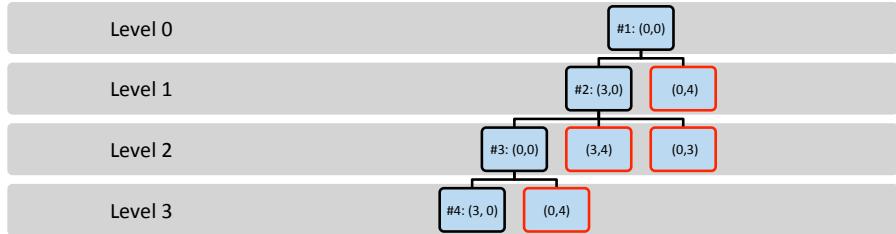
- Place the new paths that extend the current path at the **front** of OPEN.

WaterJugs. Start = (0,0), Goal = (*,2)

Green = Newly Added.

- OPEN = {<(0,0)>}
- OPEN = {<(0,0), (3,0)>, <(0,0), (0,4)>}
- OPEN = {<(0,0),(3,0),(0,0)>, <(0,0),(3,0),(3,4)>, <(0,0),(3,0),(0,3)>, <(0,4),(0,0)>}
- OPEN = {<(0,0),(3,0),(0,0),(3,0)>, <(0,0),(3,0),(0,0),(0,4)> <(0,0), (3,0), (3,4)>, <(0,0),(3,0),(0,3)>, <(0,0),(0,4)>}

Depth-First Search



- Red nodes are backtrack points (these nodes remain on open).

Depth-First Properties

Completeness?

- Infinite paths? Cause incompleteness!
- Prune paths with cycles (duplicate states)
We get completeness if state space is finite

Optimality?

No!

Depth-First Properties

Time Complexity?

- $O(b^m)$ where m is the length of the longest path in the state space.
- Very bad if m is much larger than d (shortest path to a goal state), but if there are many solution paths it can be much faster than breadth first. (Can by good luck bump into a solution quickly).

Depth-First Properties

- Depth-First Backtrack Points = unexplored siblings of nodes along current path.
 - These are the nodes that remain on open after we extract a node to expand.

Space Complexity?

- $O(bm)$, linear space!
 - Only explore a single path at a time.
 - OPEN only contains the deepest node on the current path along with the **backtrack** points.
- A significant advantage of DFS

Depth-Limited Search

Depth Limited Search

- Breadth first has space problems. Depth first can run off down a very long (or infinite) path.

Depth limited search

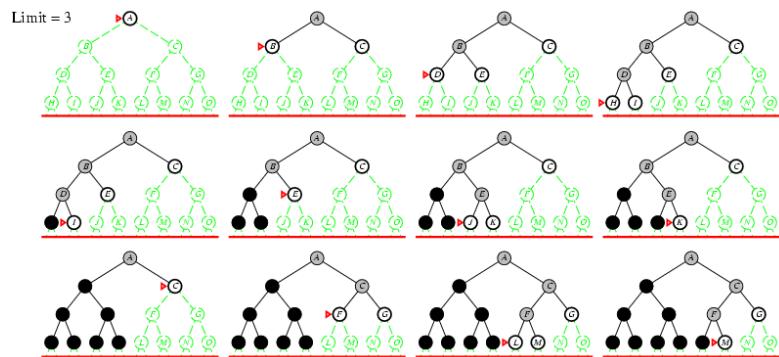
- Perform depth first search but only to a pre-specified depth limit D.
 - THE ROOT is at DEPTH 0. ROOT is a path of length 1.
 - No node representing a path of length more than $D+1$ is placed on OPEN.
 - We “truncate” the search by looking only at paths of length $D+1$ or less.
- Now infinite length paths are not a problem.
- **But will only find a solution if a solution of $\text{DEPTH} \leq D$ exists.**

Depth Limited Search

```
DLS(OPEN, Successors, Goal?) /* Call with OPEN = {<START>} */  
  WHILE(OPEN not EMPTY) {  
    n= select first node from OPEN  
    Curr = terminal state of n  
    If(Goal?(Curr)) return n  
  
    If Depth(n) < D //Don't add successors if Depth(n) = D  
      OPEN = (OPEN- {n}) Us ∈ Successors(Curr) <n,s>  
    Else  
      OPEN = OPEN - {n}  
      CutOffOccured = TRUE.  
  }  
  return FAIL
```

We will use CutOffOccured later.

Depth Limited Search Example



CSC384, University of Toronto

57

Iterative Deepening Search

CSC384, University of Toronto

58

Iterative Deepening Search

- Solve the problems of depth-first and breadth-first by extending depth limited search
- Starting at depth limit $L = 0$, we iteratively increase the depth limit, performing a depth limited search for each depth limit.
- Stop if a solution is found, or if the depth limited search failed without cutting off any nodes because of the depth limit.
 - If no nodes were cut off, the search examined all paths in the state space and found no solution → no solution exists.

CSC384, University of Toronto

59

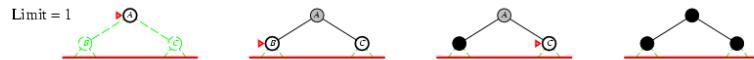
Iterative Deepening Search Example

Limit = 0 

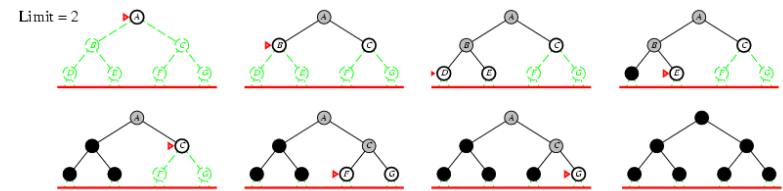
CSC384, University of Toronto

60

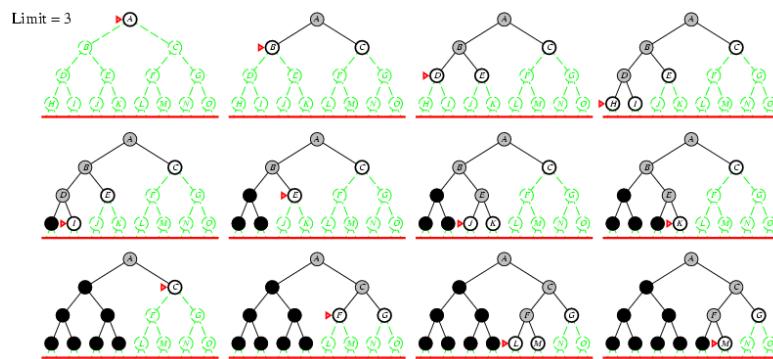
Iterative Deepening Search Example



Iterative Deepening Search Example



Iterative Deepening Search Example



Iterative Deepening Search Properties

Completeness?

- Yes if a minimal depth solution of depth d exists.
 - What happens when the depth limit $L=d$?
 - What happens when the depth limit $L < d$?

Time Complexity?

Iterative Deepening Search Properties

Time Complexity

- $(d+1)b^0 + db^1 + (d-1)b^2 + \dots + b^d = O(b^d)$
- E.g. $b=4, d=10$
 - $(11)*4^0 + 10*4^1 + 9*4^2 + \dots + 4^{10} = 1,864,131$
 - $4^{10} = 1,048,576$
- Most nodes lie on bottom layer.

BFS can explore more states than IDS!

- For IDS, the time complexity is
 - $(d+1)b^0 + db^1 + (d-1)b^2 + \dots + b^d = O(b^d)$
 - For BFS, the time complexity is
 - $1 + b + b^2 + b^3 + \dots + b^d + b(b^d - 1) = O(b^{d+1})$
- E.g. $b=4, d=10$
- For IDS
 - $(11)*4^0 + 10*4^1 + 9*4^2 + \dots + 4^{10} = 1,864,131$ (states generated)
 - For BFS
 - $1 + 4 + 4^2 + \dots + 4^{10} + 4(4^{10} - 1) = 5,592,401$ (states generated)
 - In fact IDS can be more efficient than breadth first search: nodes at limit are not expanded. BFS must expand all nodes until it expands a goal node. So at the bottom layer it will add many nodes to OPEN before finding the goal node.

Iterative Deepening Search Properties

Space Complexity

- $O(bd)$ Still linear!

Optimal?

- Will find shortest length solution which is optimal if costs are uniform.
- If costs are not uniform, we can use a “cost” bound instead.
 - Only expand paths of cost less than the cost bound.
 - Keep track of the minimum cost unexpanded path in each depth first iteration, increase the cost bound to this on the next iteration.
- This can be more expensive. Need as many iterations of the search as there are distinct path costs.

Path/Cycle Checking

Path Checking

If n_k represents the path $\langle s_0, s_1, \dots, s_k \rangle$ and we expand s_k to obtain child c , we have

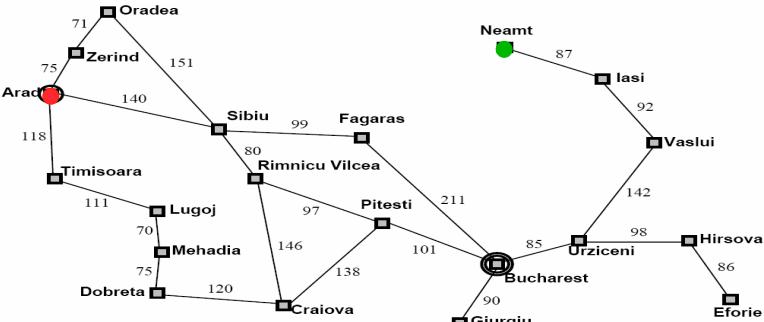
$\langle s_0, s_1, \dots, s_k, c \rangle$

As the path to “ c ”.

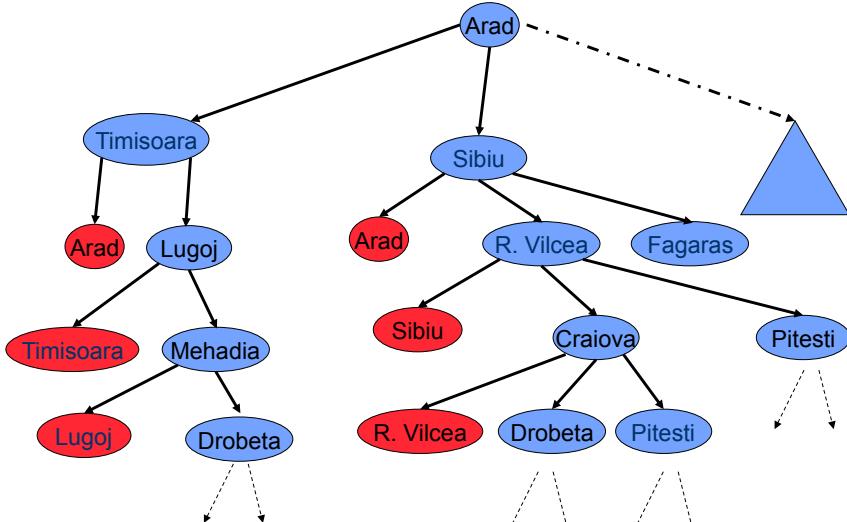
Path checking:

- Ensure that the state c is not equal to the state reached by any ancestor of c along this path.
- Paths are checked in isolation!

Example: Arad to Neamt



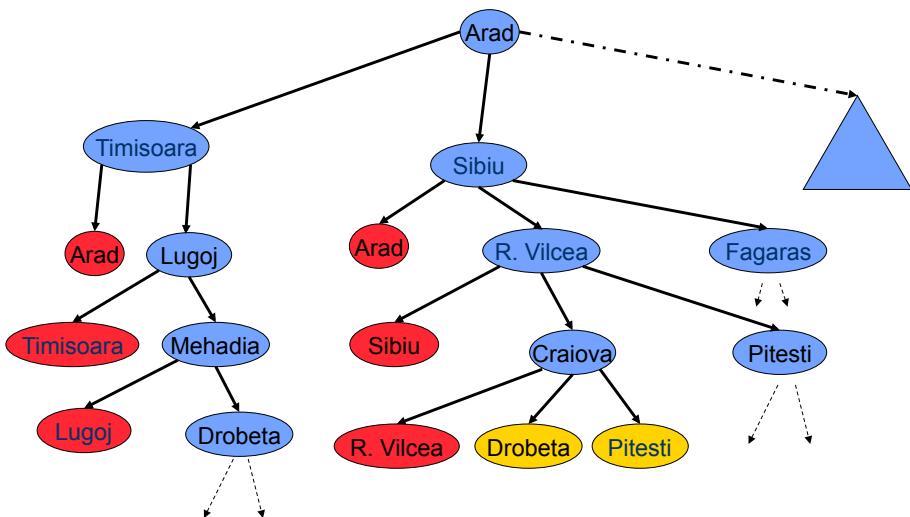
Path Checking Example



Cycle Checking

- Cycle Checking
- Keep track of **all states** previously expanded during the search.
 - When we expand n_k to obtain child c
 - Ensure that c is not equal to **any** previously expanded state.
 - This is called **cycle checking**, or **multiple path checking**.
 - What happens when we utilize this technique with depth-first search?
 - **What happens to space complexity?**

Cycle Checking Example (BFS)



CSC384, University of Toronto

73

Cycle Checking

- Higher space complexity (equal to the space complexity of breadth-first search).
- There is an additional issue when we are looking for an optimal solution
 - With uniform-cost search, we still find an optimal solution
 - **The first time uniform-cost expands a state it has found the minimal cost path to it.**
 - This means that the nodes rejected subsequently by cycle checking can't have better paths.
 - We will see later that we don't always have this property when we do heuristic search.

CSC384, University of Toronto

74

Heuristic Search (Informed Search)

CSC384, University of Toronto

75

Heuristic Search

- In **uninformed search**, we don't try to evaluate which of the nodes on OPEN are most promising. We never "look-ahead" to the goal.
E.g., in uniform cost search we always expand the cheapest path. We don't consider the cost of getting to the goal from the end of the current path.
- Often we have some other knowledge about the merit of nodes, e.g., going the wrong direction in Romania.

CSC384, University of Toronto

76

Heuristic Search

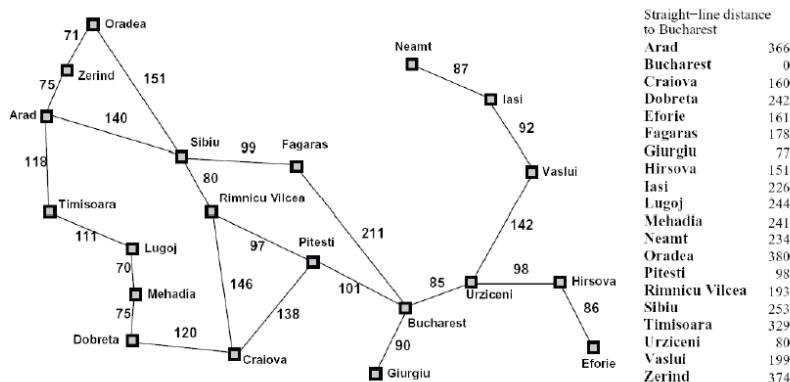
Merit of an OPEN node: different notions of merit.

- If we are concerned about the **cost of the solution**, we might want to consider how costly it is to get to the goal from the terminal state of that node.
- If we are concerned about **minimizing computation** in search we might want to consider how easy it is to find the goal from the terminal state of that node.
- We will focus on the “**cost of solution**” notion of merit.

Heuristic Search

- The idea is to develop a domain specific heuristic function $h(n)$.
- $h(n)$ **guesses** the cost of getting to the goal from node n (i.e., from the terminal state of the path represented by n).
- There are different ways of guessing this cost in different domains.
 - heuristics are **domain specific**.

Example: Euclidean distance



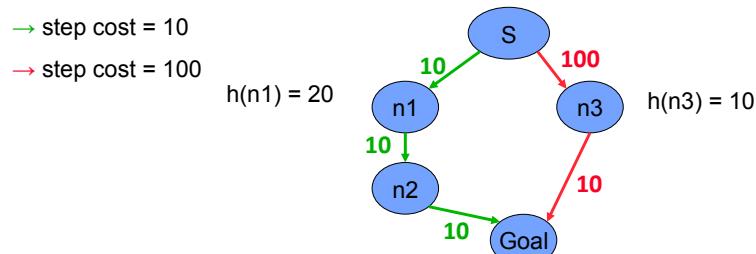
Planning a path from Arad to Bucharest, we can utilize the **straight line distance from each city to our goal**. This lets us plan our trip by picking cities at each time point that minimize the distance to our goal.

Heuristic Search

- If $h(n_1) < h(n_2)$ this means that we guess that it is cheaper to get to the goal from n_1 than from n_2 .
- We require that
 - $h(n) = 0$ for every node n whose terminal state satisfies the goal.
 - Zero cost of achieving the goal from node that already satisfies the goal.

Using only $h(n)$: Greedy best-first search

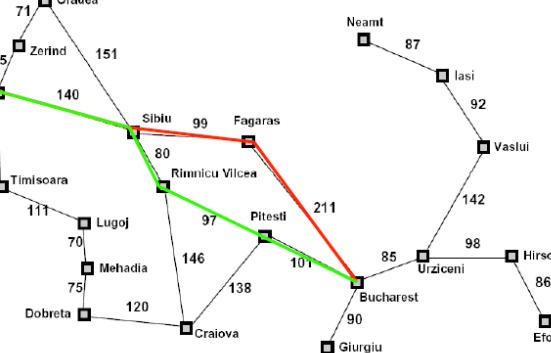
- We use $h(n)$ to rank the nodes on OPEN
 - Always expand node with lowest h -value.
- We are greedily trying to achieve a low cost solution.
- However, this method **ignores the cost of getting to n** , so it can be lead astray exploring nodes that cost a lot but seem to be close to the goal:



CSC384, University of Toronto

81

Greedy best-first search example



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

CSC384, University of Toronto

82

A* search

- Take into account the cost of getting to the node as well as our estimate of the cost of getting to the goal from the node.
- Define an evaluation function $f(n)$

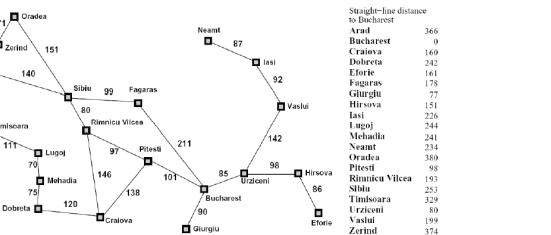
$$f(n) = g(n) + h(n)$$
 - $g(n)$ is the cost of the path represented by node n
 - $h(n)$ is the heuristic estimate of the cost of achieving the goal from n .
- Always expand the node with lowest f -value on OPEN.
- The f -value, $f(n)$ is an estimate of the cost of getting to the goal via the node ($path$) n .
 - I.e., we first follow the path n then we try to get to the goal. $f(n)$ estimates the total cost of such a solution.

CSC384, University of Toronto

83

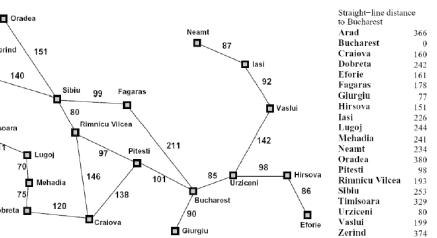
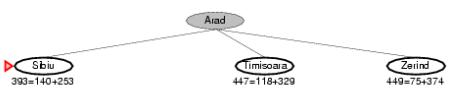
CSC384, University of Toronto

84



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

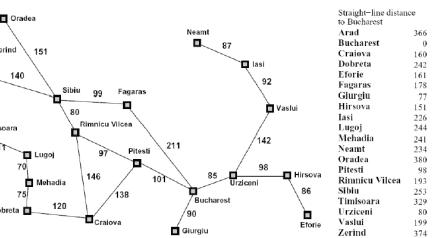
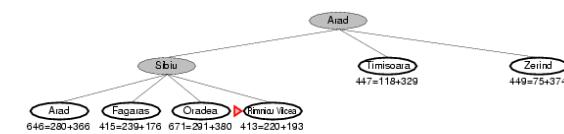
A* example



CSC384, University of Toronto

85

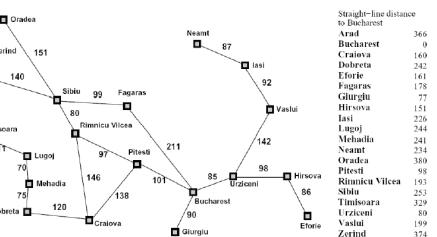
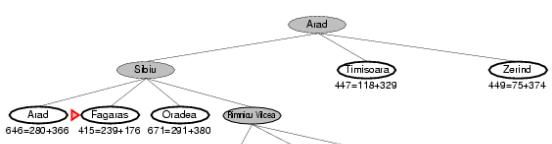
A* example



CSC384, University of Toronto

86

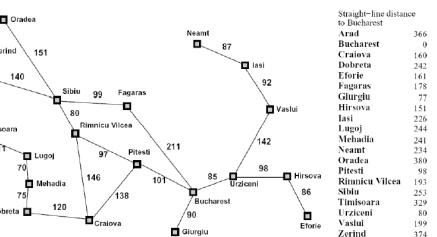
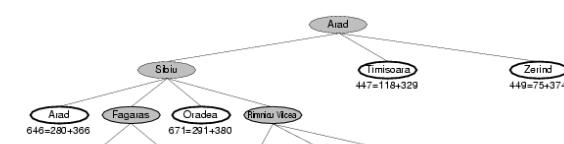
A* example



CSC384, University of Toronto

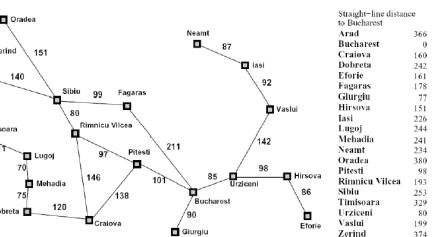
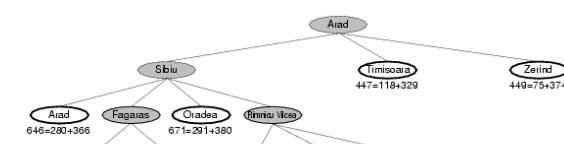
87

A* example



CSC384, University of Toronto

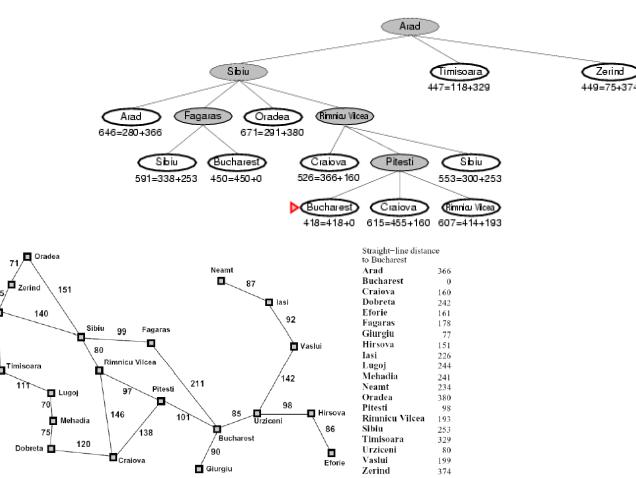
88



CSC384, University of Toronto

88

A* example



CSC384, University of Toronto

89

Properties of A* depend on conditions on h(n)

- We want to analyze the behavior of the resultant search.
 - Completeness, time and space, optimality?
- To obtain such results we must put some further conditions on the heuristic function $h(n)$ and the search space.

CSC384, University of Toronto

90

Conditions on h(n): Admissible

- We always assume that $c(s_1, a, s_2) \geq \epsilon > 0$ for any two states s_1 and s_2 and any action a : the cost of any transition is greater than zero and can't be arbitrarily small.

Let $h^*(n)$ be the **cost of an optimal path** from n to a goal node (∞ if there is no path). Then an **admissible** heuristic satisfies the condition

$$h(n) \leq h^*(n)$$

- an admissible heuristic **never over-estimates** the cost to reach the goal, i.e., it is **optimistic**
- Hence $h(g) = 0$, for any goal node g
- Also $h^*(n) = \infty$ if there is no path from n to a goal node

CSC384, University of Toronto

91

Consistency (aka monotonicity)

- A stronger condition than $h(n) \leq h^*(n)$.
- A **monotone/consistent** heuristic satisfies the triangle inequality: for all nodes n_1, n_2 and for all actions a

$$h(n_1) \leq C(n_1, a, n_2) + h(n_2)$$

Where $C(n_1, a, n_2)$ means the cost of getting from the terminal state of n_1 to the terminal state of n_2 via action a .

- Note that there might be more than one transition (action) between n_1 and n_2 , the inequality must hold for all of them.
- Monotonicity implies admissibility.
 - $(\text{forall } n_1, n_2, a) h(n_1) \leq C(n_1, a, n_2) + h(n_2) \rightarrow (\text{forall } n) h(n) \leq h^*(n)$

CSC384, University of Toronto

92

Consistency → Admissible

- Assume consistency: $h(n) \leq c(n,a,n2) + h(n2)$

Prove admissible: $h(n) \leq h^*(n)$

Proof:

Let $n \rightarrow n_1 \rightarrow \dots \rightarrow n^*$ be an OPTIMAL path from n to a goal (with actions a_1, a_2). Note the cost of this path is $h^*(n)$, and each subpath $(n_i \rightarrow \dots \rightarrow n^*)$ has cost equal to $h^*(n_i)$.

If no path exists from n to a goal then $h^*(n) = \infty$ and $h(n) \leq h^*(n)$

Otherwise prove $h(n) \leq h^*(n)$ by induction on the length of this optimal path.

Base Case: $n = n^*$

By our conditions on h , $h(n) = 0 \leq h(n)^* = 0$

Induction Hypothesis: $h(n_1) \leq h^*(n_1)$

$$h(n) \leq c(n,a_1,n_1) + h(n_1) \leq c(n,a_1,n_1) + h^*(n_1) = h^*(n)$$

Intuition behind admissibility

$h(n) \leq h^*(n)$ means that the search won't miss any promising paths.

- If it really is cheap to get to a goal via n (i.e., both $g(n)$ and $h^*(n)$ are low), then $f(n) = g(n) + h(n)$ will also be low, and the search won't ignore n in favor of more expensive options.
- This can be formalized to show that admissibility implies optimality.
- Monotonicity gives some additional properties when it comes to cycle checking.

Consequences of monotonicity

- The f -values of nodes along a path must be non-decreasing.

Let $\langle \text{Start} \rightarrow s_1 \rightarrow s_2 \rightarrow \dots \rightarrow s_k \rangle$ be a path. Let n_i be the subpath $\langle \text{Start} \rightarrow s_1 \rightarrow \dots \rightarrow s_i \rangle$:

We claim that: $f(n_i) \leq f(n_{i+1})$

Proof

$$\begin{aligned} f(n_i) &= c(\text{Start} \rightarrow \dots \rightarrow n_i) + h(n_i) \\ &\leq c(\text{Start} \rightarrow \dots \rightarrow n_i) + c(n_i \rightarrow n_{i+1}) + h(n_{i+1}) \\ &\leq c(\text{Start} \rightarrow \dots \rightarrow n_i \rightarrow n_{i+1}) + h(n_{i+1}) \\ &\leq g(n_{i+1}) + h(n_{i+1}) = f(n_{i+1}) \end{aligned}$$

Consequences of monotonicity

- If n_2 is expanded immediately after n_1 , then $f(n_1) \leq f(n_2)$

(the f -value of expanded nodes is **monotonic** non-decreasing)

Proof:

- If n_2 was on OPEN when n_1 was expanded, then $f(n_1) \leq f(n_2)$ otherwise we would have expanded n_2 .
- If n_2 was added to OPEN after n_1 's expansion, then n_2 extends n_1 's path. That is, the path represented by n_1 is a prefix of the path represented by n_2 . By property (1) we have $f(n_1) \leq f(n_2)$ as the f -values along a path are non-decreasing.

Consequences of monotonicity

3. Corollary: the sequence of f-values of the nodes expanded by A* is non-decreasing. I.e, If n_2 is expanded **after** (not necessarily immediately after) n_1 , then $f(n_1) \leq f(n_2)$
(the f-value of expanded nodes is **monotonic** non-decreasing)

Proof:

- If n_2 was on OPEN when n_1 was expanded, then $f(n_1) \leq f(n_2)$ otherwise we would have expanded n_2 .
- If n_2 was added to OPEN after n_1 's expansion, then let n be an ancestor of n_2 that was present when n_1 was being expanded (this could be n_1 itself). We have $f(n_1) \leq f(n)$ since A* chose n_1 while n was present on OPEN. Also, since n is along the path to n_2 , by property (1) we have $f(n) \leq f(n_2)$. So, we have $f(n_1) \leq f(n_2)$.

Consequences of monotonicity

4. When n is expanded every path with lower f-value has already been expanded.

- **Proof:** Assume by contradiction that there exists a path $\langle \text{Start}, n_0, n_1, n_{i-1}, n_i, n_{i+1}, \dots, n_k \rangle$ with $f(n_k) < f(n)$ and n_i is its last expanded node.
 - n_{i+1} must be on OPEN while n is expanded, so
 - a) by (1) $f(n_{i+1}) \leq f(n_k)$ since they lie along the same path.
 - b) since $f(n_k) < f(n)$ so we have $f(n_{i+1}) < f(n)$
 - c) by (2) $f(n) \leq f(n_{i+1})$ because n is expanded before n_{i+1} .
 - Contradiction from b&c!

Consequences of monotonicity

5. With a monotone heuristic, the first time A* expands a state, it has found the minimum cost path to that state.

Proof:

- Let $\text{PATH1} = \langle \text{Start}, s_0, s_1, \dots, s_k, s \rangle$ be **the first** path to a state s found. We have $f(\text{path1}) = c(\text{PATH1}) + h(s)$.
- Let $\text{PATH2} = \langle \text{Start}, t_0, t_1, \dots, t_j, s \rangle$ be another path to s found later. we have $f(\text{path2}) = c(\text{PATH2}) + h(s)$.
- Note $h(s)$ is dependent only on the state s (terminal state of the path) it does not depend on how we got to s .
- By property (3), $f(\text{path1}) \leq f(\text{path2})$
- hence: $c(\text{PATH1}) \leq c(\text{PATH2})$

Consequences of monotonicity

Complete.

- Yes, consider a least cost path to a goal node
 - $\text{SolutionPath} = \langle \text{Start} \rightarrow n_1 \rightarrow \dots \rightarrow G \rangle$ with cost $c(\text{SolutionPath})$. Since $h(G) = 0$, this means that $f(\text{SolutionPath}) = \text{cost}(\text{SolutionPath})$
 - Since each action has a cost $\geq \epsilon > 0$, there are only a finite number of paths that have f-value $< c(\text{SolutionPath})$. None of these paths lead to a goal node since SolutionPath is a least cost path to the goal.
 - So eventually SolutionPath , or some equal cost path to a goal must be expanded.

Time and Space complexity.

- When $h(n) = 0$, for all n h is monotone.
 - A* becomes uniform-cost search!
- It can be shown that when $h(n) > 0$ for some n and still admissible, the number of nodes expanded can be no larger than uniform-cost.
- Hence the same bounds as uniform-cost apply. (These are worst case bounds). Still exponential unless we have a very good h !
- In real world problems, we sometimes run out of time and memory. IDA* can sometimes be used to address memory issues, but IDA* isn't very good when many cycles are present.

Consequences of monotonicity

Optimality

- Yes, by (5) the first path to a goal node must be optimal.

5. With a monotone heuristic, the first time A* expands a state, it has found the minimum cost path to that state.

Cycle Checking

- We can use a simple implementation of cycle checking (multiple path checking)---just reject all search nodes visiting a state already visited by a previously expanded node. By property (5) we need keep only the first path to a state, rejecting all subsequent paths.

Admissibility without monotonicity

When "h" is admissible but not monotonic.

- Time and Space complexity remain the same. Completeness holds.
- Optimality still holds (without cycle checking), but need a different argument: don't know that paths are explored in order of cost.

Admissibility without monotonicity

What about Cycle Checking?

- No longer guaranteed we have found an optimal path to a node *the first time* we visit it.
- So, cycle checking might not preserve optimality.
 - To fix this: for previously visited nodes, must remember cost of previous path. If new path is cheaper must explore again.

Space Problems with A*

- A* has the same potential space problems as BFS or UCS
- IDA* - Iterative Deepening A* is similar to Iterative Deepening Search and similarly addresses space issues.

IDA* - Iterative Deepening A*

Objective: reduce memory requirements for A*

- Like iterative deepening, but now the “cutoff” is the f-value ($g+h$) rather than the depth
- At each iteration, the cutoff value is the smallest f-value of any node that exceeded the cutoff on the previous iteration
- Avoids overhead associated with keeping a sorted queue of nodes, and the open list occupies only linear space.
- Two new parameters:
 - curBound (any node with a bigger f-value is discarded)
 - smallestNotExplored (the smallest f-value for discarded nodes in a round) when OPEN becomes empty, the search starts a new round with this bound.
 - Easier to expand all nodes with f-value EQUAL to the f-limit. This way we can compute “smallestNotExplored” more easily.

Constructing Heuristics

Building Heuristics: Relaxed Problem

- One useful technique is to consider an easier problem, and let $h(n)$ be the cost of reaching the goal in the easier problem.
- 8-Puzzle moves.
 - Can move a tile from square A to B if
 - A is adjacent (left, right, above, below) to B
 - and B is blank
- Can relax some of these conditions
 1. can move from A to B if A is adjacent to B (ignore whether or not position is blank)
 2. can move from A to B if B is blank (ignore adjacency)
 3. can move from A to B (ignore both conditions).

Building Heuristics: Relaxed Problem

- #3 “can move from A to B (ignore both conditions)”.
leads to the **misplaced tiles heuristic**.
 - To solve the puzzle, we need to move each tile into its final position.
 - Number of moves = number of misplaced tiles.
 - Clearly $h(n) = \text{number of misplaced tiles} \leq h^*(n)$ the cost of an optimal sequence of moves from n.
- #1 “can move from A to B if A is adjacent to B (ignore whether or not position is blank)”
leads to the **manhattan distance heuristic**.
 - To solve the puzzle we need to slide each tile into its final position.
 - We can move vertically or horizontally.
 - Number of moves = sum over all of the tiles of the number of vertical and horizontal slides we need to move that tile into place.
 - Again $h(n) = \text{sum of the manhattan distances} \leq h^*(n)$
 - in a real solution we need to move each tile at least that far and we can only move one tile at a time.

Building Heuristics: Relaxed Problem

Comparison of IDS and A* (average total nodes expanded):

Depth	IDS	A*(Misplaced) h1	A*(Manhattan) h2
10	47,127	93	39
14	3,473,941	539	113
24	---	39,135	1,641

Let **h1**=Misplaced, **h2**=Manhattan

- Does h2 **always** expand fewer nodes than h1?
 - Yes! Note that **h2 dominates h1**, i.e. for all n: $h1(n) \leq h2(n)$. From this you can prove h2 is faster than h1 (once both are admissible).
 - Therefore, among several admissible heuristic the one with highest value is the fastest.

Building Heuristics: Relaxed Problem

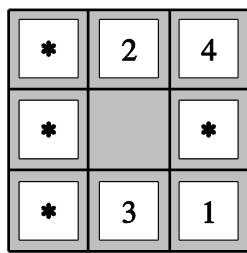
The **optimal** cost to nodes in the relaxed problem is an **admissible heuristic** for the original problem!

Proof Idea: the optimal solution in the original problem is a solution for relaxed problem, therefore it must be at least as expensive as the optimal solution in the relaxed problem.

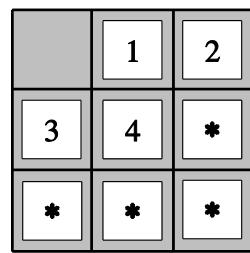
So admissible heuristics can sometimes be constructed by finding a relaxation whose optimal solution can be easily computed.

Building Heuristics: Pattern databases

- Try to generate admissible heuristics by solving a subproblem and storing the exact solution cost for that subproblem
- See Chapter 3.6.3 if you are interested.



Start State



Goal State

Constraint Satisfaction Problems (Backtracking Search)

- Chapter 6
 - 6.1: Formalism
 - 6.2: Constraint Propagation
 - 6.3: Backtracking Search for CSP
 - 6.4 is about local search which is a very useful idea but we won't cover it in class.

Representing States with Feature Vectors

- For each problem we have designed a new state representation (and designed the sub-routines called by search based on this representation).
- **Feature vectors** provide a general state representation that is useful for many different problems.
- Feature vectors are also used in many other areas of AI, particularly Machine Learning, Reasoning under Uncertainty, Computer Vision, etc.

Feature Vectors

- We have
 - A set of k variables (or features)
 - Each variable has a **domain** of different values.
 - A state is specified by an assignment of a value for each variable.
 - height = {short, average, tall},
 - weight = {light, average, heavy}
 - A partial state is specified by an assignment of a value to **some** of the variables.

Example: Sudoku

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

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

Example: Sudoku

- 81 **variables**, each representing the value of a cell.
- **Domain of Values**: a single value for those cells that are already filled in, the set $\{1, \dots, 9\}$ for those cells that are empty.
- **State**: any completed board given by specifying the value in each cell (1-9, or blank).
- **Partial State**: some incomplete filling out of the board.

Example: 8-Puzzle

2	3	7
6	4	8
5	1	

- **Variables:** 9 variables $\text{Cell}_{1,1}$, $\text{Cell}_{1,2}$, ..., $\text{Cell}_{3,3}$
- **Values:** {'B', 1, 2, ..., 8}
- **State:** Each “ $\text{Cell}_{i,j}$ ” variable specifies what is in that position of the tile.
 - If we specify a value for each cell we have completely specified a state.

This is only one of many ways to specify the state.

Constraint Satisfaction Problems

- Notice that in these problems some settings of the variables are **illegal**.
 - In Sudoku, we can't have the same number in any column, row, or subsquare.
 - In the 8 puzzle each variable must have a distinct value (same tile can't be in two places)

Constraint Satisfaction Problems

- In many practical problems finding which setting of the feature variables yields a legal state is difficult.

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

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

- We want to find a state (setting of the variables) that satisfies certain constraints.

Constraint Satisfaction Problems

- In Sudoku: The variables that form
 - a column must be distinct
 - a row must be distinct
 - a sub-square must be distinct.

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

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

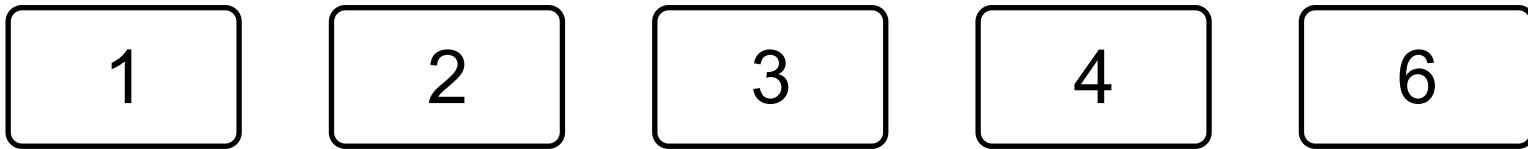
Constraint Satisfaction Problems

- In these problems we **do not care** about the sequence of moves needed to get to a goal state.
- We only care about finding a feature vector (a setting of the variables) that satisfies the goal.
 - A setting of the variables that satisfies some constraints.
- In contrast, in the 8-puzzle, the feature vector satisfying the goal is given. We care about the sequence of moves needed to move the tiles into that configuration

Example Car Sequencing

Car Factory Assembly Line—back to the days of Henry Ford

Move the items to be assembled don't move the workers



The assembly line is divided into stations. A particular task is preformed at each station.

Example Car Sequencing

1

sunroof

3

Heated
seats

6

Some stations install optional items...not every car in the assembly line is worked on in that station.

As a result the factory is designed to have lower capacity in those stations.

Example Car Sequencing



Cars move through the factory on an assembly line which is broken up into slots.

The stations might be able to process only a limited number of slots out of some group of slots that is passing through the station at any time.

E.g., the sunroof station might accommodate 4 slots, but only has capacity to process 2 slots out of the 4 at any one time.

Example Car Sequencing



Max 2

Max 2

Max 2

Max 2

Example Car Sequencing



Each car to be assembled has a list of required options. We want to assign each car to be assembled to a slot on the line.

But we want to ensure that no sequence of 4 slots has more than 2 cars assigned that require a sun roof. Finding a feasible assignment of cars with different options to slots without violating the capacity constraints of the different stations is hard.

Formalization of a CSP

- A CSP consists of
 - A set of **variables** V_1, \dots, V_n
 - For each variable a (finite) **domain** of possible values $\text{Dom}[V_i]$.
 - A set of **constraints** C_1, \dots, C_m .
- A **solution** to a CSP is an **assignment** of a value to all of the variables such that **every constraint is satisfied**.
- A CSP is unsatisfiable if no solution exists.

Formalization of a CSP

- Each variable can be assigned any value from its domain.
 - $V_i = d$ where $d \in \text{Dom}[V_i]$
- Each constraint C
 - Has a set of variables it is over, called its **scope**
 - E.g., $C(V_1, V_2, V_4)$ is a constraint over the variables V_1 , V_2 , and V_4 . Its scope is $\{V_1, V_2, V_4\}$
 - Given an assignment to its variables the constraint returns:
 - True—this assignment satisfies the constraint
 - False—this assignment falsifies the constraint.

Formalization of a CSP

- We can specify the constraint with a table
- $C(V1, V2, V4)$ with $\text{Dom}[V1] = \{1, 2, 3\}$ and $\text{Dom}[V2] = \text{Dom}[V4] = \{1, 2\}$

V1	V2	V4	$C(V1, V2, V4)$
1	1	1	False
1	1	2	False
1	2	1	False
1	2	2	False
2	1	1	True
2	1	2	False
2	2	1	False
2	2	2	False
3	1	1	False
3	1	2	True
3	2	1	True
3	2	2	False

Formalization of a CSP

- Often we can specify the constraint more compactly with an expression:
 $C(V1, V2, V4) = (V1 = V2+V4)$

V1	V2	V4	C(V1,V2,V4)
1	1	1	False
1	1	2	False
1	2	1	False
1	2	2	False
2	1	1	True
2	1	2	False
2	2	1	False
2	2	2	False
3	1	1	False
3	1	2	True
3	2	1	True
3	2	2	False

Formalization of a CSP

- Unary Constraints (over one variable)
 - e.g. $C(X):X=2$; $C(Y): Y>5$
- Binary Constraints (over two variables)
 - e.g. $C(X,Y): X+Y<6$
- Higher-order constraints: over 3 or more variables.

Example: Sudoku

- **Variables:** $V_{11}, V_{12}, \dots, V_{21}, V_{22}, \dots, V_{91}, \dots, V_{99}$
- **Domains:**
 - $\text{Dom}[V_{ij}] = \{1-9\}$ for empty cells
 - $\text{Dom}[V_{ij}] = \{k\}$ a fixed value k for filled cells.
- **Constraints:**
 - Row constraints:
 - All-Diff($V_{11}, V_{12}, V_{13}, \dots, V_{19}$)
 - All-Diff($V_{21}, V_{22}, V_{23}, \dots, V_{29}$)
 -, All-Diff($V_{91}, V_{92}, \dots, V_{99}$)
 - Column Constraints:
 - All-Diff($V_{11}, V_{21}, V_{31}, \dots, V_{91}$)
 - All-Diff($V_{21}, V_{22}, V_{13}, \dots, V_{92}$)
 -, All-Diff($V_{19}, V_{29}, \dots, V_{99}$)
 - Sub-Square Constraints:
 - All-Diff($V_{11}, V_{12}, V_{13}, V_{21}, V_{22}, V_{23}, V_{31}, V_{32}, V_{33}$)
 - All-Diff($V_{14}, V_{15}, V_{16}, \dots, V_{34}, V_{35}, V_{36}$)

Example: Sudoku

- Each of these constraints is over 9 variables, and they are all the same constraint:
 - Any assignment to these 9 variables such that each variable has a different value satisfies the constraint.
 - Any assignment where two or more variables have the same value falsifies the constraint.
- This is a special kind of constraint called an **ALL-DIFF** constraint.
 - ALL-Diff(V_1, \dots, V_n) could also be encoded as a set of binary not-equal constraints between all possible pairs of variables:
 $V_1 \neq V_2, V_1 \neq V_3, \dots, V_2 \neq V_1, \dots, V_n \neq V_1, \dots, V_n \neq V_{n-1}$

Example: Sudoku

- Thus Sudoku has 3×9 ALL-DIFF constraints, one over each set of variables in the same row, one over each set of variables in the same column, and one over each set of variables in the same sub-square.

Solving CSPs

- Because CSPs do not require finding a paths (to a goal), it is best solved by a specialized version of depth-first search.
- Key intuitions:
 - We can build up to a solution by searching through the space of partial assignments.
 - Order in which we assign the variables does not matter – eventually they all have to be assigned. **We can decide on a suitable value for one variable at a time!**
 - This is the key idea of backtracking search.
 - If we falsify a constraint during the process of building up a solution, we can immediately reject the current partial assignment:
 - All extensions of this partial assignment will falsify that constraint, and thus none can be solutions.

CSP as a Search Problem

A CSP could be viewed as a more traditional search problem

- **Initial state:** empty assignment
- **Successor function:** a value is assigned to any unassigned variable, which does not cause any constraint to return false.
- **Goal test:** the assignment is complete

Backtracking Search: The Algorithm BT

BT(Level)

If all variables assigned

PRINT Value of each Variable

RETURN or EXIT (RETURN for more solutions)

(EXIT for only one solution)

V := PickUnassignedVariable()

Assigned[V] := TRUE

for d := each member of Domain(V) (the domain values of V)

Value[V] := d

ConstraintsOK = TRUE

for each constraint C such that

a) V is a variable of C and

b) all other variables of C are assigned:

IF C is **not** satisfied by the set of current assignments:

ConstraintsOK = FALSE

If ConstraintsOk == TRUE:

 BT(Level+1)

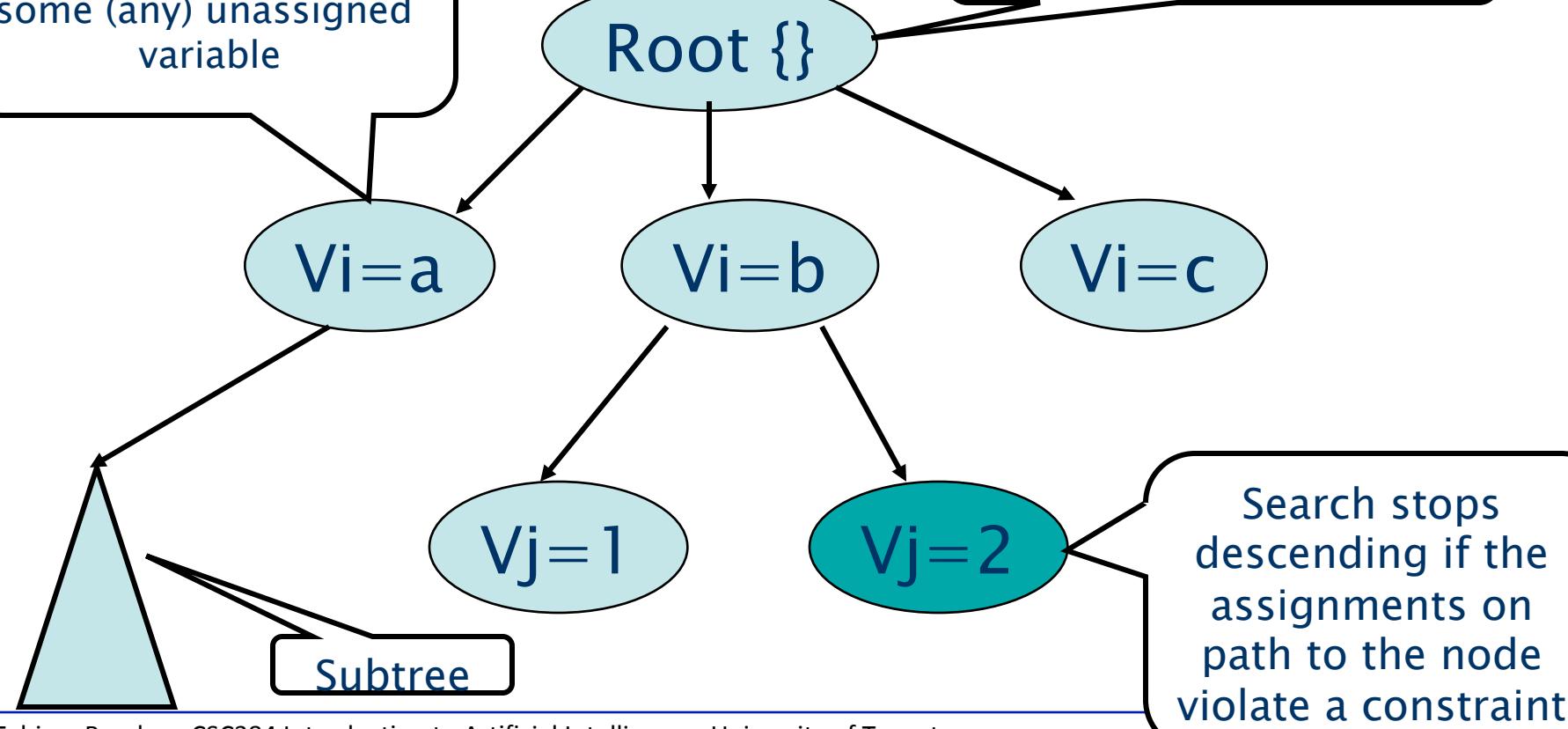
Assigned[V] := FALSE //UNDO as we have tried all of V's values
return

Backtracking Search

- The algorithm searches a tree of partial assignments.

Children of a node are all possible values of some (any) unassigned variable

The root has the empty set of assignments

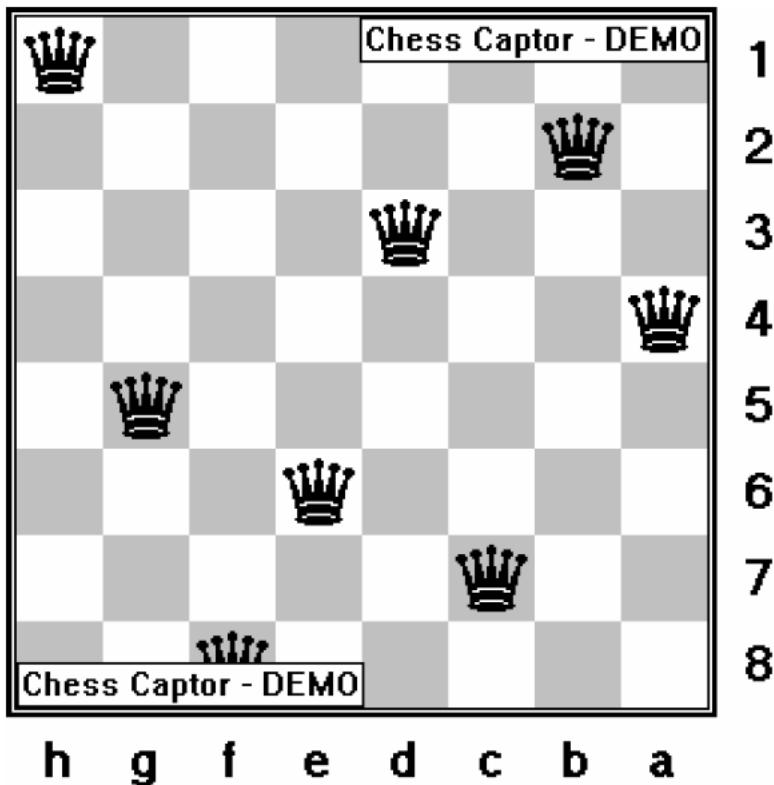


Backtracking Search

- Heuristics are used to determine
 - the order in which variables are assigned:
`PickUnassignedVariable()`
 - the order of values tried for each variable.
- The choice of the next variable can vary from branch to branch, e.g.,
 - under the assignment $V1=a$ we might choose to assign $V4$ next, while under $V1=b$ we might choose to assign $V5$ next.
- This “**dynamically**” chosen variable ordering has a tremendous impact on performance.

Example: N-Queens

- Place N Queens on an N X N chess board so that no Queen can attack any other Queen.

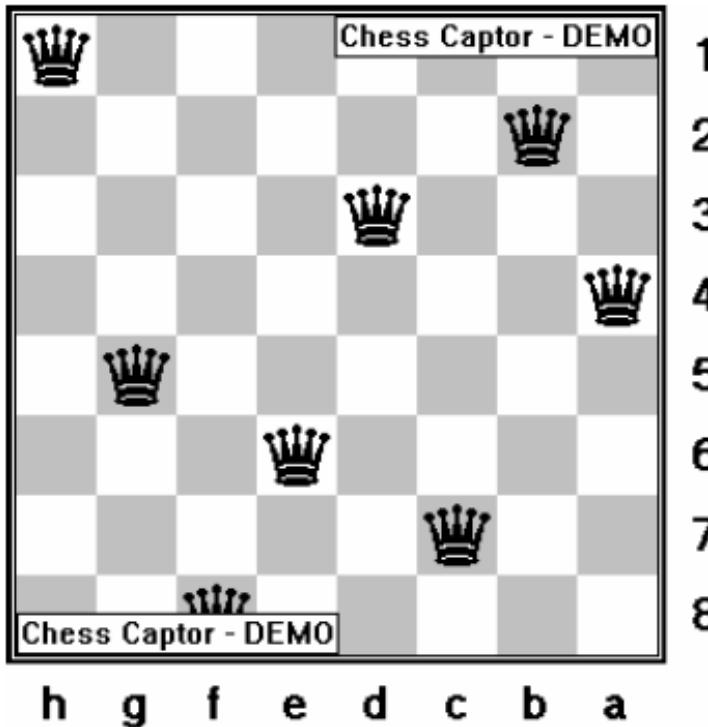


Example: N-Queens

- Problem formulation:
 - N variables (N queens)
 - N^2 values for each variable representing the positions on the chessboard
 - Value i is i 'th cell counting from the top left as 1, going left to right, top to bottom.

Example: N-Queens

- $Q1 = 1, Q2 = 15, Q3 = 21, Q4 = 32,$
 $Q5 = 34, Q6 = 44, Q7 = 54, Q8 = 59$



Example: N-Queens

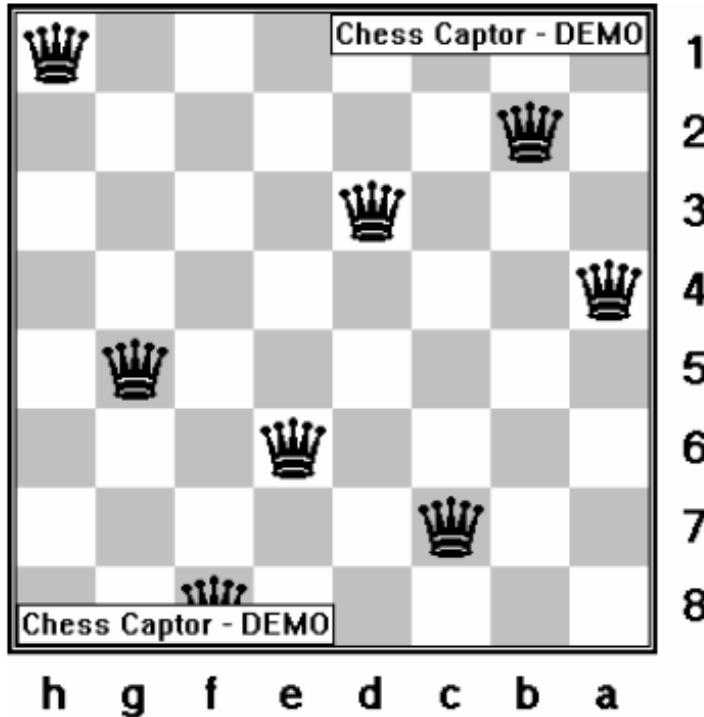
- This representation has $(N^2)^N$ states (different possible assignments in the search space)
 - For 8-Queens: $64^8 = 281,474,976,710,656$
- Is there a better way to represent the N-queens problem?
 - We know we cannot place two queens in a single row → we can exploit this fact in the choice of the CSP representation

Example: N-Queens

- Better Modeling:
 - N variables Q_i , one per row.
 - Value of Q_i is the column the Queen in row i is placed; possible values $\{1, \dots, N\}$.
- This representation has N^N states:
 - For 8-Queens: $8^8 = 16,777,216$
- The choice of a representation can make the problem solvable or unsolvable!

Example: N-Queens

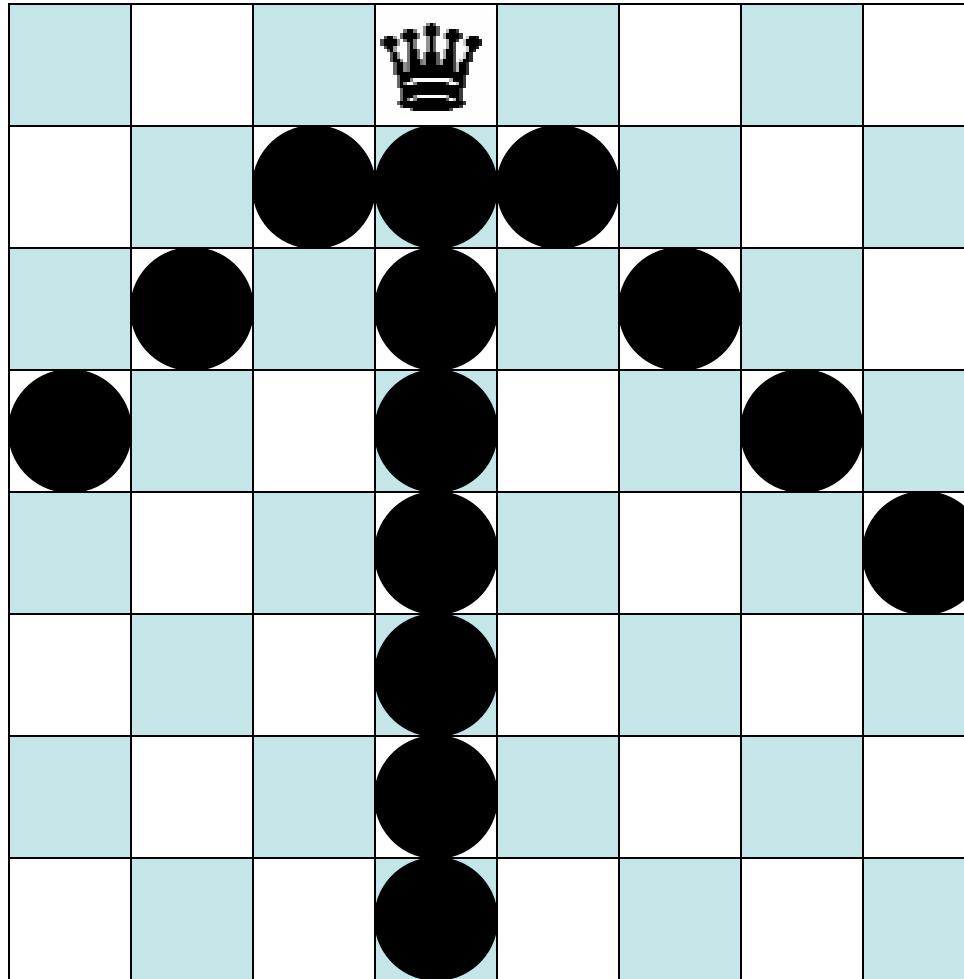
- $Q1 = 1, Q2 = 7, Q3 = 5, Q4 = 8,$
 $Q5 = 2, Q6 = 4, Q7 = 6, Q8 = 3$



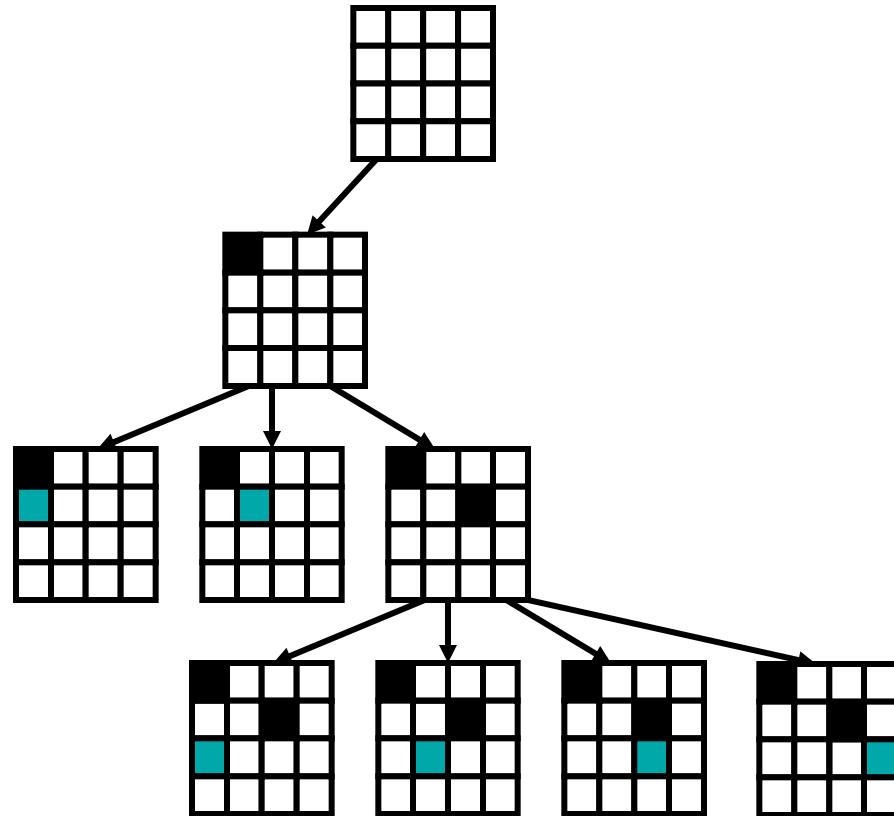
Example: N-Queens

- Constraints:
 - Can't put two Queens in same column
 $Q_i \neq Q_j$ for all $i \neq j$
 - Diagonal constraints
 $\text{abs}(Q_i - Q_j) \neq \text{abs}(i - j)$
 - i.e., the difference in the values assigned to Q_i and Q_j can't be equal to the difference between i and j .

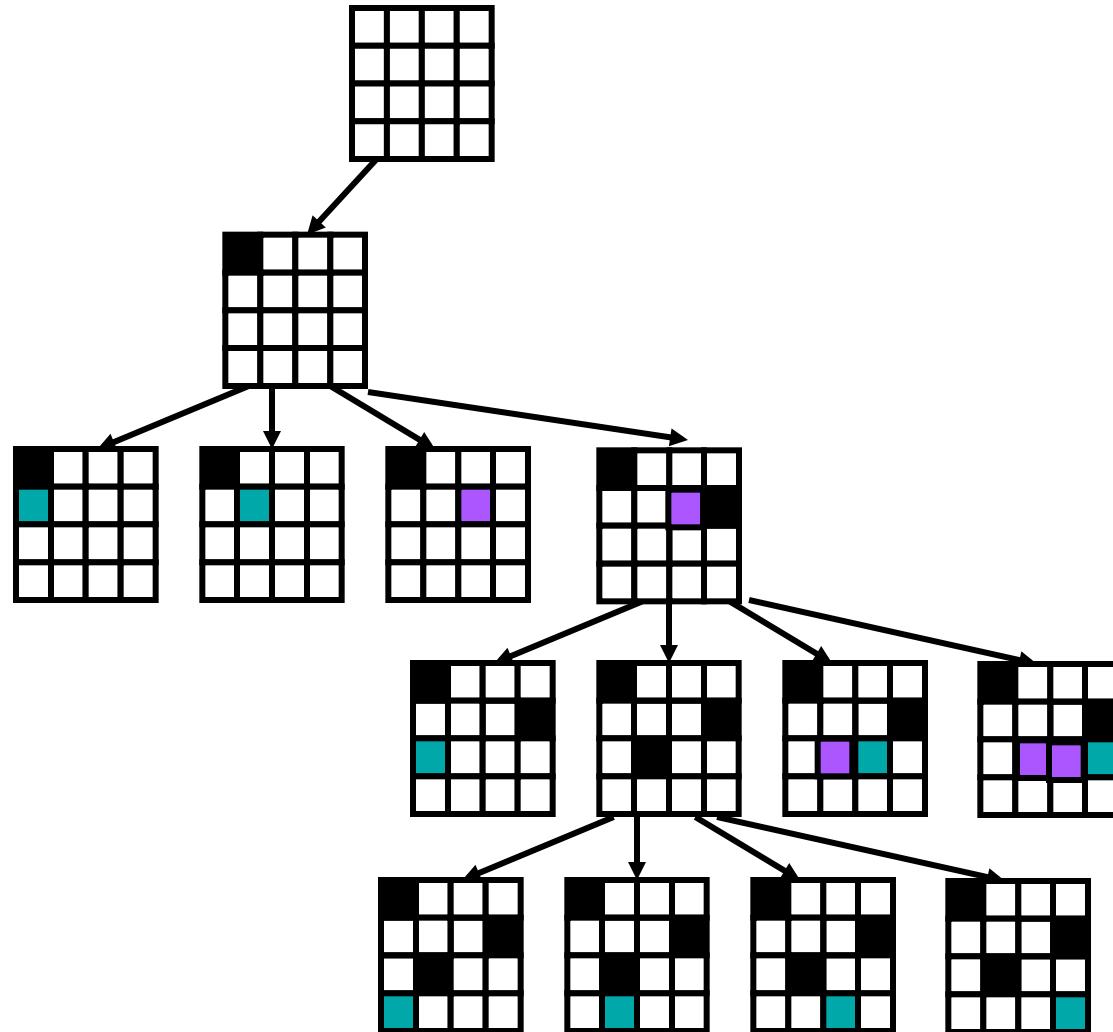
Example: N-Queens



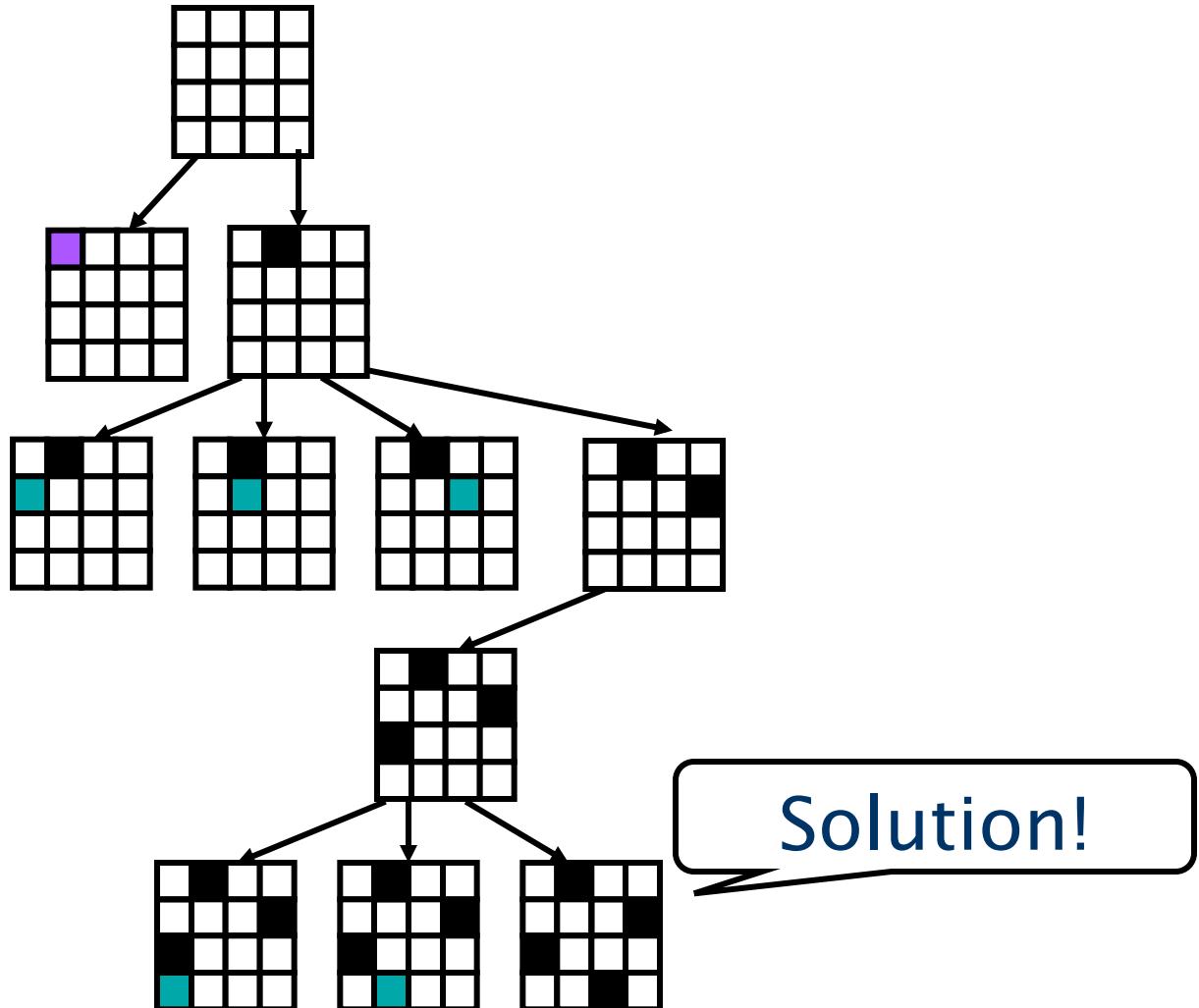
Example: N-Queens



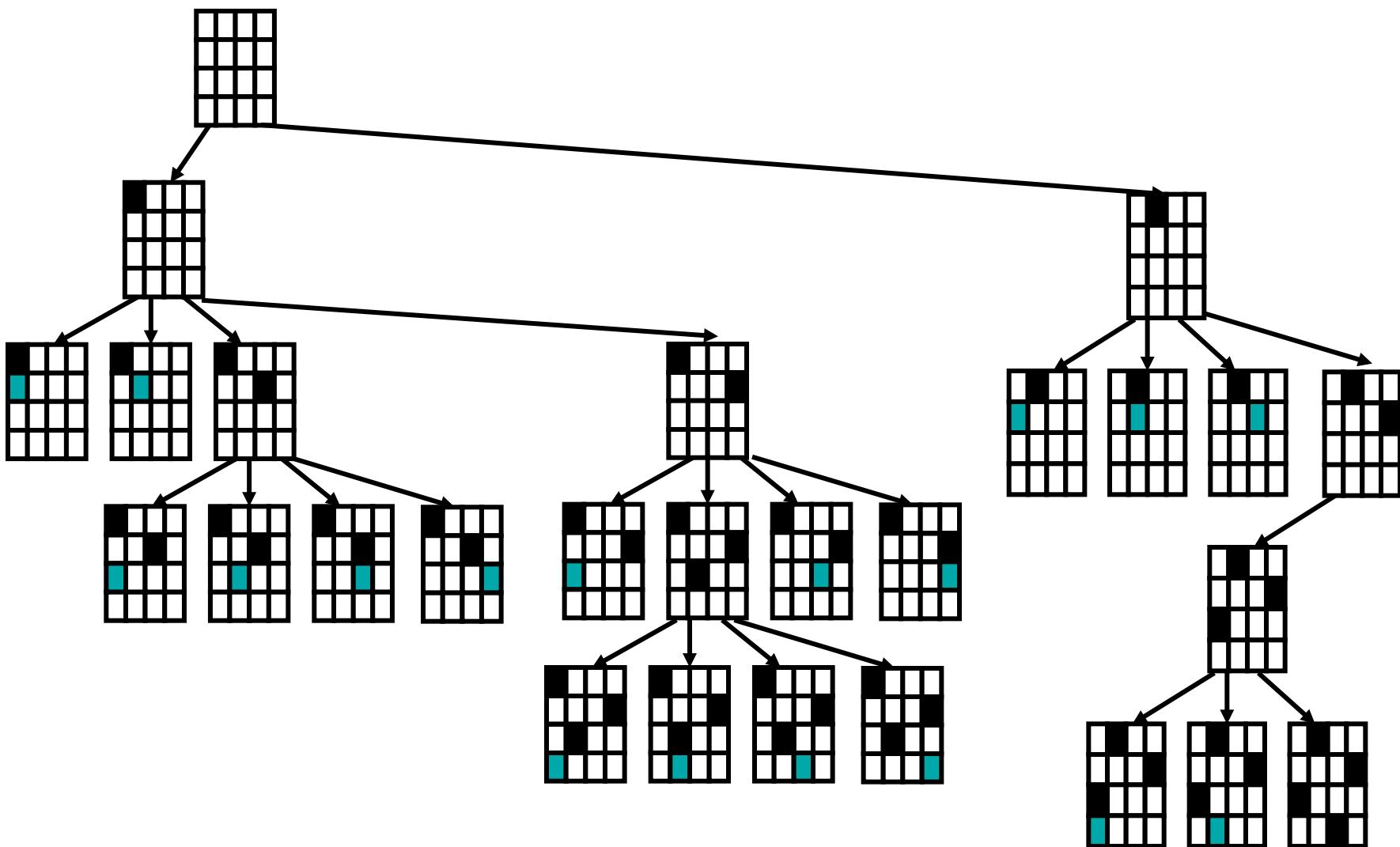
Example: N-Queens



Example: N-Queens

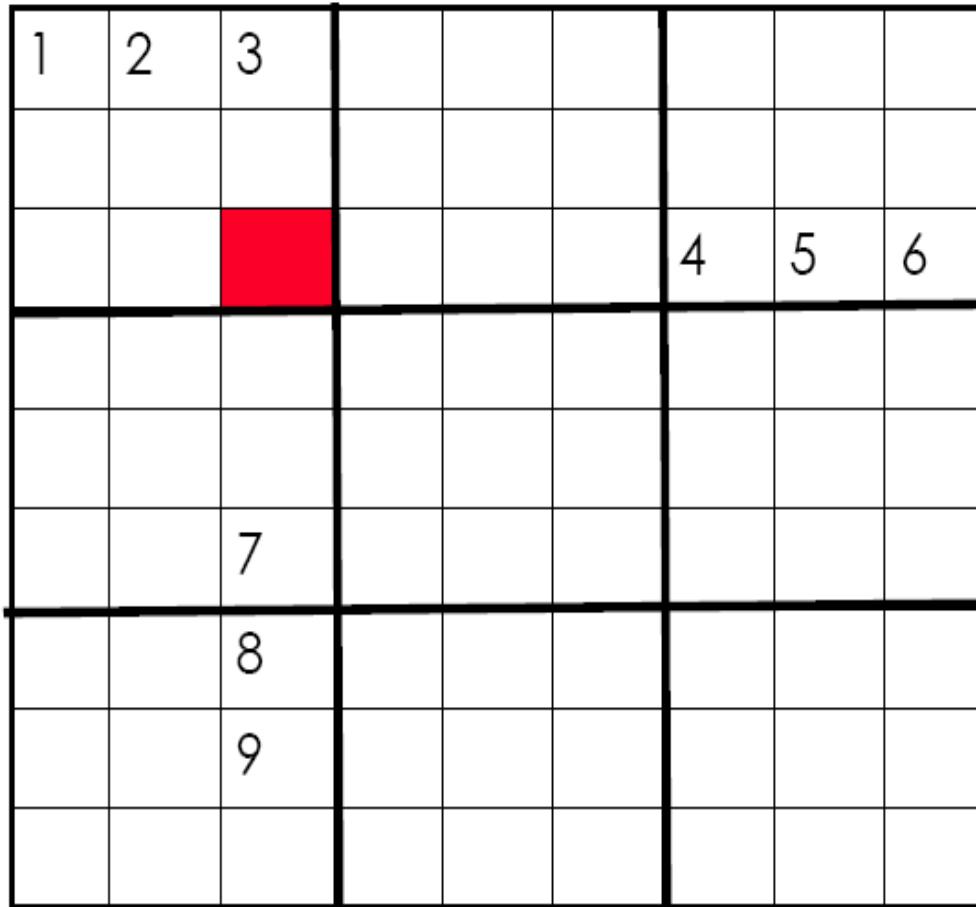


Example: N-Queens Backtracking Search Space



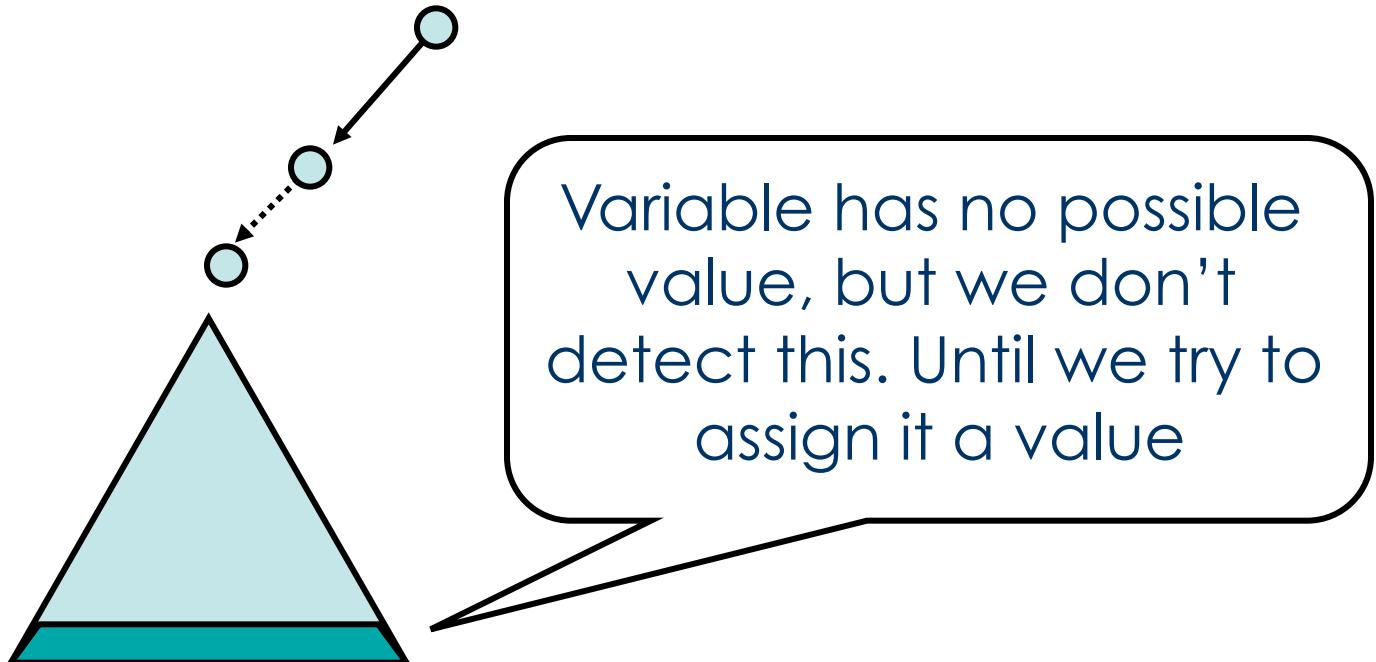
Problems with Plain Backtracking

Sudoku: The 3,3 cell has no possible value.



Problems with Plain Backtracking

- In the backtracking search we won't detect that the (3,3) cell has no possible value until all variables of the row/column (involving row or column 3) or the sub-square constraint (first sub-square) are assigned.
So we have the following situation:



- Leads to the idea of **constraint propagation**

Constraint Propagation

- Constraint propagation refers to the technique of “**looking ahead**” at the yet unassigned variables in the search .
- Try to detect obvious failures: “**Obvious**” means things we can test/detect efficiently.
- Even if we don’t detect an obvious failure we might be able to eliminate some possible part of the future search.

Constraint Propagation

- Propagation has to be applied during the search; potentially at every node of the search tree.
- Propagation itself is an inference step that needs some resources (in particular time)
 - If propagation is slow, this can slow the search down to the point where using propagation makes finding a solution take longer!
 - There is always a tradeoff between searching fewer nodes in the search, and having a higher nodes/second processing rate.
- We will look at two main types of propagation: Forward Checking & Generalized Arc Consistency

Constraint Propagation: Forward Checking

- Forward checking is an extension of backtracking search that employs a “modest” amount of propagation (look ahead).
- When a variable is instantiated we check all constraints that have **only one uninstantiated variable** remaining.
- For that uninstantiated variable, we check all of its values, pruning those values that violate the constraint.

Forward Checking Algorithm

- For a single constraint C:

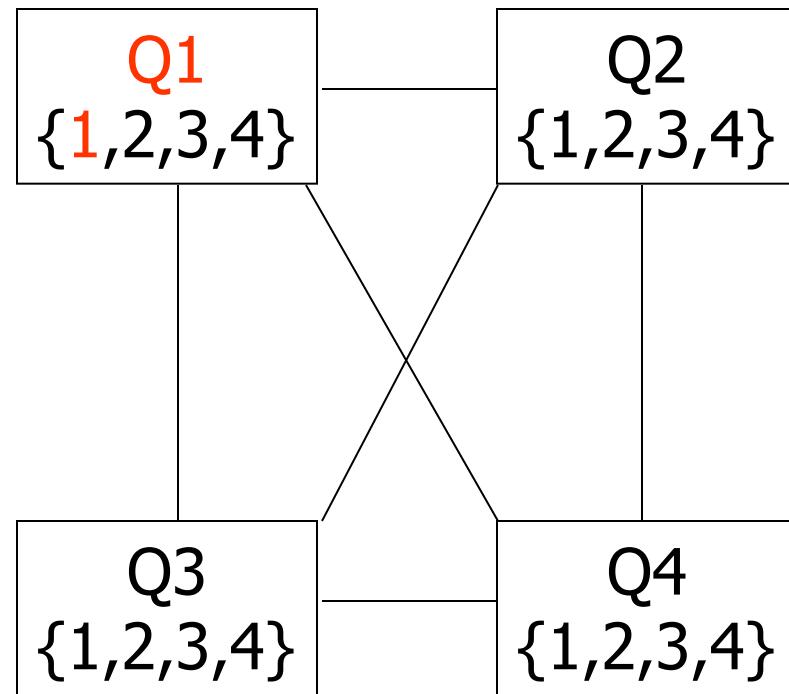
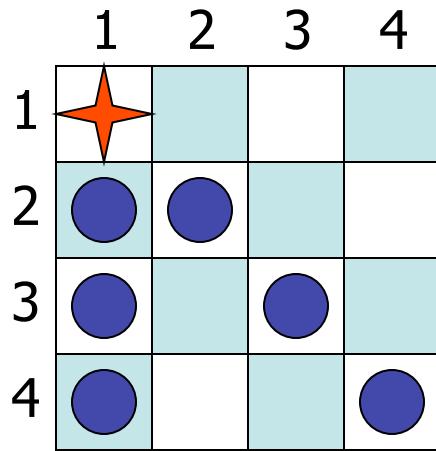
```
FCCheck(C, x)
  // C is a constraint with all its variables already
  // assigned, except for variable x.
  for d := each member of CurDom[x]
    IF making x = d together with previous assignments
      to variables in scope C falsifies C
      THEN remove d from CurDom[x]
    IF CurDom[x] = {} then return DWO (Domain Wipe Out)
  ELSE return ok
```

Forward Checking Algorithm

```
FC(Level) /*Forward Checking Algorithm */
    If all variables are assigned
        PRINT Value of each Variable
        RETURN or EXIT (RETURN for more solutions)
                            (EXIT for only one solution)
    V := PickAnUnassignedVariable()
    Assigned[V] := TRUE
    for d := each member of CurDom(V)
        Value[V] := d
        DWOoccurred:= False
        for each constraint C over V such that
            a) C has only one unassigned variable X in its scope
            if(FCCheck(C,X) == DWO)      /* X domain becomes empty*/
                DWOoccurred:= True
                break /* stop checking constraints */
        if(not DWOoccurred) /*all constraints were ok*/
            FC(Level+1)
            RestoreAllValuesPrunedByFCCheck()
    Assigned[V] := FALSE //undo since we have tried all of V's values
    return;
```

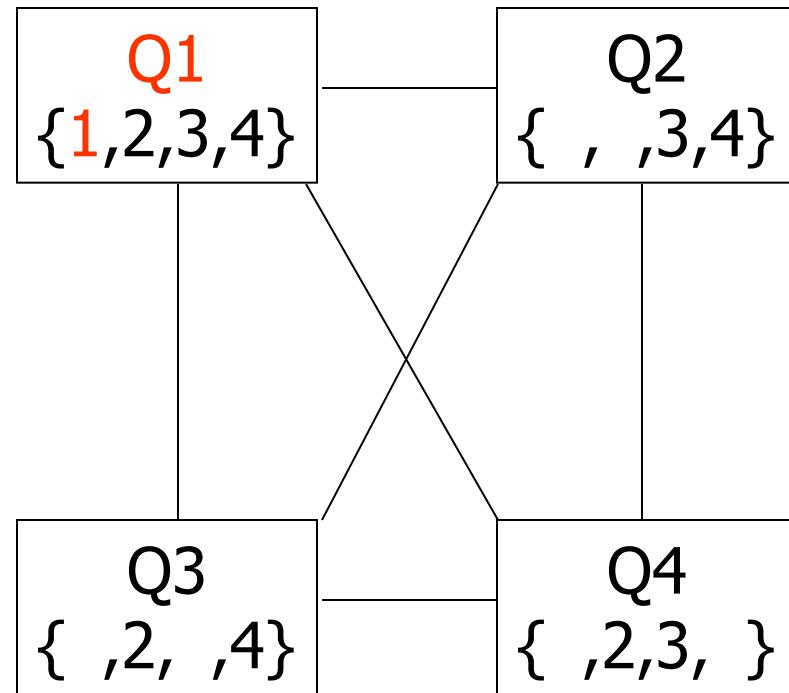
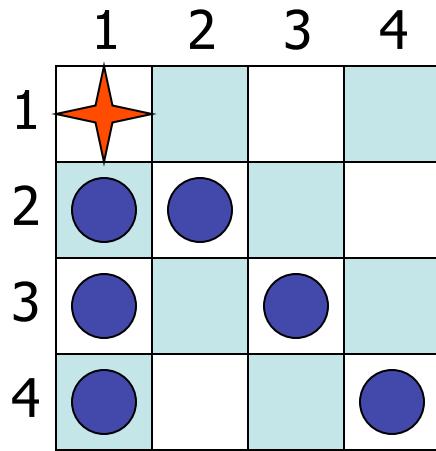
4-Queens Problem

- Encoding with Q_1, \dots, Q_4 denoting a queen per row
 - cannot put two queens in same column

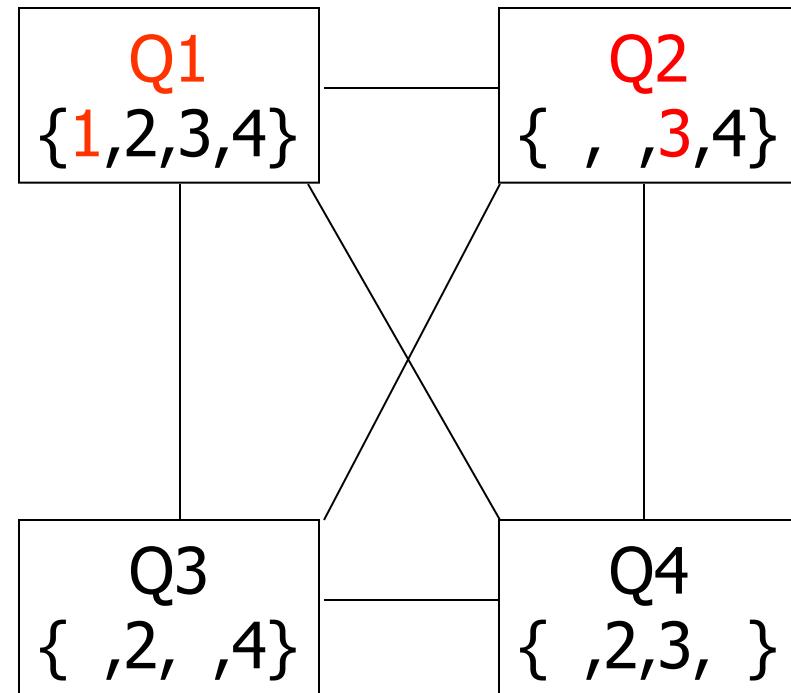
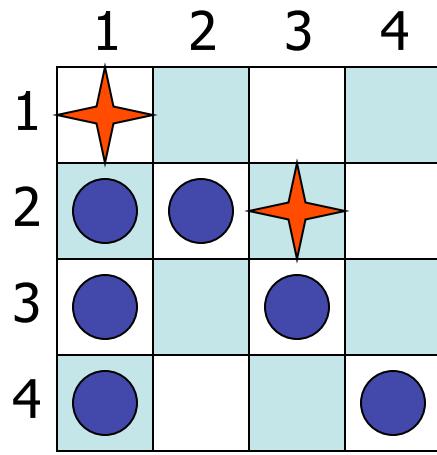


4-Queens Problem

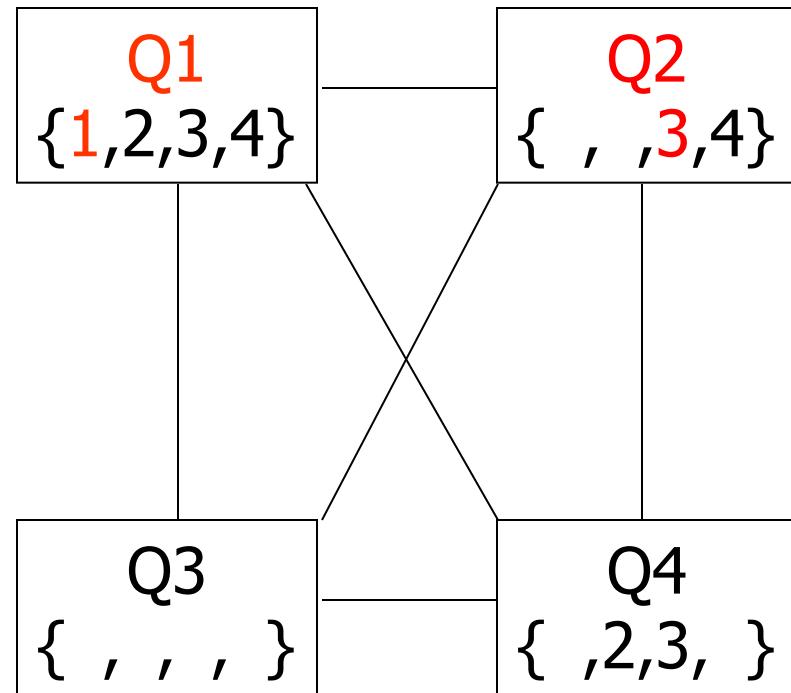
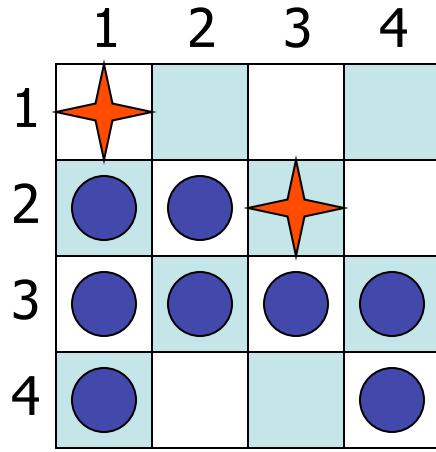
- Forward checking reduced the domains of all variables that are involved in a constraint with one uninstantiated variable:
 - Here all of Q2, Q3, Q4



4-Queens Problem

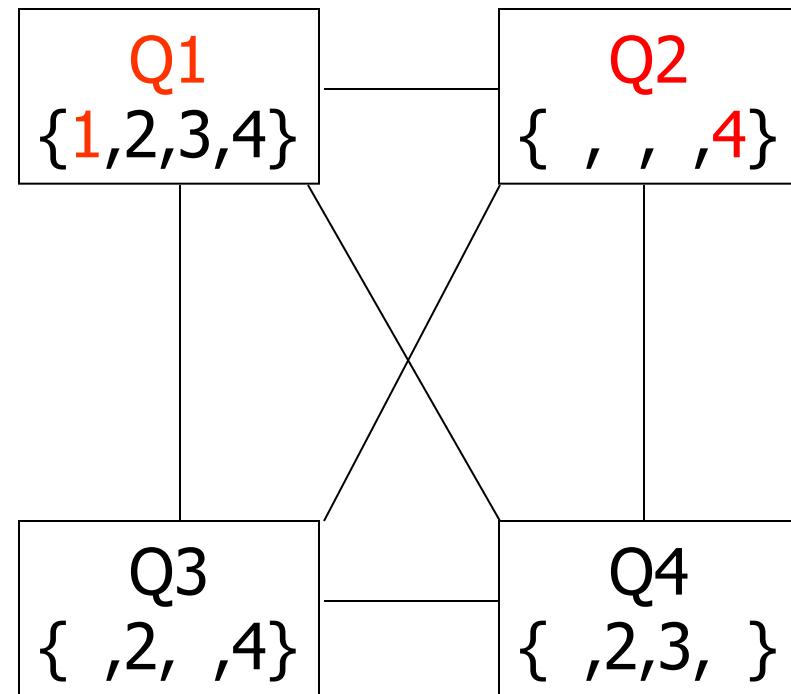
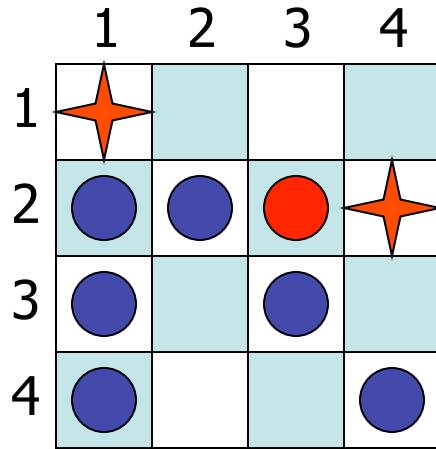


4-Queens Problem

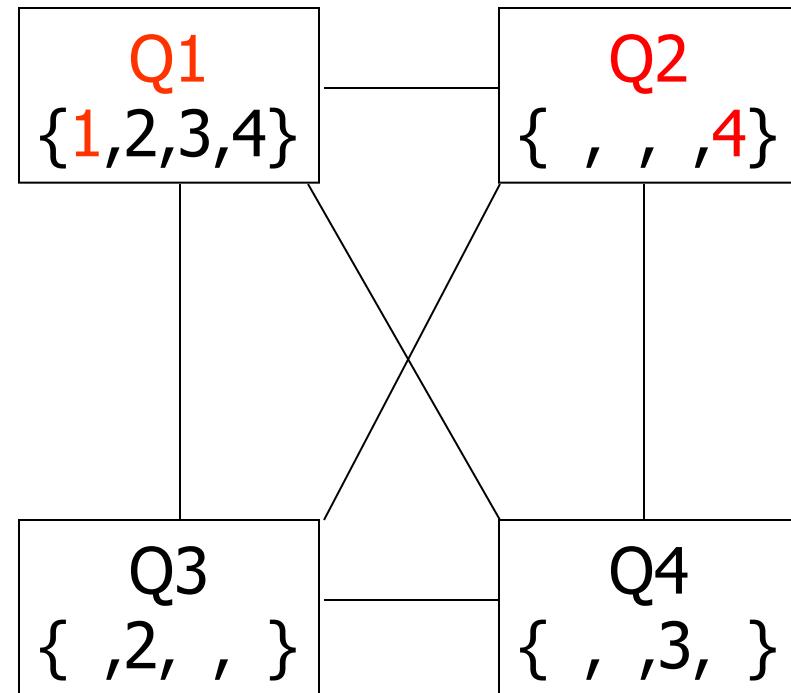
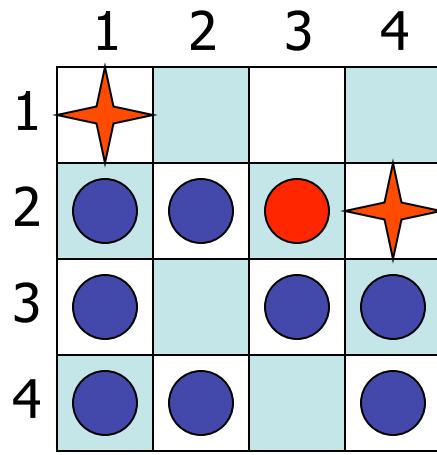


DWO

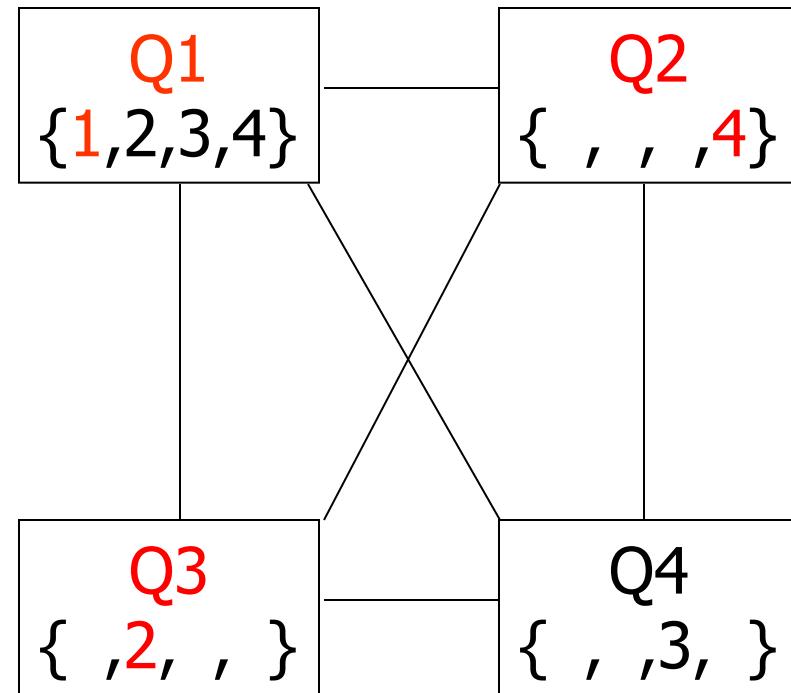
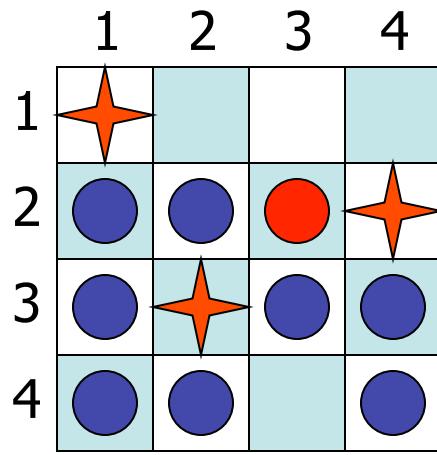
4-Queens Problem



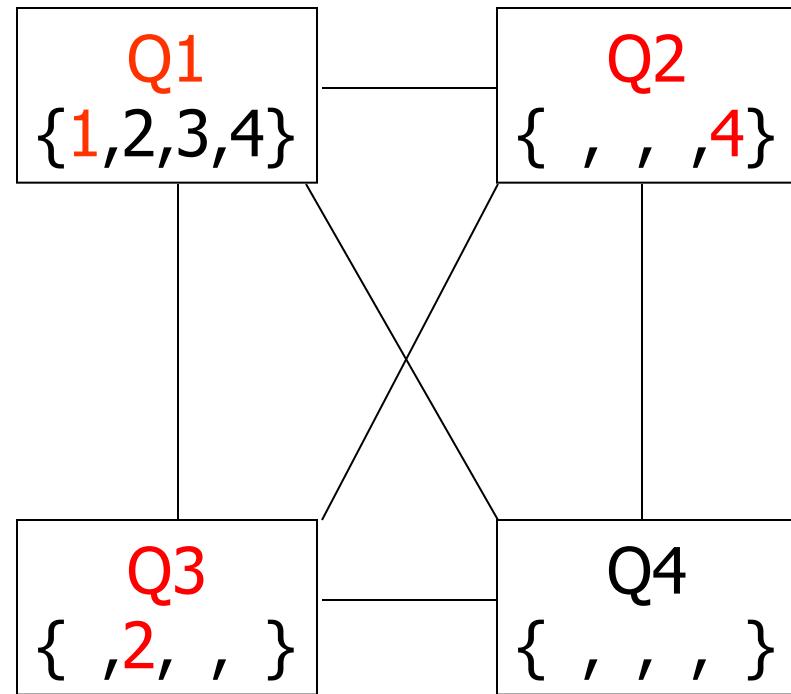
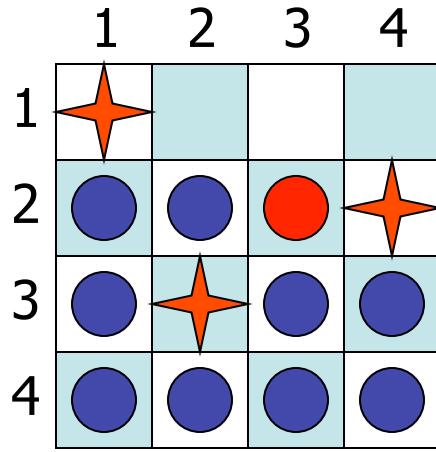
4-Queens Problem



4-Queens Problem



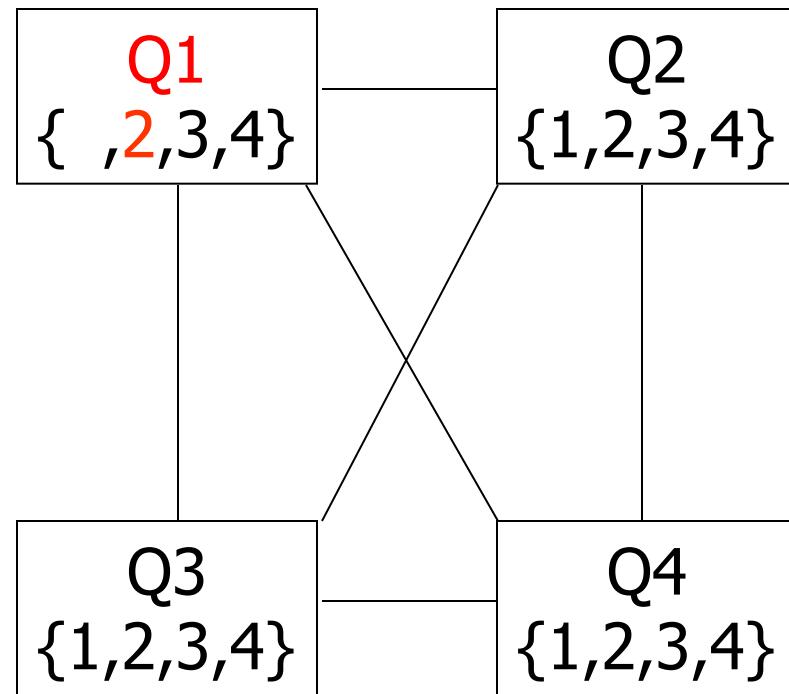
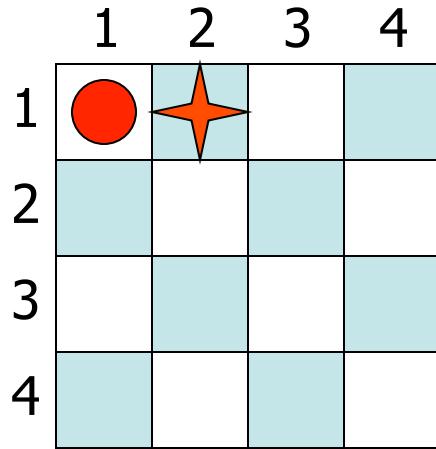
4-Queens Problem



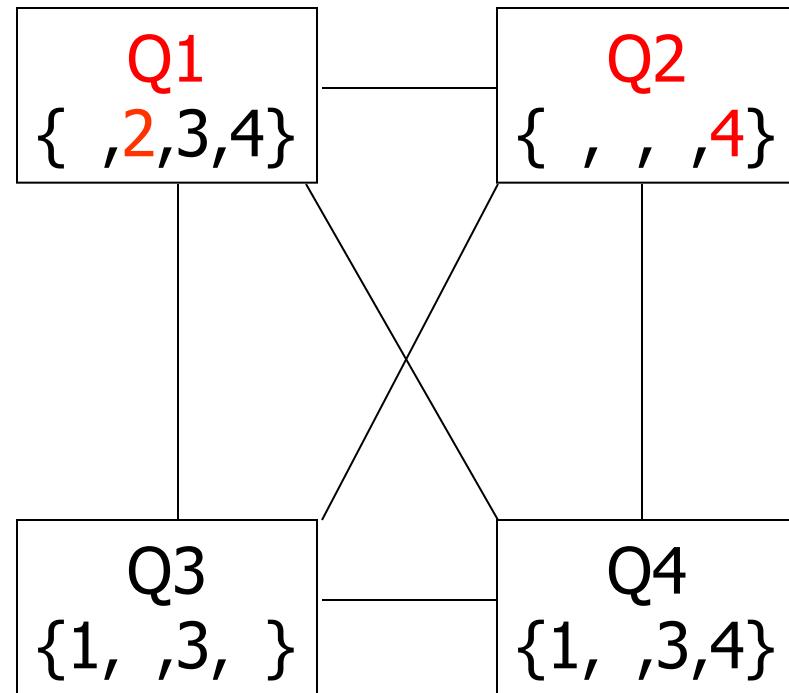
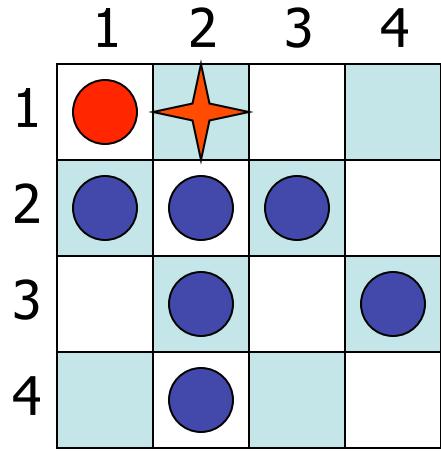
DWO

4-Queens Problem

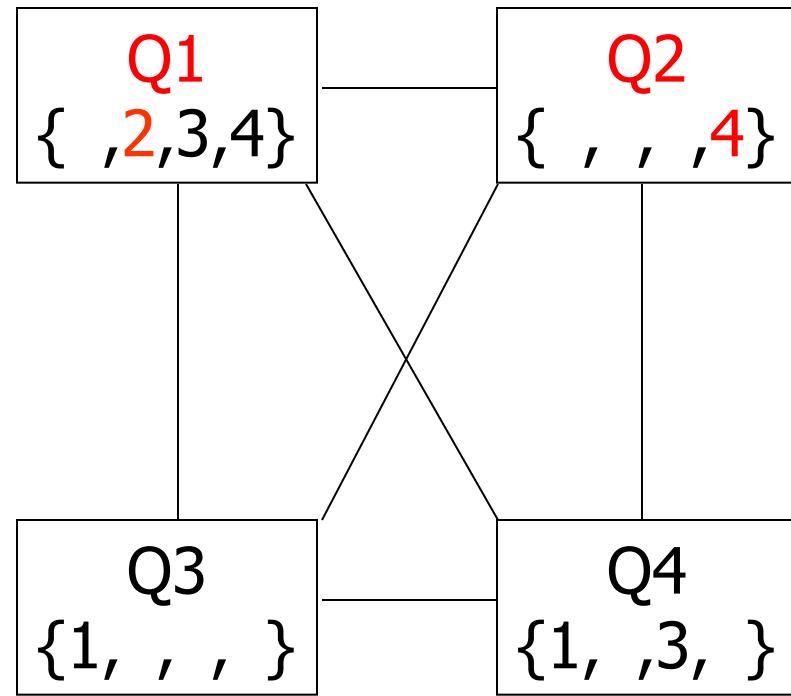
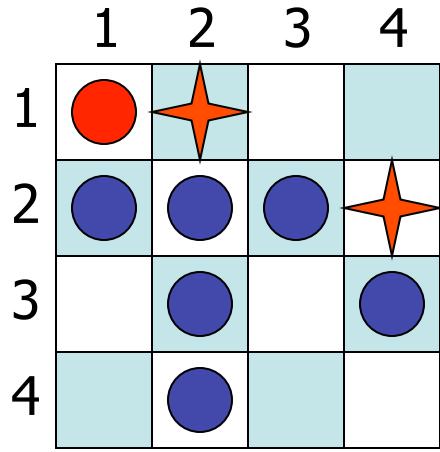
- Exhausted the subtree with $Q_1=1$; try now $Q_1=2$



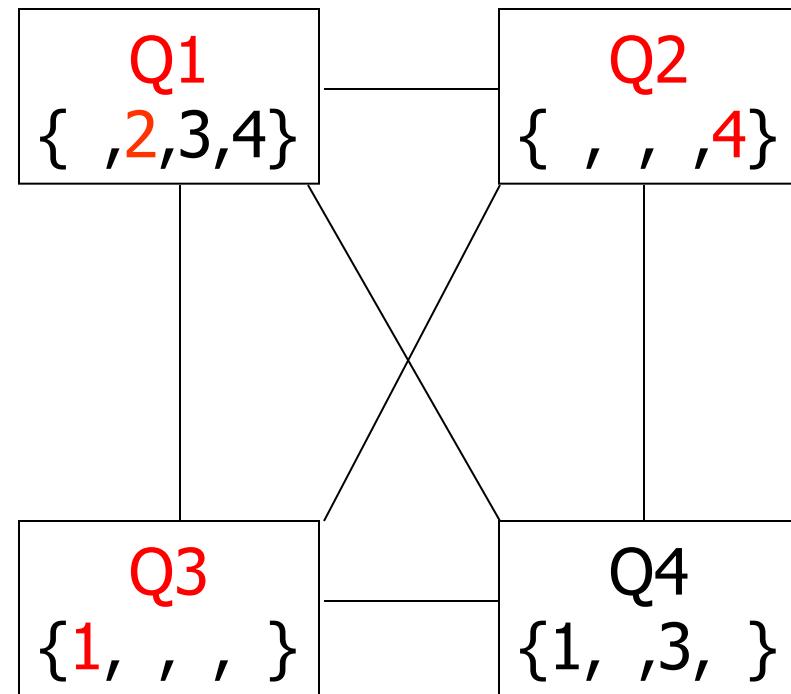
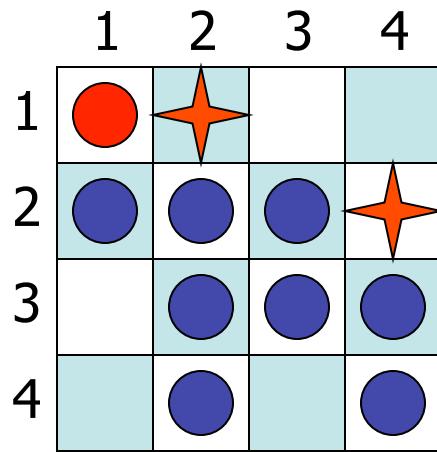
4-Queens Problem



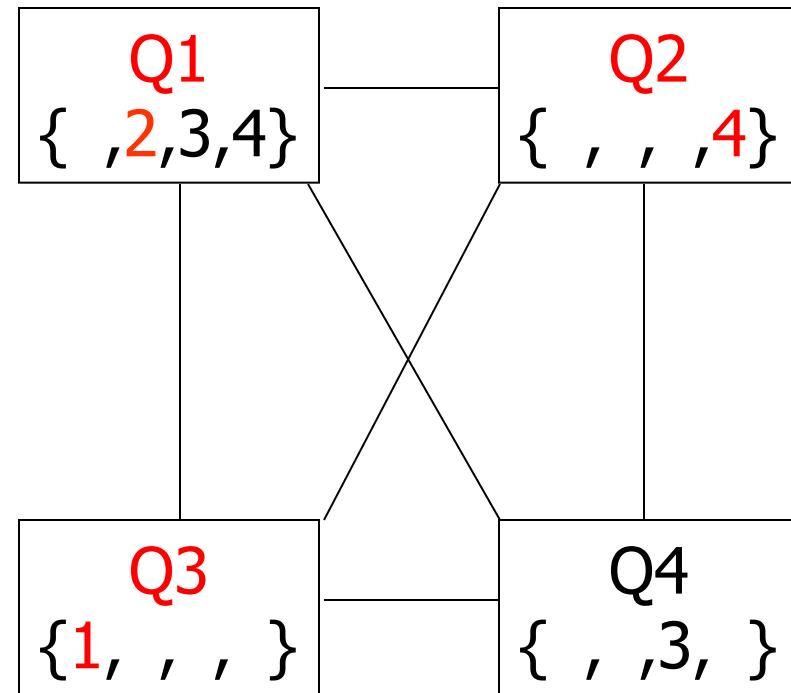
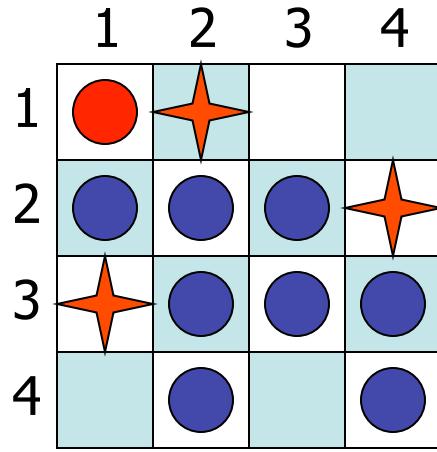
4-Queens Problem



4-Queens Problem

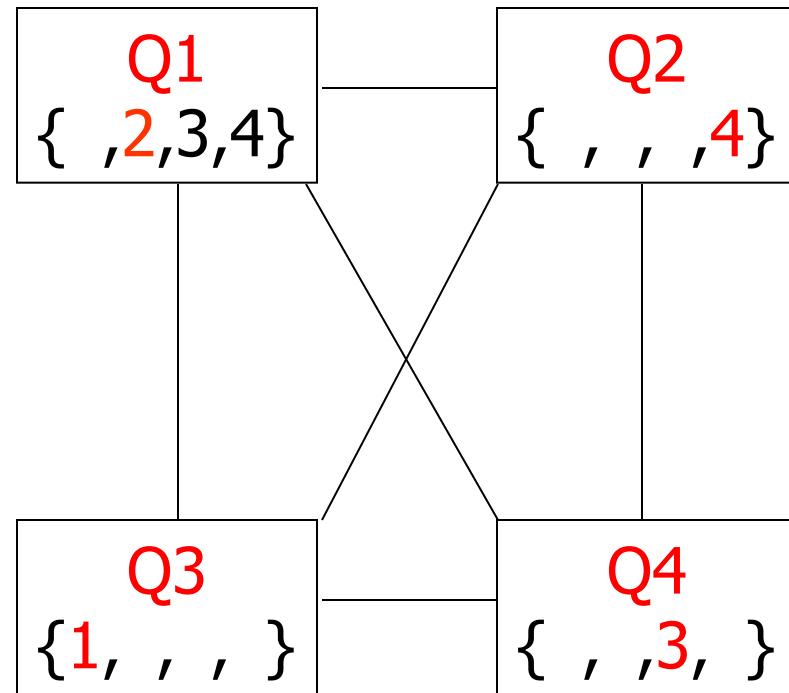
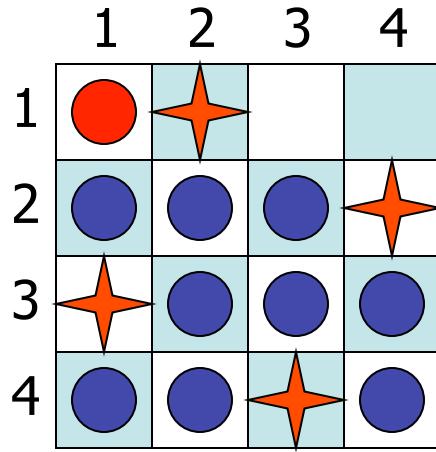


4-Queens Problem

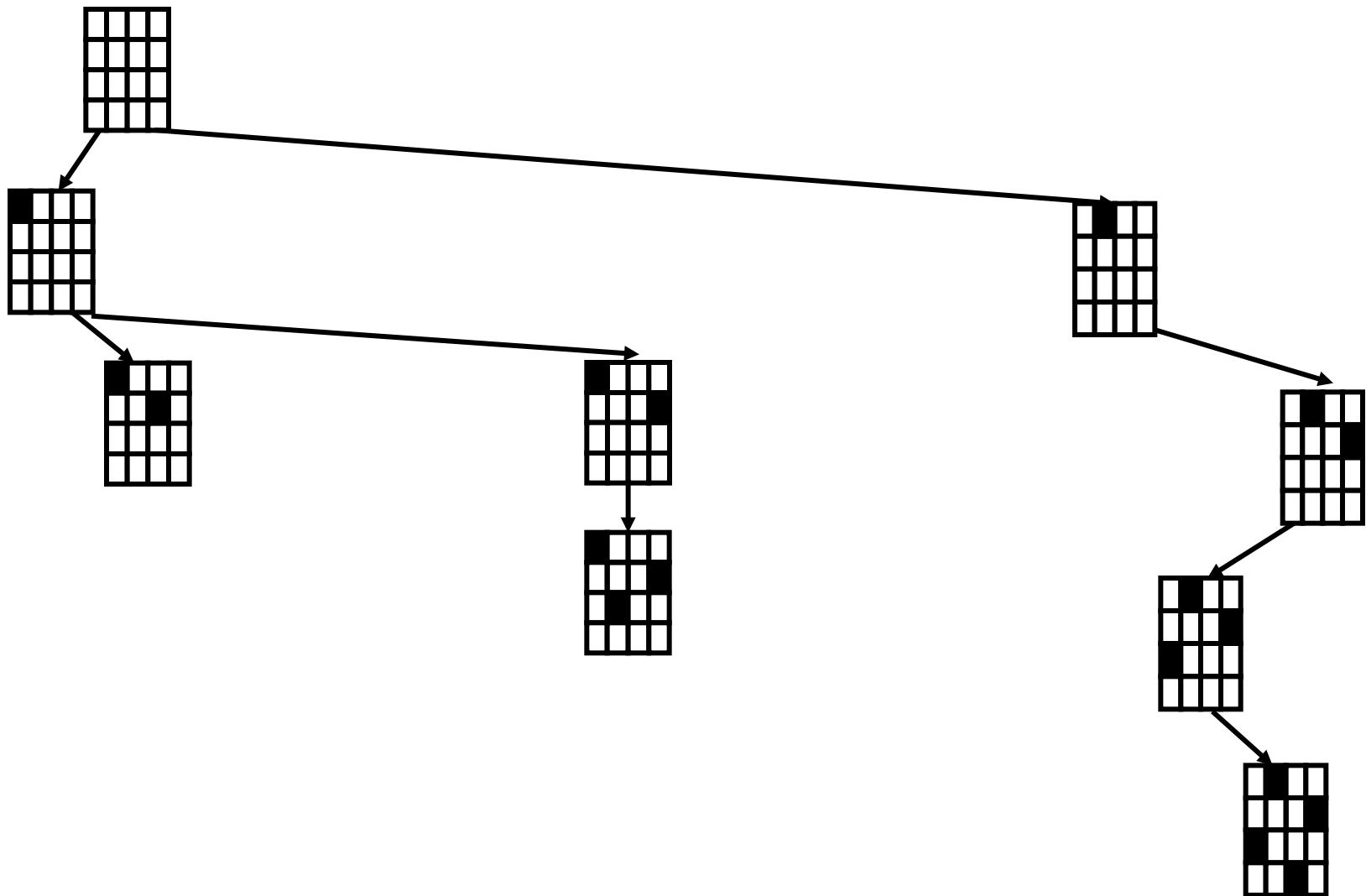


4-Queens Problem

- We have now find a solution: an assignment of all variables to values of their domain so that all constraints are satisfied



Example: N-Queens FC search Space



FC: Restoring Values

- After we backtrack from the current assignment (in the for loop) we must restore the values that were pruned as a result of that assignment.
- Some bookkeeping needs to be done, as we must remember which values were pruned by which assignment (FCCheck is called at every recursive invocation of FC).

FC: Minimum Remaining Values Heuristics (MRV)

- FC also gives us for free a very powerful heuristic to guide us which variables to try next:
 - Always branch on a variable with the smallest remaining values (smallest CurDom).
 - If a variable has only one value left, that value is forced, so we should propagate its consequences immediately.
 - This heuristic tends to produce skinny trees at the top. This means that more variables can be instantiated with fewer nodes searched, and thus more constraint propagation/DWO failures occur when the tree starts to branch out (we start selecting variables with larger domains)
 - We can find a inconsistency much faster

MRV Heuristic: Human Analogy

- What variables would you try first?

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

Domain of each variable:
 $\{1, \dots, 9\}$

(1, 5): impossible values:
Row: $\{1, 4, 5, 6, 8\}$
Column: $\{1, 3, 4, 5, 7, 9\}$
Subsquare: $\{5, 7, 9\}$
→ Domain = $\{2\}$

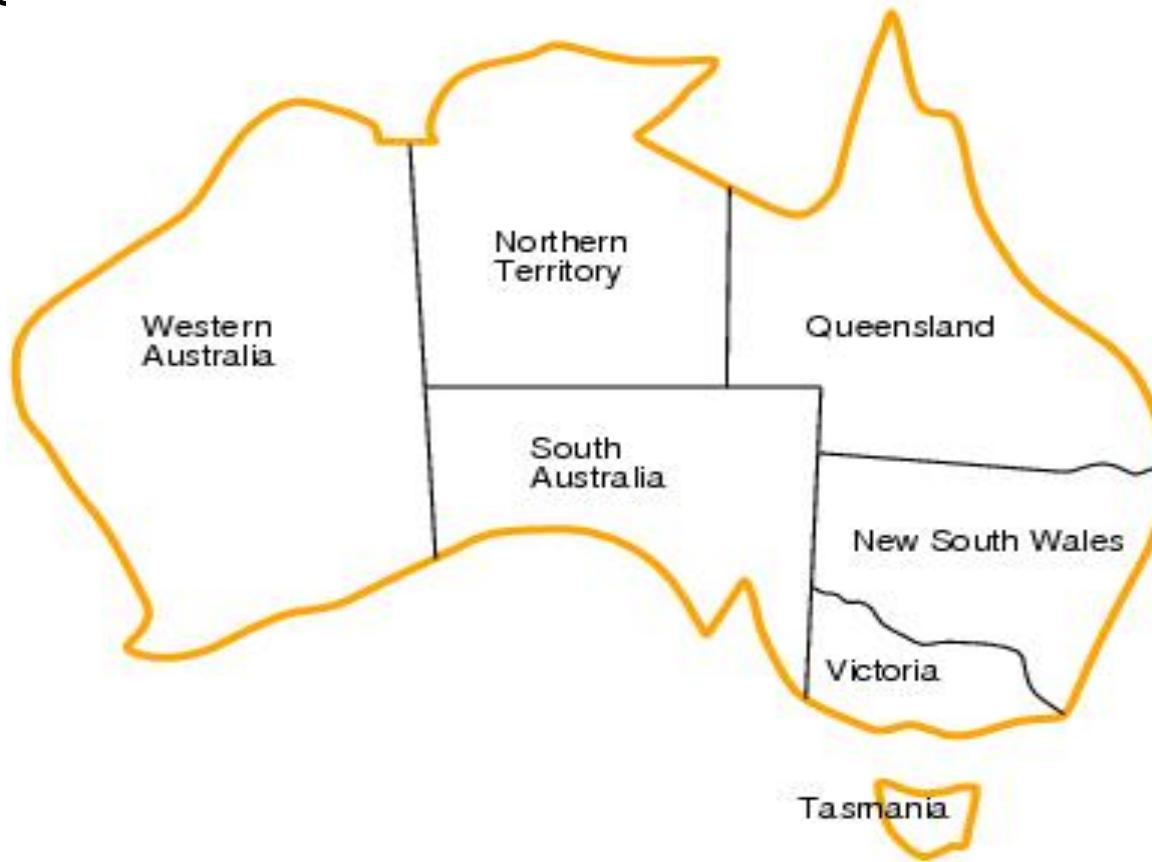
(9, 5): impossible values:
Row: $\{1, 5, 7, 8, 9\}$
Column: $\{1, 3, 4, 5, 7, 9\}$
Subsquare: $\{1, 5, 7, 9\}$
→ Domain = $\{2, 6\}$

After assigning value 2 to
cell (1,5): Domain = {6}

Most restricted variables! = MRV

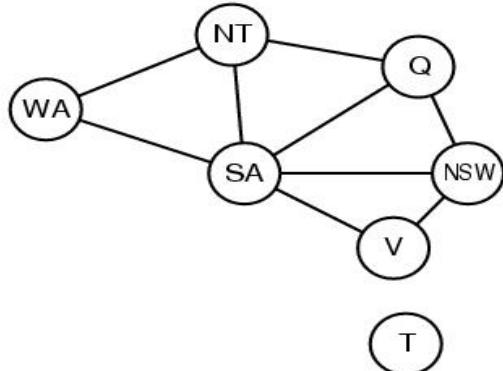
Example – Map Colouring

- Color the following map using red, green, and blue such that adjacent regions have different colors.



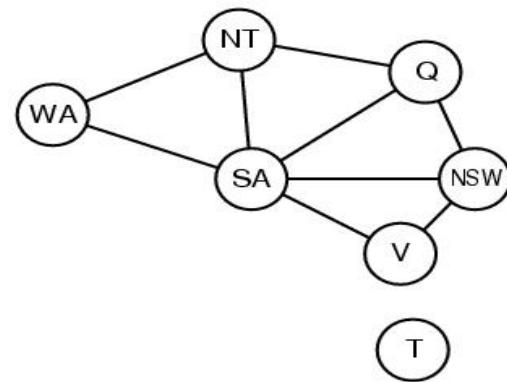
Example – Map Colouring

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



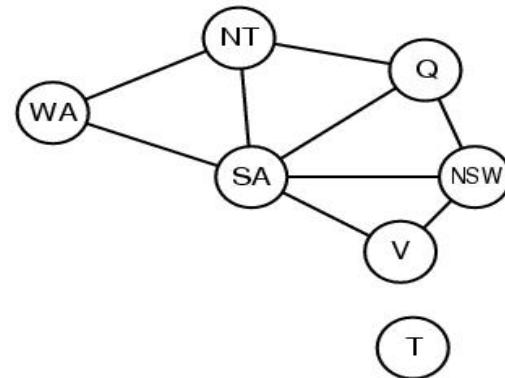
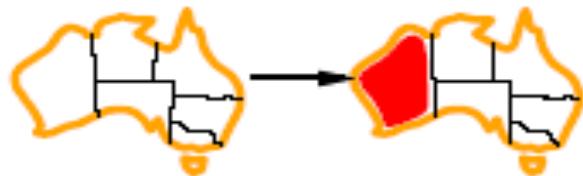
Example – Map Colouring

- *Forward checking idea*: keep track of remaining legal values for unassigned variables.
- Terminate search when any variable has no legal values.



Example – Map Colouring

- Assign {WA=red}
- Effects on other variables connected by constraints to WA
 - NT can no longer be red
 - SA can no longer be red



WA

NT

Q

NSW

V

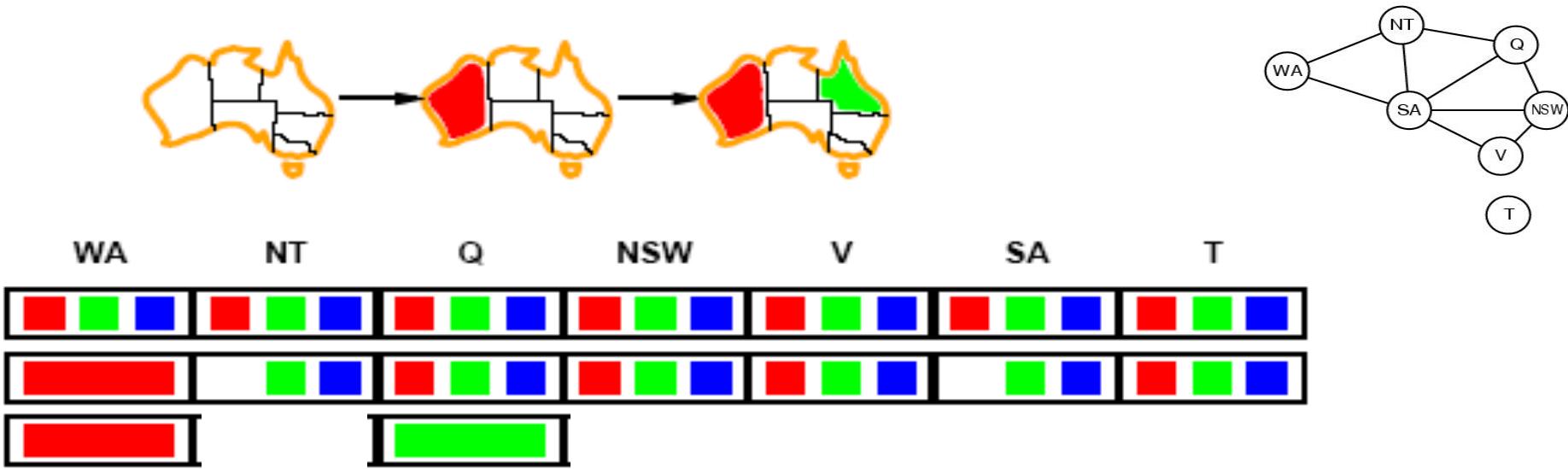
SA

T



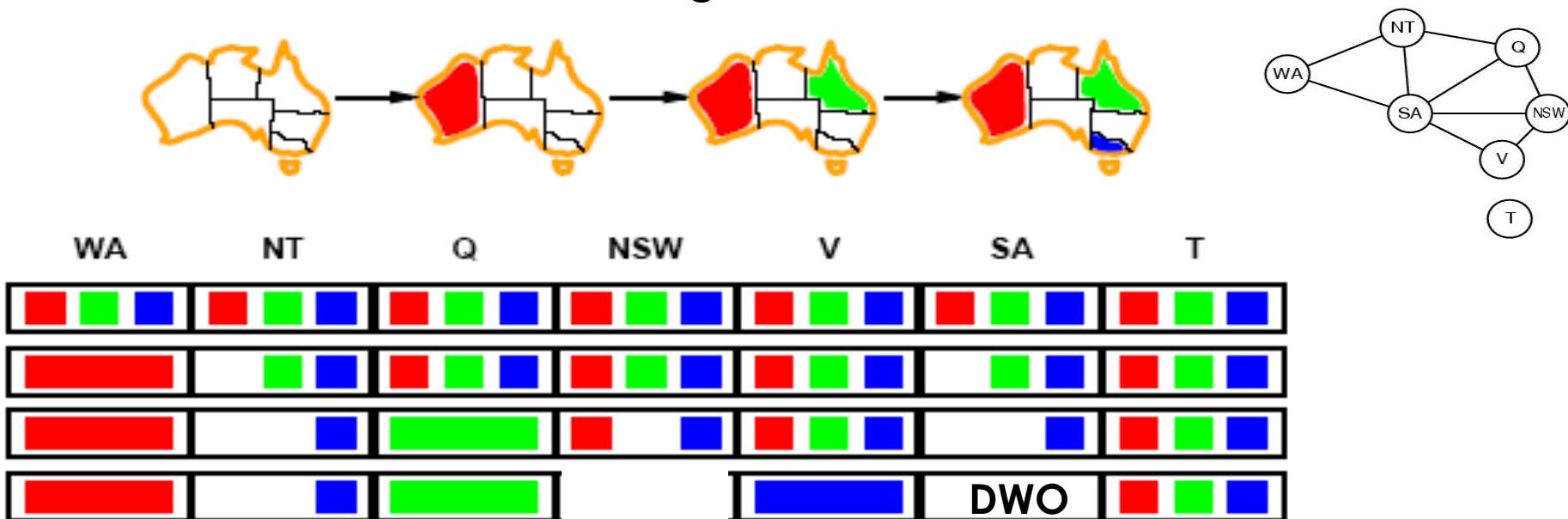
Example – Map Colouring

- Assign $\{Q=\text{green}\}$ (Note: Not using MRV)
- Effects on other variables connected by constraints with Q
 - NT can no longer be green
 - NSW can no longer be green
 - SA can no longer be green
- MRV heuristic would automatically select NT or SA next



Example – Map Colouring

- Assign $\{V=\text{blue}\}$ (not using MRV)
- Effects on other variables connected by constraints with V
 - NSW can no longer be blue
 - SA is empty
- FC has detected that partial assignment is *inconsistent* with the constraints and backtracking can occur.



Empirically

- FC often is about 100 times faster than BT
- FC with MRV (minimal remaining values) often 10000 times faster.
- But on some problems the speed up can be much greater
 - Converts problems that are not solvable to problems that are solvable.
- Still FC is not that powerful. Other more powerful forms of constraint propagation are used in practice.
- Try the previous map coloring example with MRV.

Constraint Propagation: Generalized Arc Consistency

GAC—Generalized Arc Consistency

1. $C(V_1, V_2, V_3, \dots, V_n)$ is GAC with respect to variable V_i , if and only if

For **every** value of V_i , there exists values of $V_1, V_2, V_{i-1}, V_{i+1}, \dots, V_n$ that satisfy C .

Note that the values are removed from the variable domains during search. So variables that are GAC in a constraint might become inconsistent (non-GAC).

Constraint Propagation: Generalized Arc Consistency

- $C(V_1, V_2, \dots, V_n)$ is GAC if and only if

It is GAC with respect to every variable in its scope.

- A CSP is GAC if and only if

all of its constraints are GAC.

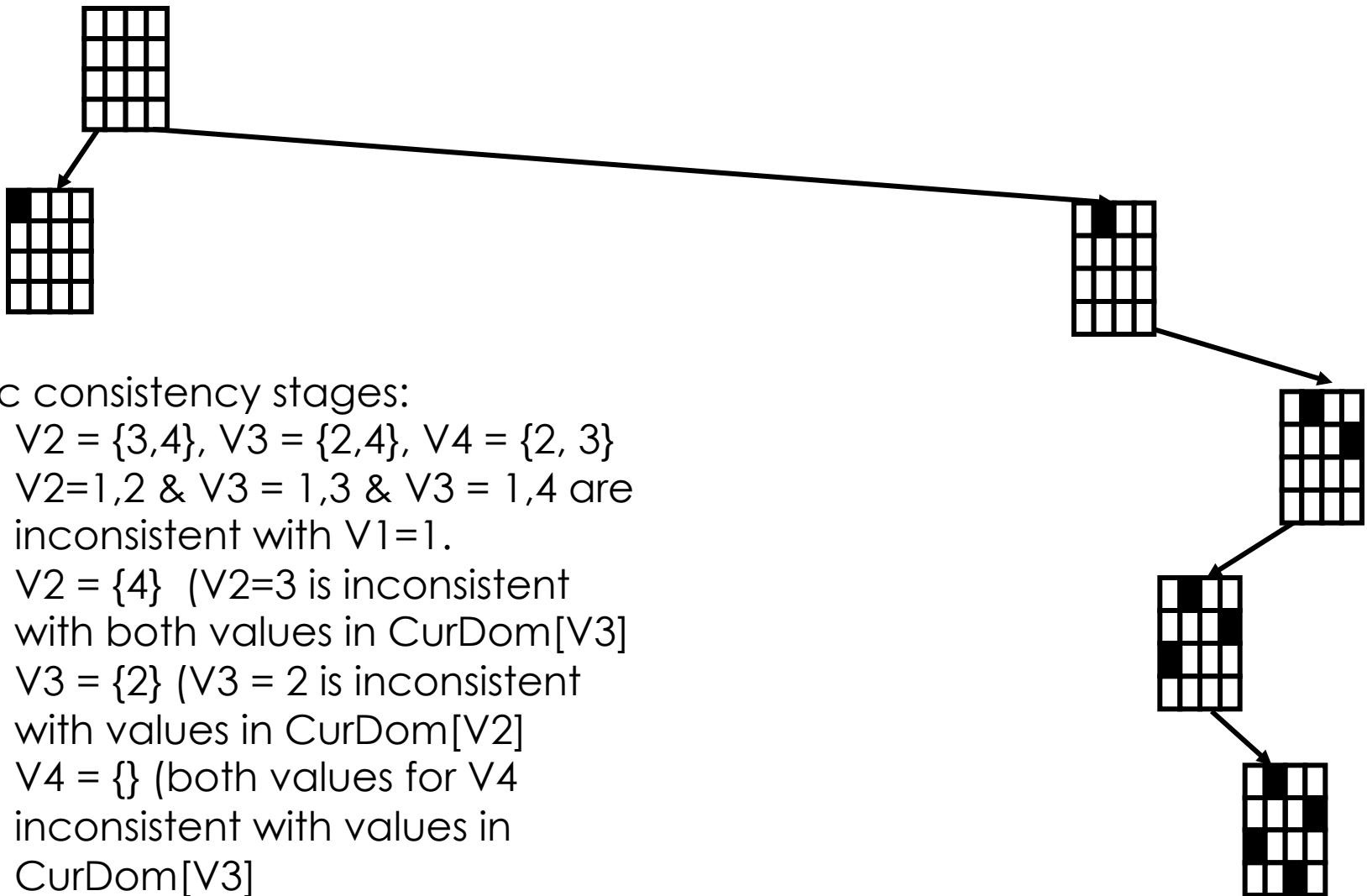
Constraint Propagation: Arc Consistency

- Say we find a value d of variable V_i that is not consistent: That is, there is no assignments to the other variables that satisfy the constraint when $V_i = d$
 - d is said to be **Arc Inconsistent**
 - We **can remove** d from the domain of V_i —this value cannot lead to a solution (much like Forward Checking, but more powerful).
- e.g. $C(X,Y): X > Y$ $\text{Dom}(X)=\{1,5,11\}$ $\text{Dom}(Y)=\{3,8,15\}$
 - For $X=1$ there is no value of Y s.t. $1 > Y \Rightarrow$ so we can remove 1 from domain X
 - For $Y=15$ there is no value of X s.t. $X > 15$, so remove 15 from domain Y
 - We obtain more restricted domains $\text{Dom}(X)=\{5,11\}$ and $\text{Dom}(Y)=\{3,8\}$

Constraint Propagation: Arc Consistency

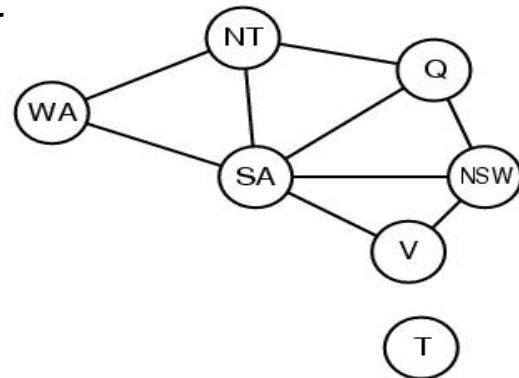
- If we apply arc consistency propagation during search the search tree's size will typically be much reduced in size.
- Removing a value from a variable domain may trigger further inconsistency, so we have to repeat the procedure until everything is consistent.
 - We put constraints on a queue and add new constraints to the queue as we need to check for arc consistency.

Example: N-Queens GAC search Space



Example – Map Colouring

- Assign $\{WA=\text{red}\}$
- Effects on other variables connected by constraints to WA
 - NT can no longer be red = $\{G, B\}$
 - SA can no longer be red = $\{G, B\}$
- All other values are arc-consistent



Example – Map Colouring

- Assign $\{Q=\text{green}\}$
- Effects on other variables connected by constraints with Q
 - NT can no longer be green = $\{B\}$
 - NSW can no longer be green = $\{R, B\}$
 - SA can no longer be green = $\{B\}$
- DWO there is no value for SA that will be consistent with $NT \neq SA$ and $NT = B$

Note Forward Checking would not have detected this DWO.

GAC Algorithm

- We make all constraints GAC at every node of the search space.
- This is accomplished by removing from the domains of the variables all arc inconsistent values.

GAC Algorithm, enforce GAC during search

```
GAC(Level) /*Maintain GAC Algorithm */
If all variables are assigned
    PRINT Value of each Variable
    RETURN or EXIT (RETURN for more solutions)
                    (EXIT for only one solution)
V := PickAnUnassignedVariable()
Assigned[V] := TRUE
for d := each member of CurDom(V)
    Value[V] := d
    Prune all values of V ≠ d from CurDom[V]
    for each constraint C whose scope contains V
        Put C on GACQueue
    if(GAC_Enforce() != DWO)
        GAC(Level+1) /*all constraints were ok*/
        RestoreAllValuesPrunedFromCurDoms()
Assigned[V] := FALSE
return;
```

Enforce GAC (prune all GAC inconsistent values)

GAC_Enforce()

```
// GAC-Queue contains all constraints one of whose variables has
// had its domain reduced. At the root of the search tree
// first we run GAC_Enforce with all constraints on GAC-Queue

while GACQueue not empty
    C = GACQueue.extract()
    for V := each member of scope(C)
        for d := CurDom[V]
            Find an assignment A for all other
            variables in scope(C) such that
            C(A ∪ V=d) = True
            if A not found
                CurDom[V] = CurDom[V] - d
                if CurDom[V] = ∅
                    empty GACQueue
                    return DWO //return immediately
                else
                    push all constraints C' such that
                    V ∈ scope(C') and C' ∉ GACQueue
                    on to GACQueue
    return TRUE //while loop exited without DWO
```

Enforce GAC

- A **support** for $V=d$ in constraint C is an assignment \mathbf{A} to all of the other variables in $\text{scope}(C)$ such that $\mathbf{A} \cup \{V=d\}$ satisfies C . (\mathbf{A} is what the algorithm's inner loop looks for).
- Smarter implementations keep track of “supports” to avoid having to search through all possible assignments to the other variables for a satisfying assignment.

Enforce GAC

- Rather than search for a satisfying assignment to C containing $V=d$, we check to see if the current support is still valid: i.e., all values it assigns still lie in the variable's current domains
- Also we take advantage that a support for $V=d$, e.g. $\{V=d, X=a, Y=b, Z=c\}$ is also a support for $X=a$, $Y=b$, and $Z=c$

Enforce GAC

- However, finding a support for $V=d$ in constraint C still in the worst case requires $O(2^k)$ work, where k is the **arity** of C , i.e., $|\text{scope}(C)|$.
- Another key development in practice is that for some constraints this computation can be done in polynomial time.
E.g., $\text{all-diff}(V_1, \dots, V_n)$ we can check if $V_i=d$ has a support in the current domains of the other variables in polynomial time using ideas from graph theory.
We do not need to examine all combinations of values for the other variables looking for a support

GAC enforce example

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



= All-diff

$$C_{SS2} = \text{All-diff}(V_{1,4}, V_{1,5}, V_{1,6}, V_{2,4}, V_{2,5}, V_{2,6}, V_{3,4}, V_{3,5}, V_{3,6})$$

$$C_{R1} = \text{All-diff}(V_{1,1}, V_{1,2}, V_{1,3}, V_{1,4}, V_{1,5}, V_{1,6}, V_{1,7}, V_{1,8}, V_{1,9})$$

$$C_{C5} = \text{All-diff}(V_{1,5}, V_{2,5}, V_{35}, V_{4,5}, V_{5,5}, V_{6,5}, V_{7,5}, V_{8,5}, V_{9,5})$$

By going back and forth
between constraints we get
more values pruned.

$$\text{GAC}(C_{SS8}) \rightarrow \text{CurDom of } V_{1,5}, V_{1,6}, V_{2,4}, V_{3,4}, V_{3,6} = \{1, 2, 3, 4, 8\}$$

$$\text{GAC}(C_{R1}) \rightarrow \text{CurDom of } V_{1,7}, V_{1,8} = \{2, 3, 7, 9\}$$

$$\text{CurDom of } V_{1,5}, V_{1,6} = \{2, 3\}$$

$$\text{GAC}(C_{SS8}) \rightarrow \text{CurDom of } V_{2,4}, V_{3,4}, V_{3,6} = \{1, 4, 8\}$$

$$\text{GAC}(C_{C5}) \rightarrow \text{CurDom of } V_{5,5}, V_{9,5} = \{2, 6, 8\}$$

$$\rightarrow \text{CurDom of } V_{1,5} = \{2\}$$

$$\text{GAC}(C_{SS8}) \rightarrow \text{CurDom of } V_{1,6} = \{3\}$$

Many real-world applications of CSP

- Assignment problems
 - who teaches what class
- Timetabling problems
 - exam schedule
- Transportation scheduling
- Floor planning
- Factory scheduling
- Hardware configuration
 - a set of compatible components

Constraint Satisfaction Problems (Backtracking Search)

- Chapter 6
 - 6.1: Formalism
 - 6.2: Constraint Propagation
 - 6.3: Backtracking Search for CSP
 - 6.4 is about local search which is a very useful idea but we won't cover it in class.

Representing States with Feature Vectors

- For each problem we have designed a new state representation (and designed the sub-routines called by search based on this representation).
- **Feature vectors** provide a general state representation that is useful for many different problems.
- Feature vectors are also used in many other areas of AI, particularly Machine Learning, Reasoning under Uncertainty, Computer Vision, etc.

Feature Vectors

- We have
 - A set of k variables (or features)
 - Each variable has a **domain** of different values.
 - A state is specified by an assignment of a value for each variable.
 - height = {short, average, tall},
 - weight = {light, average, heavy}
 - A partial state is specified by an assignment of a value to **some** of the variables.

Example: Sudoku

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

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

Example: Sudoku

- 81 **variables**, each representing the value of a cell.
- **Domain of Values**: a single value for those cells that are already filled in, the set {1, ...9} for those cells that are empty.
- **State**: any completed board given by specifying the value in each cell (1-9, or blank).
- **Partial State**: some incomplete filling out of the board.

Fahiem Bacchus, CSC384 Introduction to Artificial Intelligence, University of Toronto

5

Example: 8-Puzzle

2	3	7
6	4	8
5	1	

- **Variables**: 9 variables $\text{Cell}_{1,1}$, $\text{Cell}_{1,2}$, ..., $\text{Cell}_{3,3}$
- **Values**: {'B', 1, 2, ..., 8}
- **State**: Each "Cell_{i,j}" variable specifies what is in that position of the tile.
 - If we specify a value for each cell we have completely specified a state.

This is only one of many ways to specify the state.

Fahiem Bacchus, CSC384 Introduction to Artificial Intelligence, University of Toronto

6

Constraint Satisfaction Problems

- Notice that in these problems some settings of the variables are **illegal**.
 - In Sudoku, we can't have the same number in any column, row, or subsquare.
 - In the 8 puzzle each variable must have a distinct value (same tile can't be in two places)

Fahiem Bacchus, CSC384 Introduction to Artificial Intelligence, University of Toronto

7

Constraint Satisfaction Problems

- In many practical problems finding which setting of the feature variables yields a legal state is difficult.

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

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

- We want to find a state (setting of the variables) that satisfies certain constraints.

Fahiem Bacchus, CSC384 Introduction to Artificial Intelligence, University of Toronto

8

Constraint Satisfaction Problems

- In Sudoku: The variables that form
 - a column must be distinct
 - a row must be distinct
 - a sub-square must be distinct.

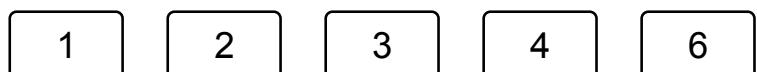
2		6						3
7	4		8					
			3		2			
8			4		1			
6		5						
			1	7	8			
5			9			4		

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

Example Car Sequencing

Car Factory Assembly Line—back to the days of Henry Ford

Move the items to be assembled don't move the workers

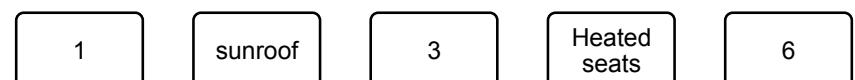


The assembly line is divided into stations. A particular task is preformed at each station.

Constraint Satisfaction Problems

- In these problems we **do not care** about the sequence of moves needed to get to a goal state.
- We only care about finding a feature vector (a setting of the variables) that satisfies the goal.
 - A setting of the variables that satisfies some constraints.
- In contrast, in the 8-puzzle, the feature vector satisfying the goal is given. We care about the sequence of moves needed to move the tiles into that configuration

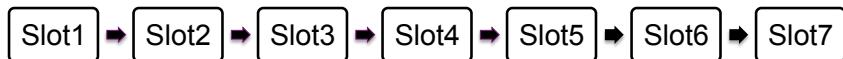
Example Car Sequencing



Some stations install optional items...not every car in the assembly line is worked on in that station.

As a result the factory is designed to have lower capacity in those stations.

Example Car Sequencing



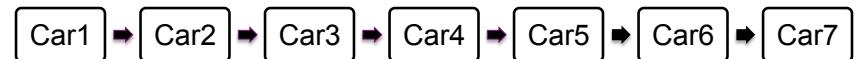
Cars move through the factory on an assembly line which is broken up into slots.

The stations might be able to process only a limited number of slots out of some group of slots that is passing through the station at any time.

E.g., the sunroof station might accommodate 4 slots, but only has capacity to process 2 slots out of the 4 at any one time.

Fahiem Bacchus, CSC384 Introduction to Artificial Intelligence, University of Toronto

Example Car Sequencing



Max 2

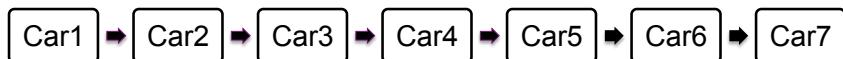
Max 2

Max 2

Max 2

Fahiem Bacchus, CSC384 Introduction to Artificial Intelligence, University of Toronto

Example Car Sequencing



Each car to be assembled has a list of required options. We want to assign each car to be assembled to a slot on the line.

But we want to ensure that no sequence of 4 slots has more than 2 cars assigned that require a sun roof.

Finding a feasible assignment of cars with different options to slots without violating the capacity constraints of the different stations is hard.

Fahiem Bacchus, CSC384 Introduction to Artificial Intelligence, University of Toronto

Formalization of a CSP

- A CSP consists of
 - A set of **variables** V_1, \dots, V_n
 - For each variable a (finite) **domain** of possible values $\text{Dom}[V_i]$.
 - A set of **constraints** C_1, \dots, C_m .
 - A **solution** to a CSP is an **assignment** of a value to all of the variables such that **every constraint is satisfied**.
 - A CSP is unsatisfiable if no solution exists.

Fahiem Bacchus, CSC384 Introduction to Artificial Intelligence, University of Toronto

Formalization of a CSP

- Each variable can be assigned any value from its domain.
 - $V_i = d$ where $d \in \text{Dom}[V_i]$
- Each constraint C
 - Has a set of variables it is over, called its **scope**
 - E.g., $C(V_1, V_2, V_4)$ is a constraint over the variables V_1 , V_2 , and V_4 . Its scope is $\{V_1, V_2, V_4\}$
 - Given an assignment to its variables the constraint returns:
 - True—this assignment satisfies the constraint
 - False—this assignment falsifies the constraint.

Formalization of a CSP

- We can specify the constraint with a table
- $C(V_1, V_2, V_4)$ with $\text{Dom}[V_1] = \{1, 2, 3\}$ and $\text{Dom}[V_2] = \text{Dom}[V_4] = \{1, 2\}$

V1	V2	V4	C(V1,V2,V4)
1	1	1	False
1	1	2	False
1	2	1	False
1	2	2	False
2	1	1	True
2	1	2	False
2	2	1	False
2	2	2	False
3	1	1	False
3	1	2	True
3	2	1	True
3	2	2	False

Formalization of a CSP

- Often we can specify the constraint more compactly with an expression:
 $C(V_1, V_2, V_4) = (V_1 = V_2 + V_4)$

V1	V2	V4	C(V1,V2,V4)
1	1	1	False
1	1	2	False
1	2	1	False
1	2	2	False
2	1	1	True
2	1	2	False
2	2	1	False
2	2	2	False
3	1	1	False
3	1	2	True
3	2	1	True
3	2	2	False

Formalization of a CSP

- **Unary** Constraints (over one variable)
 - e.g. $C(X): X=2$; $C(Y): Y>5$
- **Binary** Constraints (over two variables)
 - e.g. $C(X,Y): X+Y<6$
- **Higher-order** constraints: over 3 or more variables.

Example: Sudoku

- **Variables:** $V_{11}, V_{12}, \dots, V_{21}, V_{22}, \dots, V_{91}, \dots, V_{99}$
- **Domains:**
 - $\text{Dom}[V_{ij}] = \{1-9\}$ for empty cells
 - $\text{Dom}[V_{ij}] = \{k\}$ a fixed value k for filled cells.
- **Constraints:**
 - Row constraints:
 - All-Diff($V_{11}, V_{12}, V_{13}, \dots, V_{19}$)
 - All-Diff($V_{21}, V_{22}, V_{23}, \dots, V_{29}$)
 - ..., All-Diff($V_{91}, V_{92}, \dots, V_{99}$)
 - Column Constraints:
 - All-Diff($V_{11}, V_{21}, V_{31}, \dots, V_{91}$)
 - All-Diff($V_{12}, V_{22}, V_{32}, \dots, V_{92}$)
 - ..., All-Diff($V_{19}, V_{29}, \dots, V_{99}$)
 - Sub-Square Constraints:
 - All-Diff($V_{11}, V_{12}, V_{13}, V_{21}, V_{22}, V_{23}, V_{31}, V_{32}, V_{33}$)
 - All-Diff($V_{14}, V_{15}, V_{16}, \dots, V_{34}, V_{35}, V_{36}$)

Example: Sudoku

- Each of these constraints is over 9 variables, and they are all the same constraint:
 - Any assignment to these 9 variables such that each variable has a different value satisfies the constraint.
 - Any assignment where two or more variables have the same value falsifies the constraint.
- This is a special kind of constraint called an **ALL-DIFF** constraint.
 - ALL-Diff(V_1, \dots, V_n) could also be encoded as a set of binary not-equal constraints between all possible pairs of variables: $V_1 \neq V_2, V_1 \neq V_3, \dots, V_2 \neq V_1, \dots, V_n \neq V_1, \dots, V_n \neq V_{n-1}$

Example: Sudoku

- Thus Sudoku has 3×9 ALL-DIFF constraints, one over each set of variables in the same row, one over each set of variables in the same column, and one over each set of variables in the same sub-square.

Solving CSPs

- Because CSPs do not require finding a paths (to a goal), it is best solved by a specialized version of depth-first search.
- **Key intuitions:**
 - We can build up to a solution by searching through the space of partial assignments.
 - Order in which we assign the variables does not matter – eventually they all have to be assigned. **We can decide on a suitable value for one variable at a time!**
→ **This is the key idea of backtracking search.**
 - If we falsify a constraint during the process of building up a solution, we can immediately reject the current partial assignment:
 - All extensions of this partial assignment will falsify that constraint, and thus none can be solutions.

CSP as a Search Problem

A CSP could be viewed as a more traditional search problem

- **Initial state:** empty assignment
- **Successor function:** a value is assigned to any unassigned variable, which does not cause any constraint to return false.
- **Goal test:** the assignment is complete

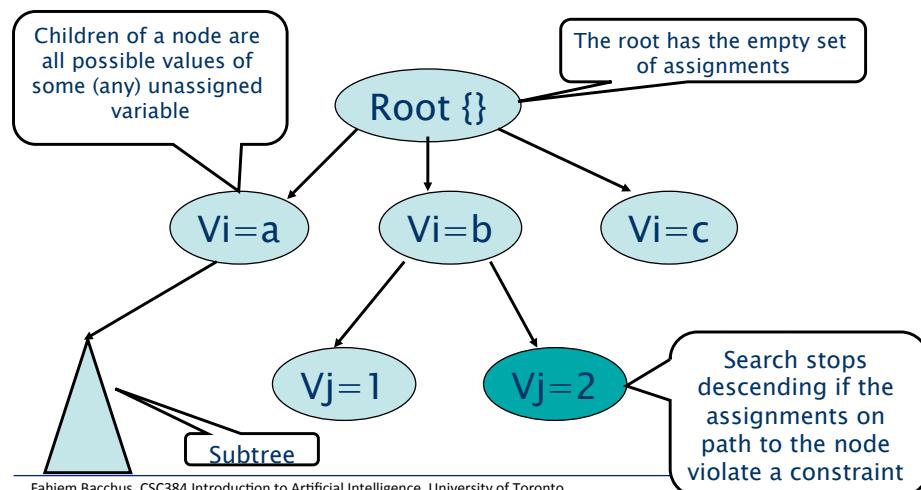
Backtracking Search: The Algorithm BT

```
BT(Level)
  If all variables assigned
    PRINT Value of each Variable
    RETURN or EXIT (RETURN for more solutions)
      (EXIT for only one solution)
  V := PickUnassignedVariable()
  Assigned[V] := TRUE
  for d := each member of Domain(V) (the domain values of V)
    Value[V] := d
    ConstraintsOK = TRUE
    for each constraint C such that
      a) V is a variable of C and
      b) all other variables of C are assigned:
        IF C is not satisfied by the set of current
          assignments:
          ConstraintsOK = FALSE
    If ConstraintsOk == TRUE:
      BT(Level+1)

  Assigned[V] := FALSE //UNDO as we have tried all of V's values
  return
```

Backtracking Search

- The algorithm searches a tree of partial assignments.

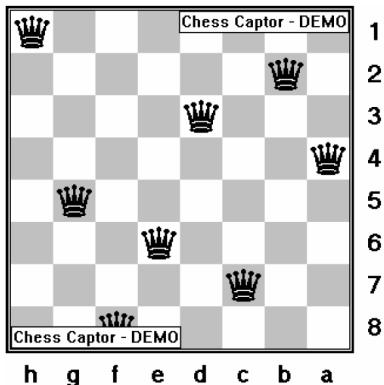


Backtracking Search

- Heuristics are used to determine
 - the order in which variables are assigned:
PickUnassignedVariable()
 - the order of values tried for each variable.
- The choice of the next variable can vary from branch to branch, e.g.,
 - under the assignment $V1=a$ we might choose to assign $V4$ next, while under $V1=b$ we might choose to assign $V5$ next.
- This “dynamically” chosen variable ordering has a tremendous impact on performance.

Example: N-Queens

- Place N Queens on an N X N chess board so that no Queen can attack any other Queen.



Fahiem Bacchus, CSC384 Introduction to Artificial Intelligence, University of Toronto

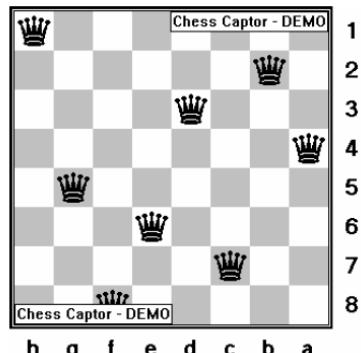
29

Example: N-Queens

- Problem formulation:
 - N variables (N queens)
 - N^2 values for each variable representing the positions on the chessboard
 - Value i is i 'th cell counting from the top left as 1, going left to right, top to bottom.

Example: N-Queens

- $Q_1 = 1, Q_2 = 15, Q_3 = 21, Q_4 = 32,$
 $Q_5 = 34, Q_6 = 44, Q_7 = 54, Q_8 = 59$



Fahiem Bacchus, CSC384 Introduction to Artificial Intelligence, University of Toronto

31

Example: N-Queens

- This representation has $(N^2)^N$ states (different possible assignments in the search space)
 - For 8-Queens: $64^8 = 281,474,976,710,656$
- Is there a better way to represent the N-queens problem?
 - We know we cannot place two queens in a single row → we can exploit this fact in the choice of the CSP representation

Fahiem Bacchus, CSC384 Introduction to Artificial Intelligence, University of Toronto

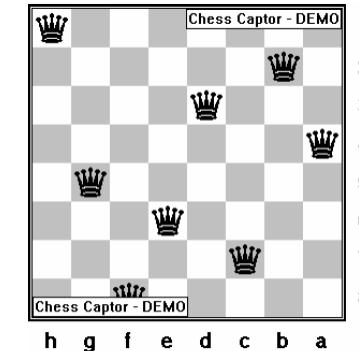
32

Example: N-Queens

- Better Modeling:
 - N variables Q_i , one per row.
 - Value of Q_i is the column the Queen in row i is placed; possible values $\{1, \dots, N\}$.
- This representation has N^N states:
 - For 8-Queens: $8^8 = 16,777,216$
- The choice of a representation can make the problem solvable or unsolvable!

Example: N-Queens

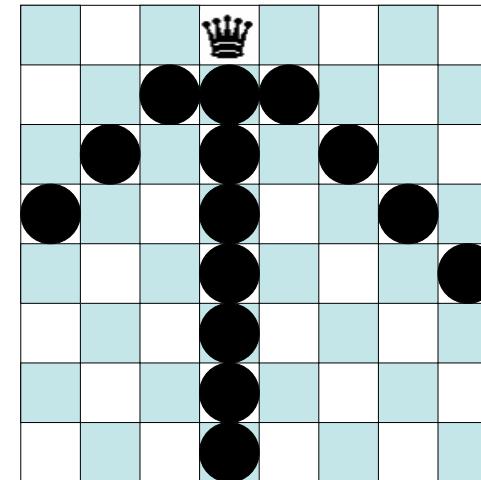
- $Q_1 = 1, Q_2 = 7, Q_3 = 5, Q_4 = 8,$
 $Q_5 = 2, Q_6 = 4, Q_7 = 6, Q_8 = 3$



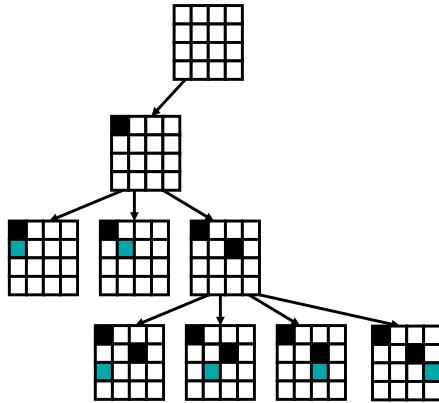
Example: N-Queens

- Constraints:
 - Can't put two Queens in same column
 $Q_i \neq Q_j$ for all $i \neq j$
 - Diagonal constraints
 $\text{abs}(Q_i - Q_j) \neq \text{abs}(i - j)$
 - i.e., the difference in the values assigned to Q_i and Q_j can't be equal to the difference between i and j .

Example: N-Queens



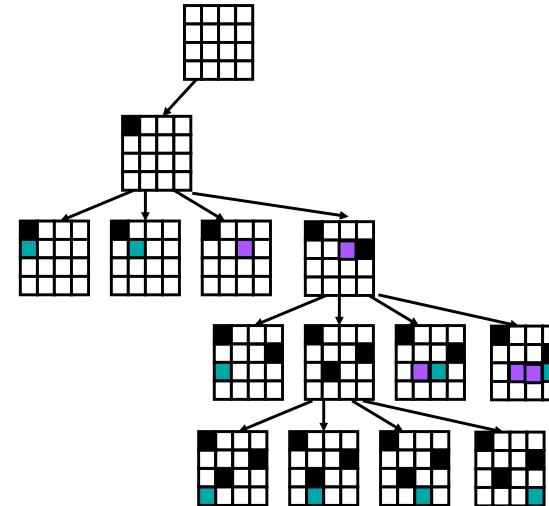
Example: N-Queens



Fahiem Bacchus, CSC384 Introduction to Artificial Intelligence, University of Toronto

37

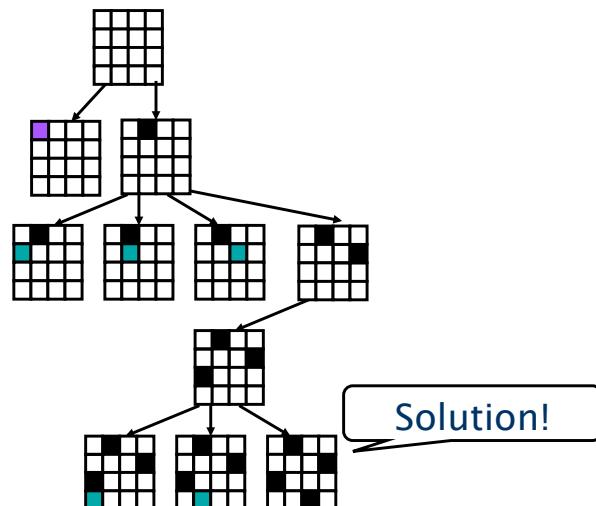
Example: N-Queens



Fahiem Bacchus, CSC384 Introduction to Artificial Intelligence, University of Toronto

38

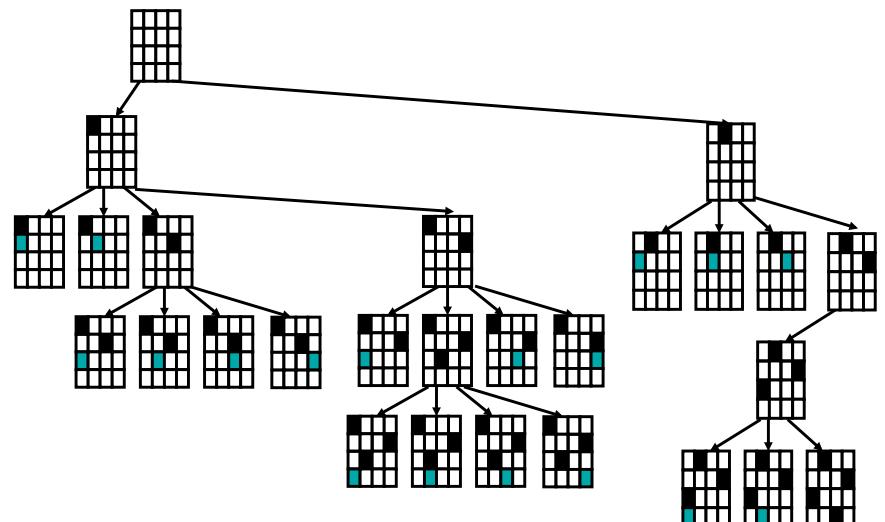
Example: N-Queens



Fahiem Bacchus, CSC384 Introduction to Artificial Intelligence, University of Toronto

39

Example: N-Queens Backtracking Search Space



Fahiem Bacchus, CSC384 Introduction to Artificial Intelligence, University of Toronto

40

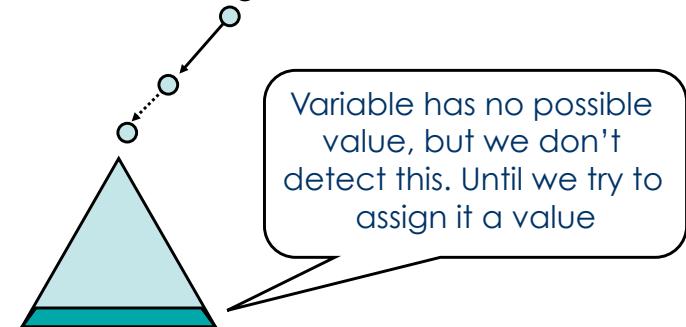
Problems with Plain Backtracking

Sudoku: The 3,3 cell has no possible value.

1	2	3						
			4	5	6			
7								
8								
9								

Problems with Plain Backtracking

- In the backtracking search we won't detect that the (3,3) cell has no possible value until all variables of the row/column (involving row or column 3) or the sub-square constraint (first sub-square) are assigned. So we have the following situation:



- Leads to the idea of **constraint propagation**

Constraint Propagation

- Constraint propagation refers to the technique of "**looking ahead**" at the yet unassigned variables in the search .
- Try to detect obvious failures: "**Obvious**" means things we can test/detect efficiently.
- Even if we don't detect an obvious failure we might be able to eliminate some possible part of the future search.

Constraint Propagation

- Propagation has to be applied during the search; potentially at every node of the search tree.
- Propagation itself is an inference step that needs some resources (in particular time)
 - If propagation is slow, this can slow the search down to the point where using propagation makes finding a solution take longer!
 - There is always a tradeoff between searching fewer nodes in the search, and having a higher nodes/second processing rate.
- We will look at two main types of propagation: Forward Checking & Generalized Arc Consistency

Constraint Propagation: Forward Checking

- Forward checking is an extension of backtracking search that employs a “modest” amount of propagation (look ahead).
- When a variable is instantiated we check all constraints that have **only one uninstantiated variable** remaining.
- For that uninstantiated variable, we check all of its values, pruning those values that violate the constraint.

Fahiem Bacchus, CSC384 Introduction to Artificial Intelligence, University of Toronto

45

Forward Checking Algorithm

- For a single constraint C:

```
FCCheck(C, x)
    // C is a constraint with all its variables already
    // assigned, except for variable x.
    for d := each member of CurDom[x]
        IF making x = d together with previous assignments
            to variables in scope C falsifies C
            THEN remove d from CurDom[x]
    IF CurDom[x] = {} then return DWO (Domain Wipe Out)
    ELSE return ok
```

Fahiem Bacchus, CSC384 Introduction to Artificial Intelligence, University of Toronto

46

Forward Checking Algorithm

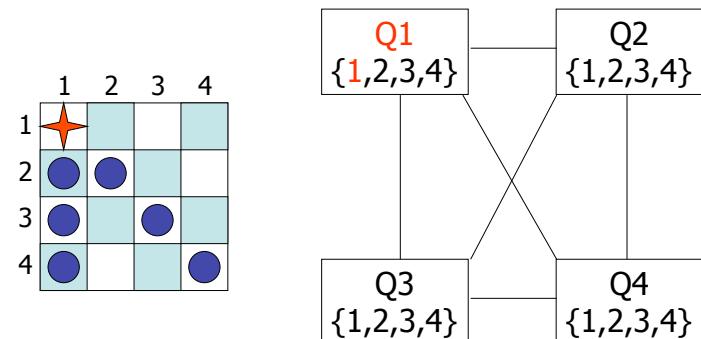
```
FC(Level) /*Forward Checking Algorithm */
    If all variables are assigned
        PRINT Value of each Variable
        RETURN or EXIT (RETURN for more solutions)
            (EXIT for only one solution)
    V := PickAnUnassignedVariable()
    Assigned[V] := TRUE
    for d := each member of CurDom(V)
        Value[V] := d
        DWOoccurred:= False
        for each constraint C over V such that
            a) C has only one unassigned variable X in its scope
            if(FCCheck(C,X) == DWO) /* X domain becomes empty*/
                DWOoccurred:= True
            break /* stop checking constraints */
        if(not DWOoccurred) /*all constraints were ok*/
            FC(Level+1)
        RestoreAllValuesPrunedByFCCheck()
    Assigned[V] := FALSE //undo since we have tried all of V's values
    return;
```

Fahiem Bacchus, CSC384 Introduction to Artificial Intelligence, University of Toronto

47

4-Queens Problem

- Encoding with Q1, ..., Q4 denoting a queen per row
 - cannot put two queens in same column

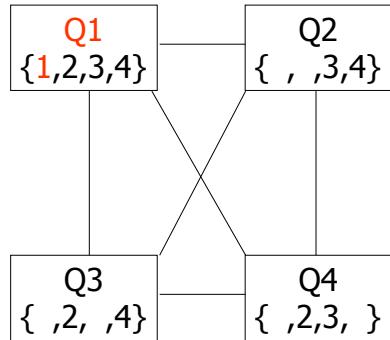
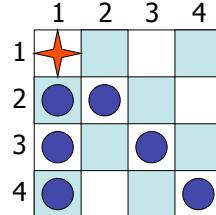


Fahiem Bacchus, CSC384 Introduction to Artificial Intelligence, University of Toronto

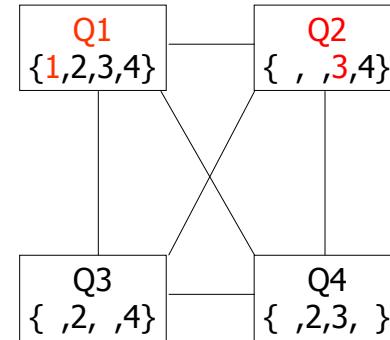
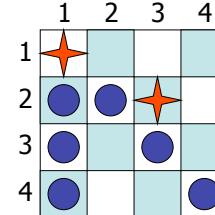
48

4-Queens Problem

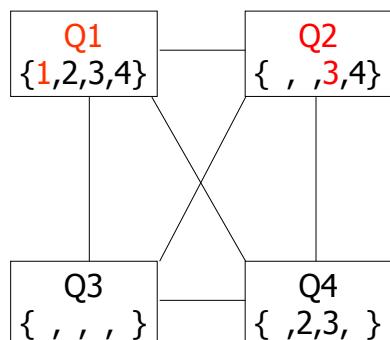
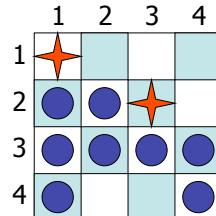
- Forward checking reduced the domains of all variables that are involved in a constraint with one uninstantiated variable:
 - Here all of Q2, Q3, Q4



4-Queens Problem

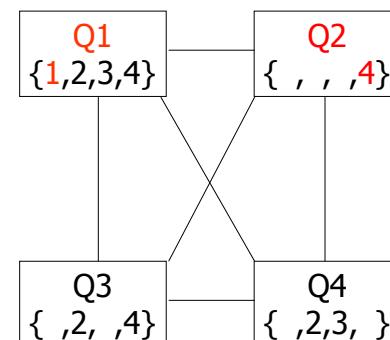
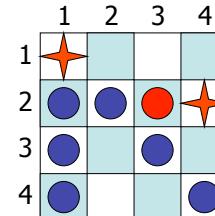


4-Queens Problem

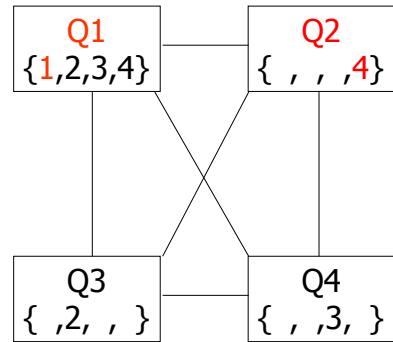
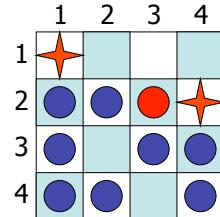


DWO

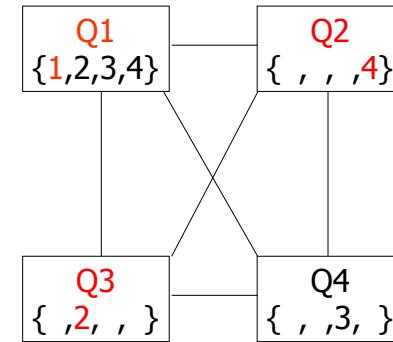
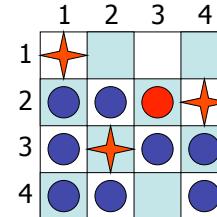
4-Queens Problem



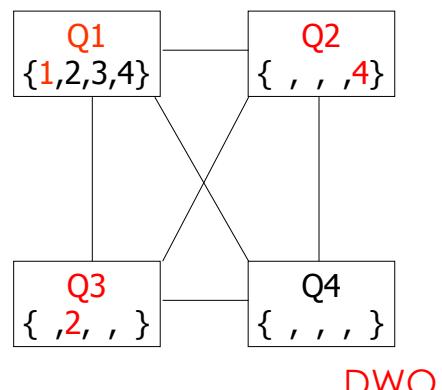
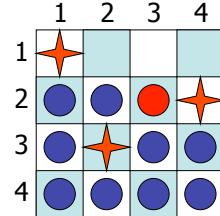
4-Queens Problem



4-Queens Problem

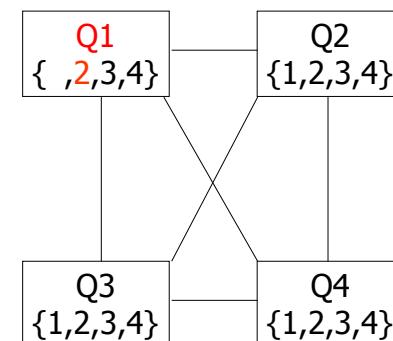
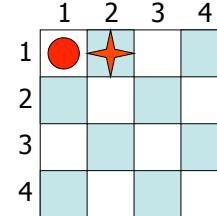


4-Queens Problem

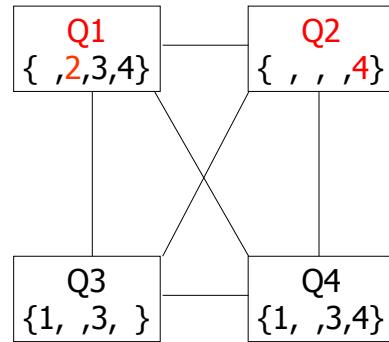
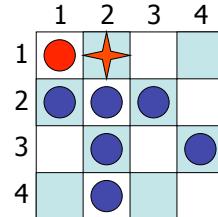


DWO

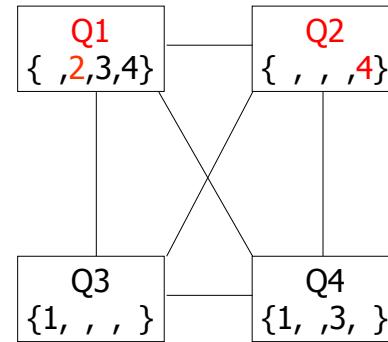
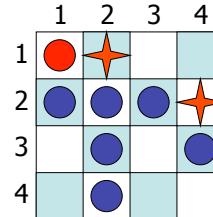
4-Queens Problem



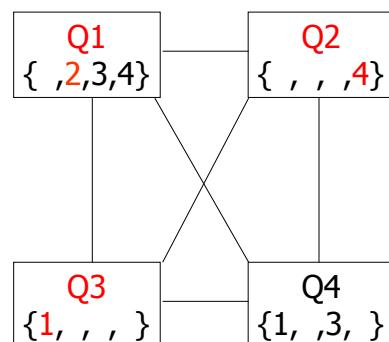
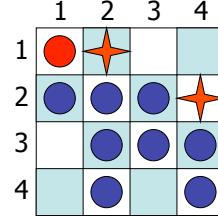
4-Queens Problem



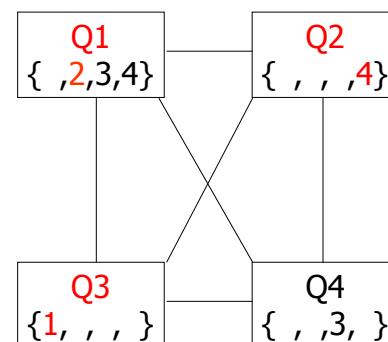
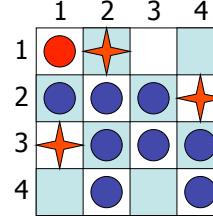
4-Queens Problem



4-Queens Problem

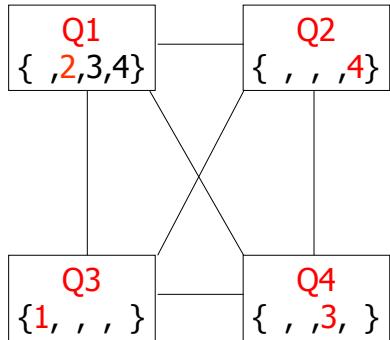
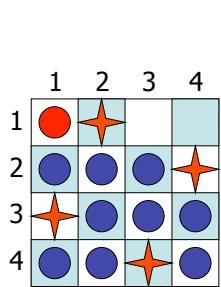


4-Queens Problem

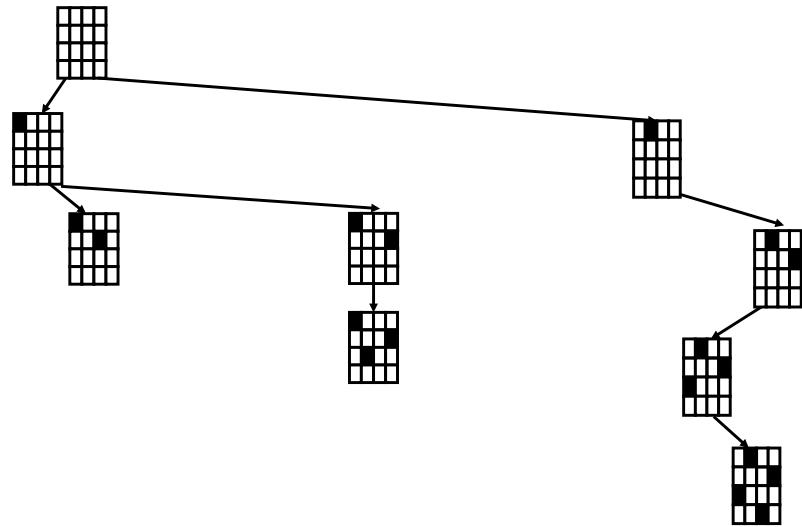


4-Queens Problem

- We have now find a solution: an assignment of all variables to values of their domain so that all constraints are satisfied



Example: N-Queens FC search Space



FC: Restoring Values

- After we backtrack from the current assignment (in the for loop) we must restore the values that were pruned as a result of that assignment.
- Some bookkeeping needs to be done, as we must remember which values were pruned by which assignment (FCCheck is called at every recursive invocation of FC).

FC: Minimum Remaining Values Heuristics (MRV)

- FC also gives us for free a very powerful heuristic to guide us which variables to try next:
 - Always branch on a variable with the smallest remaining values (smallest CurDom).
 - If a variable has only one value left, that value is forced, so we should propagate its consequences immediately.
 - This heuristic tends to produce skinny trees at the top. This means that more variables can be instantiated with fewer nodes searched, and thus more constraint propagation/DWO failures occur when the tree starts to branch out (we start selecting variables with larger domains)
 - We can find a inconsistency much faster

MRV Heuristic: Human Analogy

- What variables would you try first?

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

Domain of each variable:
 $\{1, \dots, 9\}$

(1, 5): impossible values:
Row: $\{1, 4, 5, 6, 8\}$
Column: $\{1, 3, 4, 5, 7, 9\}$
Subsquare: $\{5, 7, 9\}$
 \rightarrow Domain = {2}

(9, 5): impossible values:
Row: $\{1, 5, 7, 8, 9\}$
Column: $\{1, 3, 4, 5, 7, 9\}$
Subsquare: $\{1, 5, 7, 9\}$
 \rightarrow Domain = {2, 6}

After assigning value 2 to
cell (1,5): Domain = {6}

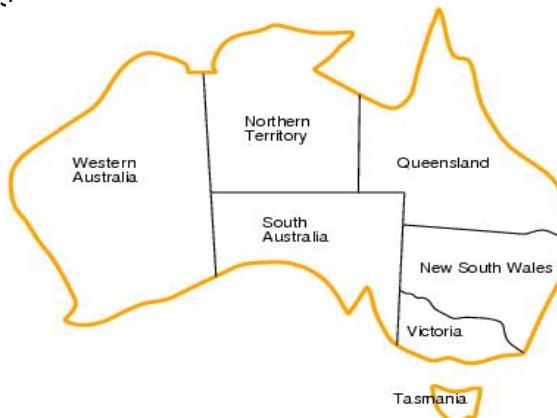
Most restricted variables! = MRV

Fahiem Bacchus, CSC384 Introduction to Artificial Intelligence, University of Toronto

65

Example – Map Colouring

- Color the following map using red, green, and blue such that adjacent regions have different colors



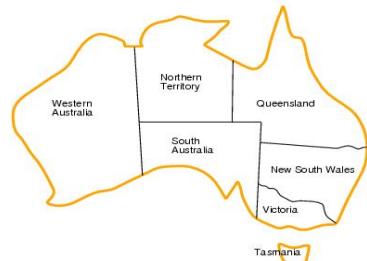
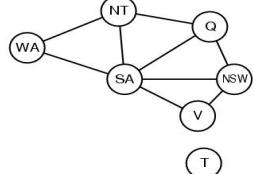
Fahiem Bacchus, CSC384 Introduction to Artificial Intelligence, University of Toronto

66

Example – Map Colouring

• Modeling

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

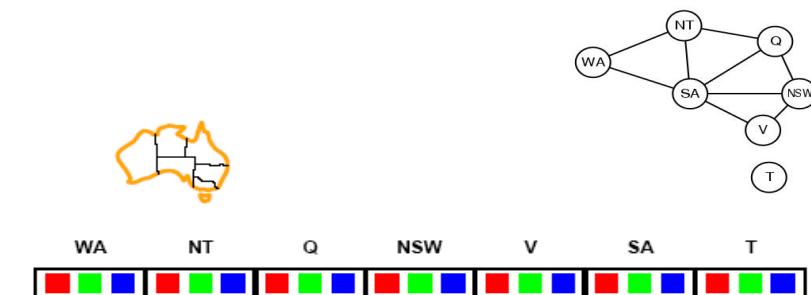


Fahiem Bacchus, CSC384 Introduction to Artificial Intelligence, University of Toronto

67

Example – Map Colouring

- *Forward checking idea:* keep track of remaining legal values for unassigned variables.
- Terminate search when any variable has no legal values.

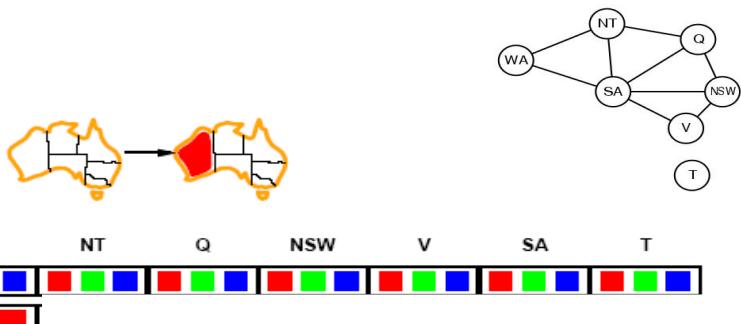


Fahiem Bacchus, CSC384 Introduction to Artificial Intelligence, University of Toronto

68

Example – Map Colouring

- Assign {WA=red}
- Effects on other variables connected by constraints to WA
 - NT can no longer be red
 - SA can no longer be red

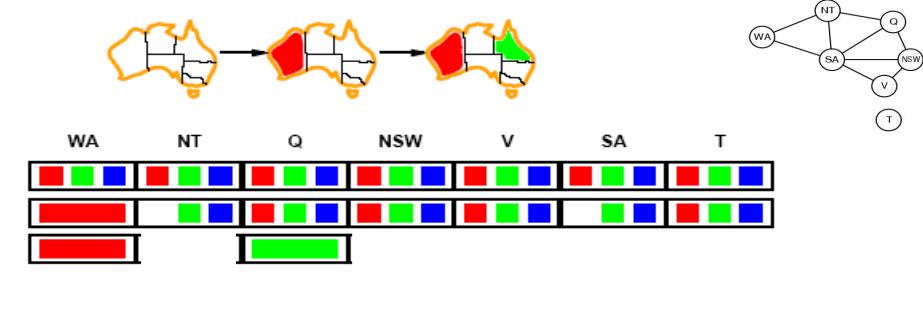


Fahiem Bacchus, CSC384 Introduction to Artificial Intelligence, University of Toronto

69

Example – Map Colouring

- Assign {Q=green} (Note: Not using MRV)
- Effects on other variables connected by constraints with Q
 - NT can no longer be green
 - NSW can no longer be green
 - SA can no longer be green
- MRV heuristic would automatically select NT or SA next

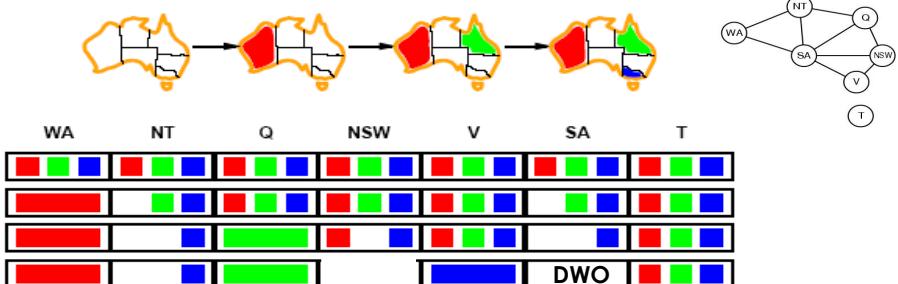


Fahiem Bacchus, CSC384 Introduction to Artificial Intelligence, University of Toronto

70

Example – Map Colouring

- Assign {V=blue} (not using MRV)
- Effects on other variables connected by constraints with V
 - NSW can no longer be blue
 - SA is empty
- FC has detected that partial assignment is inconsistent with the constraints and backtracking can occur.



Fahiem Bacchus, CSC384 Introduction to Artificial Intelligence, University of Toronto

71

Empirically

- FC often is about 100 times faster than BT
- FC with MRV (minimal remaining values) often 10000 times faster.
- But on some problems the speed up can be much greater
 - Converts problems that are not solvable to problems that are solvable.
- Still FC is not that powerful. Other more powerful forms of constraint propagation are used in practice.
- Try the previous map coloring example with MRV.

Fahiem Bacchus, CSC384 Introduction to Artificial Intelligence, University of Toronto

72

Constraint Propagation: Generalized Arc Consistency

GAC—Generalized Arc Consistency

1. $C(V_1, V_2, V_3, \dots, V_n)$ is GAC with respect to variable V_i , if and only if

For every value of V_i , there exists values of $V_1, V_2, V_{i-1}, V_{i+1}, \dots, V_n$ that satisfy C .

Note that the values are removed from the variable domains during search. So variables that are GAC in a constraint might become inconsistent (non-GAC).

Constraint Propagation: Generalized Arc Consistency

- $C(V_1, V_2, \dots, V_n)$ is GAC if and only if

It is GAC with respect to every variable in its scope.

- A CSP is GAC if and only if

all of its constraints are GAC.

Constraint Propagation: Arc Consistency

- Say we find a value d of variable V_i that is not consistent: That is, there is no assignments to the other variables that satisfy the constraint when $V_i = d$

- d is said to be **Arc Inconsistent**
 - We **can remove** d from the domain of V_i —this value cannot lead to a solution (much like Forward Checking, but more powerful).

- e.g. $C(X,Y): X > Y$ $\text{Dom}(X)=\{1,5,11\}$ $\text{Dom}(Y)=\{3,8,15\}$

- For $X=1$ there is no value of Y s.t. $1 > Y \Rightarrow$ so we can remove 1 from domain X
 - For $Y=15$ there is no value of X s.t. $X > 15$, so remove 15 from domain Y
 - We obtain more restricted domains $\text{Dom}(X)=\{5,11\}$ and $\text{Dom}(Y)=\{3,8\}$

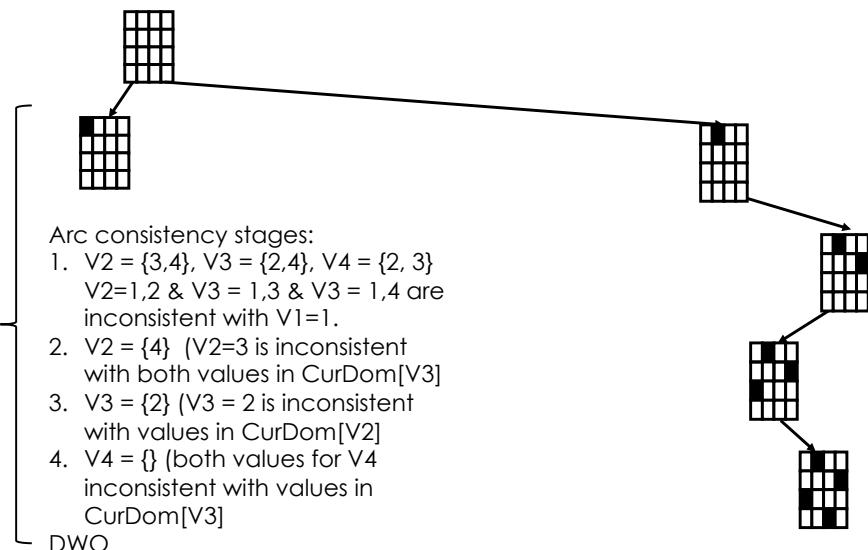
Constraint Propagation: Arc Consistency

- If we apply arc consistency propagation during search the search tree's size will typically be much reduced in size.

- Removing a value from a variable domain may trigger further inconsistency, so we have to repeat the procedure until everything is consistent.

- We put constraints on a queue and add new constraints to the queue as we need to check for arc consistency.

Example: N-Queens GAC search Space

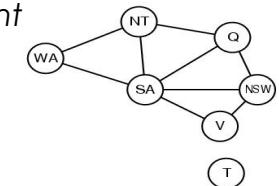


Fahiem Bacchus, CSC384 Introduction to Artificial Intelligence, University of Toronto

77

Example – Map Colouring

- Assign $\{WA=\text{red}\}$
- Effects on other variables connected by constraints to WA
 - NT can no longer be red = $\{G, B\}$
 - SA can no longer be red = $\{G, B\}$
- All other values are arc-consistent



Fahiem Bacchus, CSC384 Introduction to Artificial Intelligence, University of Toronto

78

Example – Map Colouring

- Assign $\{Q=\text{green}\}$
- Effects on other variables connected by constraints with Q
 - NT can no longer be green = $\{B\}$
 - NSW can no longer be green = $\{R, B\}$
 - SA can no longer be green = $\{B\}$
- DWO there is no value for SA that will be consistent with $NT \neq SA$ and $NT = B$

Note Forward Checking would not have detected this DWO.

Fahiem Bacchus, CSC384 Introduction to Artificial Intelligence, University of Toronto

79

GAC Algorithm

- We make all constraints GAC at every node of the search space.
- This is accomplished by removing from the domains of the variables all arc inconsistent values.

Fahiem Bacchus, CSC384 Introduction to Artificial Intelligence, University of Toronto

80

GAC Algorithm, enforce GAC during search

```
GAC(Level) /*Maintain GAC Algorithm */
  If all variables are assigned
    PRINT Value of each Variable
    RETURN or EXIT (RETURN for more solutions)
      (EXIT for only one solution)
  V := PickAnUnassignedVariable()
  Assigned[V] := TRUE
  for d := each member of CurDom(V)
    Value[V] := d
    Prune all values of V ≠ d from CurDom[V]
    for each constraint C whose scope contains V
      Put C on GACQueue
    if(GAC_Enforce() != DWO)
      GAC(Level+1) /*all constraints were ok*/
      RestoreAllValuesPrunedFromCurDoms()
  Assigned[V] := FALSE
  return;
```

Enforce GAC (prune all GAC inconsistent values)

```
GAC_Enforce()
  // GAC-Queue contains all constraints one of whose variables has
  // had its domain reduced. At the root of the search tree
  // first we run GAC_Enforce with all constraints on GAC-Queue
  while GACQueue not empty
    C = GACQueue.extract()
    for V := each member of scope(C)
      for d := CurDom[V]
        Find an assignment A for all other
        variables in scope(C) such that
        C(A U V=d) = True
        if A not found
          CurDom[V] = CurDom[V] - d
          if CurDom[V] = ∅
            empty GACQueue
            return DWO //return immediately
          else
            push all constraints C' such that
            V ∈ scope(C') and C' ∉ GACQueue
            on to GACQueue
    return TRUE //while loop exited without DWO
```

Enforce GAC

- A **support** for $V=d$ in constraint C is an assignment **A** to all of the other variables in $\text{scope}(C)$ such that $C \cup \{V=d\}$ satisfies C . (**A** is what the algorithm's inner loop looks for).
- Smarter implementations keep track of "supports" to avoid having to search through all possible assignments to the other variables for a satisfying assignment.

Enforce GAC

- Rather than search for a satisfying assignment to C containing $V=d$, we check to see if the current support is still valid: i.e., all values it assigns still lie in the variable's current domains
- Also we take advantage that a support for $V=d$, e.g. $\{V=d, X=a, Y=b, Z=c\}$ is also a support for $X=a$, $Y=b$, and $Z=c$

Enforce GAC

- However, finding a support for $V=d$ in constraint C still in the worst case requires $O(2^k)$ work, where k is the **arity** of C , i.e., $|\text{scope}(C)|$.
- Another key development in practice is that for some constraints this computation can be done in polynomial time.
E.g., all-diff(V_1, \dots, V_n) we can be check if $V_i=d$ has a support in the current domains of the other variables in polynomial time using ideas from graph theory.
We do not need to examine all combinations of values for the other variables looking for a support

GAC enforce example

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

 = All-diff

$$C_{SS2} = \text{All-diff}(V_{1,4}, V_{1,5}, V_{1,6}, V_{2,4}, V_{2,5}, V_{2,6}, V_{3,4}, V_{3,5}, V_{3,6})$$

$$C_{R1} = \text{All-diff}(V_{1,1}, V_{1,2}, V_{1,3}, V_{1,4}, V_{1,5}, V_{1,6}, V_{1,7}, V_{1,8}, V_{1,9})$$

$$C_{CS} = \text{All-diff}(V_{1,5}, V_{2,5}, V_{3,5}, V_{4,5}, V_{5,5}, V_{6,5}, V_{7,5}, V_{8,5}, V_{9,5})$$

By going back and forth between constraints we get more values pruned.

$$\text{GAC}(C_{SS8}) \rightarrow \text{CurDom of } V_{1,5}, V_{1,6}, V_{2,4}, V_{3,4}, V_{3,6} = \{1, 2, 3, 4, 8\}$$

$$\text{GAC}(C_{R1}) \rightarrow \text{CurDom of } V_{1,7}, V_{1,8} = \{2, 3, 7, 9\}$$
$$\text{CurDom of } V_{1,5}, V_{1,6} = \{2, 3\}$$

$$\text{GAC}(C_{SS8}) \rightarrow \text{CurDom of } V_{2,4}, V_{3,4}, V_{3,6} = \{1, 4, 8\}$$

$$\text{GAC}(C_{CS}) \rightarrow \text{CurDom of } V_{5,5}, V_{9,5} = \{2, 6, 8\}$$
$$\rightarrow \text{CurDom of } V_{1,5} = \{2\}$$

$$\text{GAC}(C_{SS8}) \rightarrow \text{CurDom of } V_{1,6} = \{3\}$$

Many real-world applications of CSP

- Assignment problems
 - who teaches what class
- Timetabling problems
 - exam schedule
- Transportation scheduling
- Floor planning
- Factory scheduling
- Hardware configuration
 - a set of compatible components

CSC384: Introduction to Artificial Intelligence

Game Tree Search

- Chapter 5.1, 5.2, 5.3, 5.6 cover some of the material we cover here. Section 5.6 has an interesting overview of State-of-the-Art game playing programs.
- Section 5.5 extends the ideas to games with uncertainty (We won't cover that material but it makes for interesting reading).

Generalizing Search Problem

- So far: our search problems have assumed agent has complete control of environment
 - State does not change unless the agent (robot) changes it.
 - All we need to compute is a single path to a goal state.
- Assumption not always reasonable
 - Stochastic environment (e.g., the weather, traffic accidents).
 - Other agents whose interests conflict with yours
 - Search can find a path to a goal state, but the actions might not lead you to the goal as the state can be changed by other agents (nature or other intelligent agents)

Generalizing Search Problem

- We need to generalize our view of search to handle state changes that are not in the control of the agent.
- One generalization yields game tree search
 - Agent and some other agents.
 - The other agents are acting to maximize their profits
 - this might not have a positive effect on your profits.

General Games

- What makes something a game?
 - There are two (or more) agents making changes to the world (the state)
 - Each agent has their own interests
 - e.g., each agent has a different goal; or assigns different costs to different paths/states
 - Each agent tries to alter the world so as to best benefit itself.

General Games

- What makes games hard?
 - How you should play depends on how you think the other person will play; but how they play depends on how they think you will play; so how you should play depends on how you think they think you will play; but how they play should depend on how they think you think they think you will play; ...

Properties of Games considered here

- Zero-sum games: Fully competitive
 - Competitive: if one play wins, the others lose; e.g. *Poker* – you win what the other player lose
 - Games can also be cooperative: some outcomes are preferred by both of us, or at least our values aren't diametrically opposed
- Deterministic: no chance involved
 - (no dice, or random deals of cards, or coin flips, etc.)
- Perfect information (all aspects of the state are fully observable, e.g., no hidden cards)

Our Focus: Two-Player Zero-Sum Games

- Fully competitive two player games
 - If you win, the other player (opponent) loses
 - Zero-sum means the sum of your and your opponent's payoff is zero---any thing you gain come at your opponent's cost (and vice-versa).
 - Key insight: How you act depends on how the other agent acts (or how you think they will act)
 - and vice versa (if your opponent acts rational)
- Examples of two-person zero-sum games:
 - Chess, checkers, tic-tac-toe, backgammon, go, Doom, "find the last parking space"
- Most of the ideas extend to multiplayer zero-sum games (cf. Chapter 5.2.2)

Game 1: Rock, Paper, Scissors

- Scissors cut paper, paper covers rock, rock smashes scissors
- Represented as a matrix: Player I chooses a row, Player II chooses a column
- Payoff to each player in each cell (Pl.I / Pl.II)
- 1: win, 0: tie, -1: loss
 - so it's zero-sum

		Player II		
		R	P	S
Player I		R	0/0	-1/1
		P	1/-1	0/0
S		-1/1	1/-1	0/0

Game 2: Prisoner's Dilemma

- Two prisoner's in separate cells, sheriff doesn't have enough evidence to convict them. They agree ahead of time to both deny the crime (*they will cooperate*).
- If one defects (i.e., confesses) and the other doesn't
 - confessor goes free
 - other sentenced to 4 years
- If both defect (confess)
 - both sentenced to 3 years
- If both cooperate (neither confesses)
 - both sentenced to 1 year on minor charge
- Payoff: 4 minus sentence

	Coop	Def
Coop	3/3	0/4
Def	4/0	1/1

Extensive Form Two-Player Zero-Sum Games

- Key point of previous games: what you should do depends on what other guy does
- But previous games are simple “one shot” games
 - single move each
 - in game theory: *strategic or normal form games*
- Many games extend over multiple moves
 - turn-taking: players act alternatively
 - e.g., chess, checkers, etc.
 - in game theory: *extensive form games*
- We'll focus on the extensive form
 - that's where the computational questions emerge

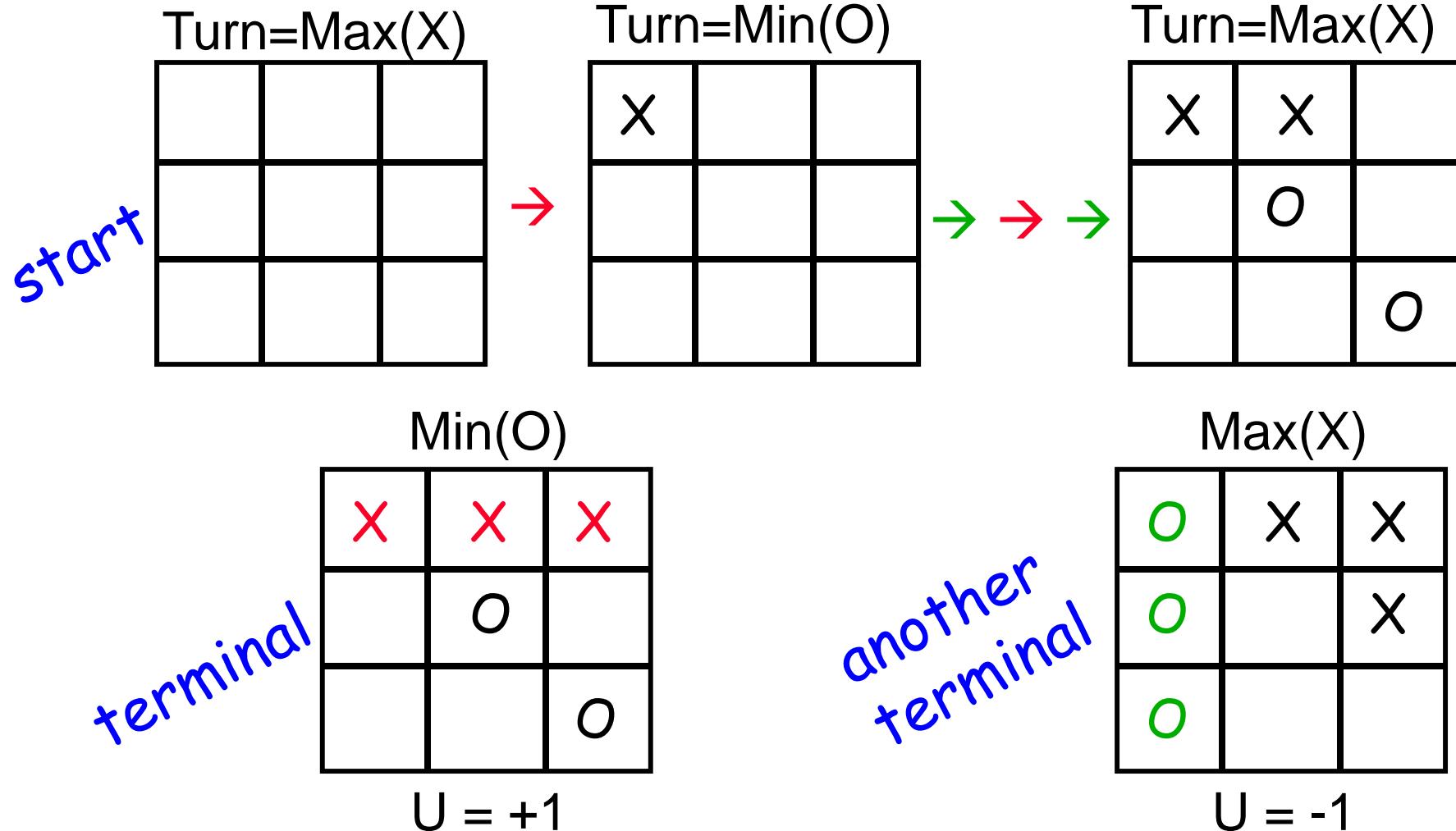
Two-Player Zero-Sum Game – Definition

- Two *players* A (Max) and B (Min)
- Set of *positions* P (states of the game)
- A *starting position* $s \in P$ (where game begins)
- *Terminal positions* $T \subseteq P$ (where game can end)
- Set of directed edges E_A between states (A's *moves*)
- set of directed edges E_B between states (B's *moves*)
- *Utility* or *payoff function* $U : T \rightarrow \mathbb{R}$ (how good is each terminal state for player A)
 - Why don't we need a utility function for B?

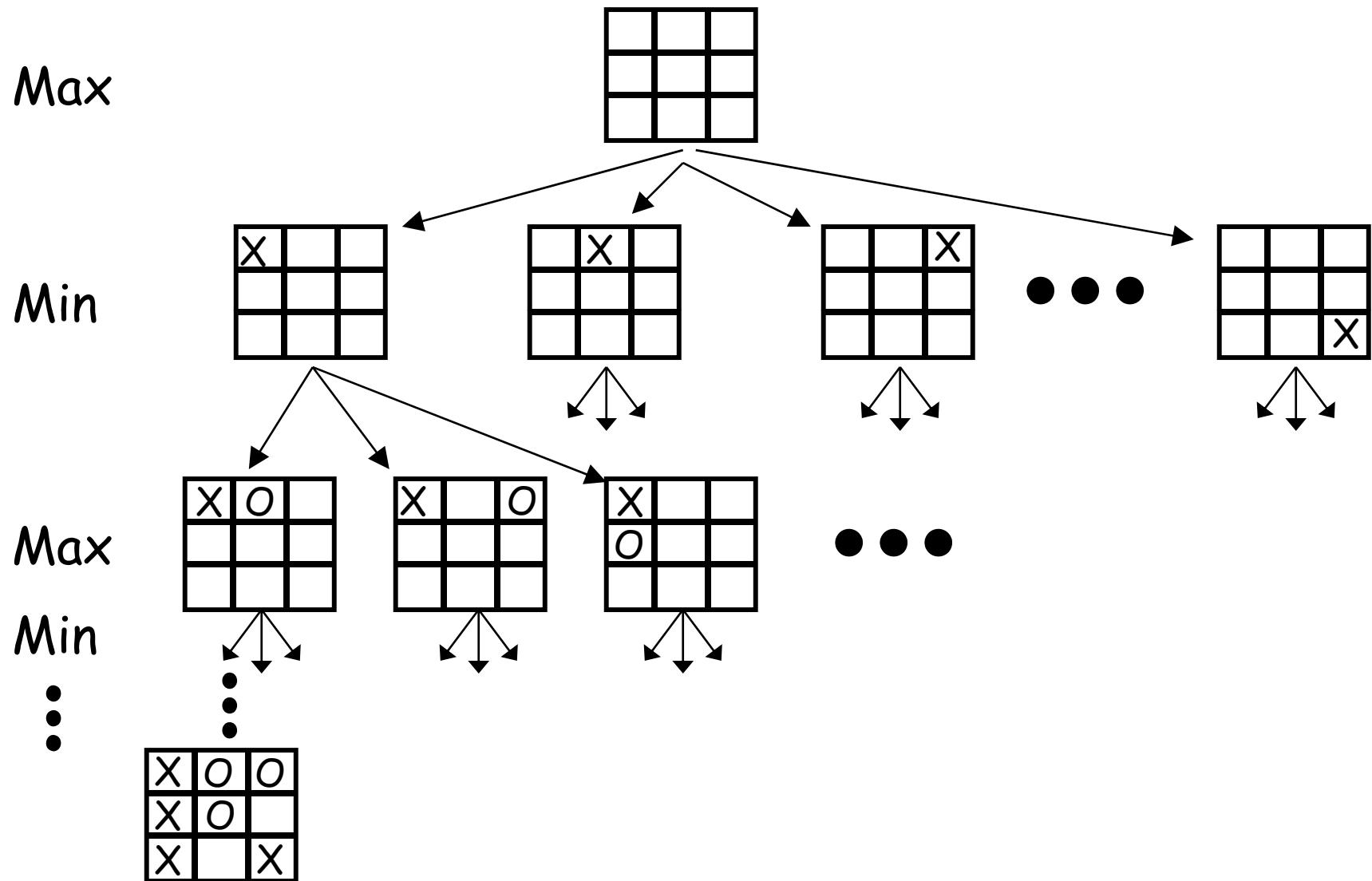
Two-Player Zero-Sum Game – Intuition

- Players alternate moves (starting with Max)
 - Game ends when some terminal $p \in T$ is reached
- A game **state**: a state-player pair
 - Tells us what state we're in and whose move it is
- Utility function and terminals replace goals
 - Max wants to maximize the terminal payoff
 - Min wants to minimize the terminal payoff
- Think of it as:
 - Max gets $U(t)$, Min gets $-U(t)$ for terminal node t
 - This is why it's called zero (or constant) sum

Tic Tac Toe States



Tic Tac Toe Game Tree



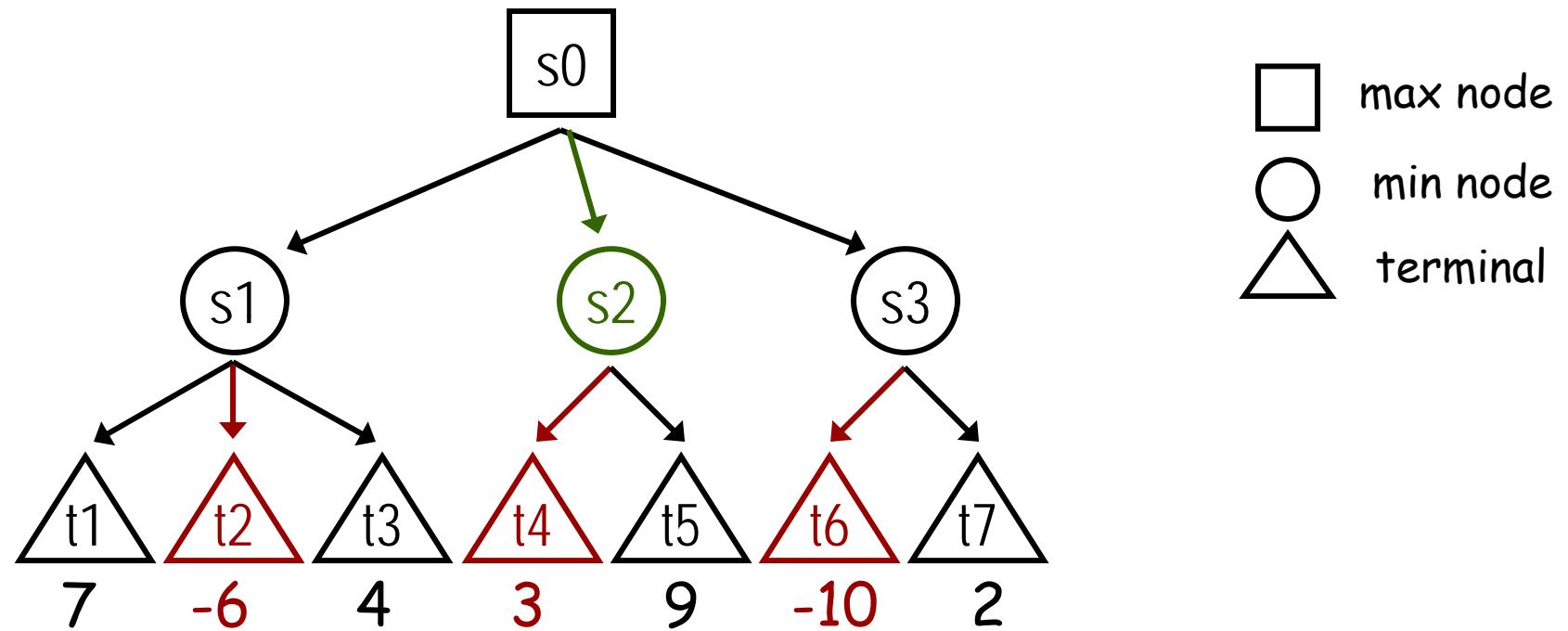
Game Tree

- Game tree looks like a search tree
 - Layers reflect alternating moves between A and B
 - The search tree in game playing is a subtree of the game tree
- Player A doesn't decide where to go alone
 - After A moves to a state, B decides which of the states children to move to
- Thus A must have a *strategy*
 - Must know what to do for each possible move of B
 - One sequence of moves will not suffice: "What to do" will depend on how B will play
- What is a reasonable strategy?

Minimax Strategy

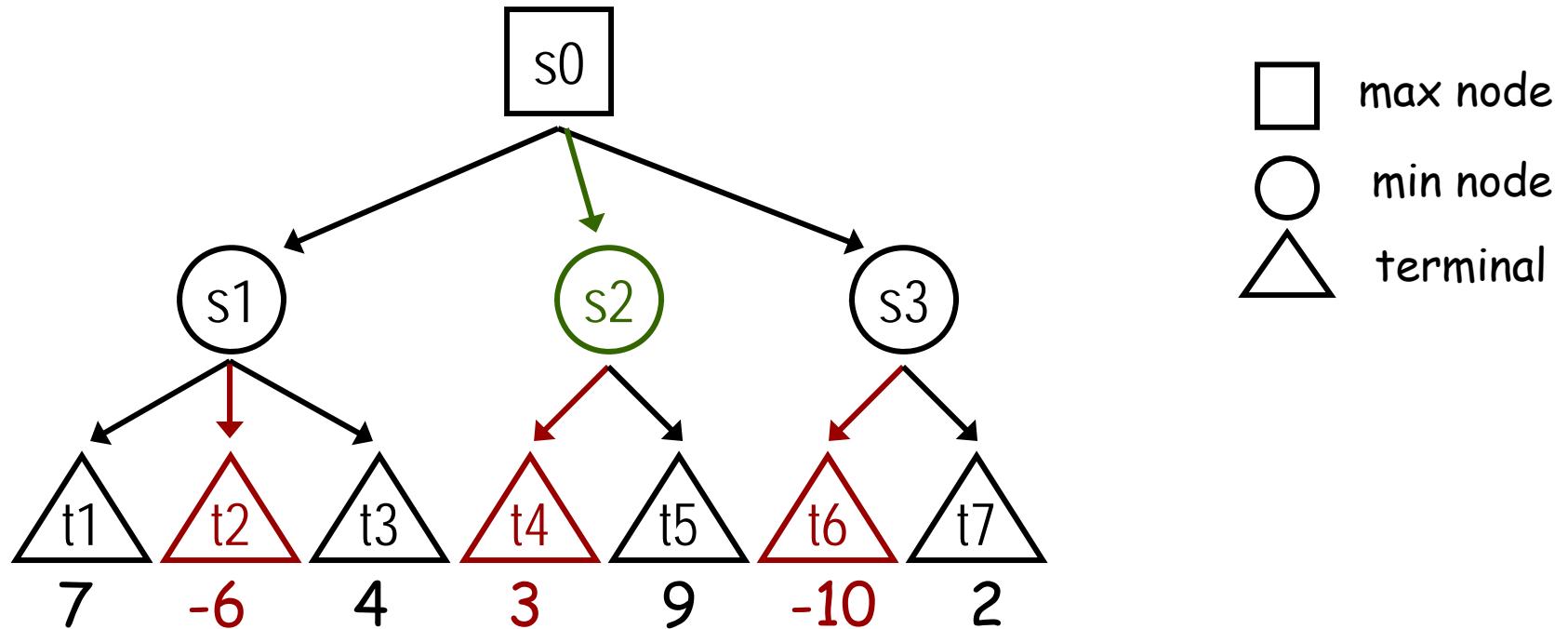
- Assume that the other player will always play their best move,
 - you always play a move that will minimize the payoff that could be gained by the other player.
 - By minimizing the other player's payoff you maximize yours.
- If however you know that Min will play poorly in some circumstances, there might be a better strategy than MiniMax (i.e., a strategy that gives you a better payoff).
- But in the absence of that knowledge minimax “plays it safe”

Minimax Strategy payoffs



The terminal nodes have utilities.
But we can compute a “utility” for the non-terminal states, by assuming both players always play their best move.

Minimax Strategy – Intuitions



If Max goes to s_1 , Min goes to t_2 , $U(s_1) = \min\{U(t_1), U(t_2), U(t_3)\} = -6$
If Max goes to s_2 , Min goes to t_4 , $U(s_2) = \min\{U(t_4), U(t_5)\} = 3$
If Max goes to s_3 , Min goes to t_6 , $U(s_3) = \min\{U(t_6), U(t_7)\} = -10$

So Max goes to s_2 : so $U(s_0) = \max\{U(s_1), U(s_2), U(s_3)\} = 3$

Minimax Strategy

- Build full game tree (all leaves are terminals)
 - Root is start state, edges are possible moves, etc.
 - Label terminal nodes with utilities
- Back values up the tree
 - $U(t)$ is defined for all terminals (part of input)
 - $U(n) = \min \{U(c) : c \text{ is a child of } n\}$ if n is a Min node
 - $U(n) = \max \{U(c) : c \text{ is a child of } n\}$ if n is a Max node

Minimax Strategy

- The values labeling each state are the values that Max will achieve in that state if both Max and Min play their best moves.
 - Max plays a move to change the state to the highest valued min child.
 - Min plays a move to change the state to the lowest valued max child.
- If Min plays poorly, Max could do better, but never worse.
 - If Max, however knows that Min will play poorly, there might be a better strategy of play for Max than Minimax.

Depth-First Implementation of Minimax

- *Building the entire game tree and backing up values gives each player their strategy.*
- *However, the game tree is exponential in size.*
- *Furthermore, as we will see later it is not necessary to know all of the tree.*
- To solve these problems we find a **depth-first** implementation of minimax.

We run the depth-first search after each move to compute what is the next move for the **MAX** player. (We could do the same for the **MIN** player).

- This avoids explicitly representing the exponentially sized game tree: we just compute each move as it is needed.

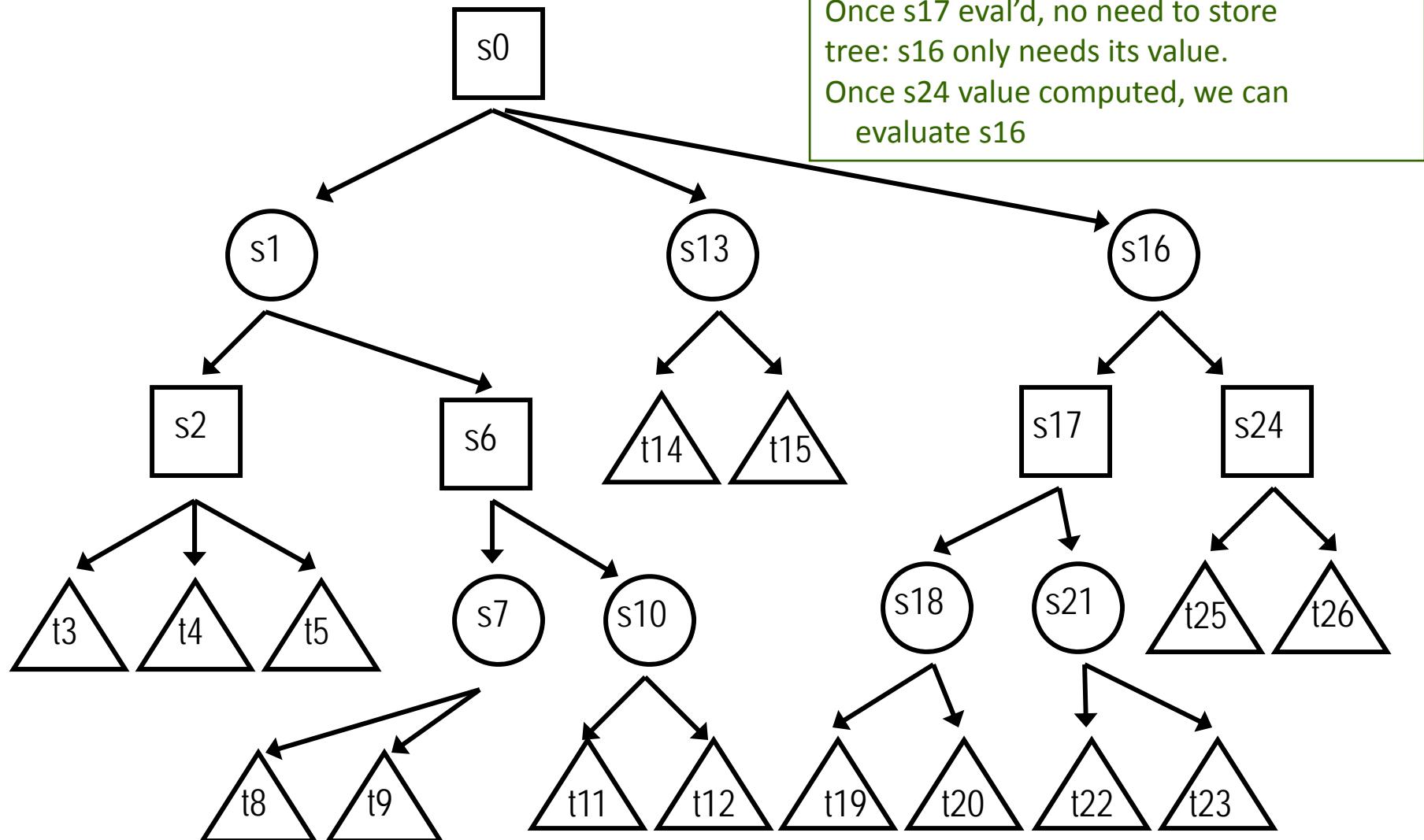
Depth-First Implementation of Minimax

```
DFMiniMax(n, Player) //return Utility of state n given that
                      //Player is MIN or MAX
If n is TERMINAL
    Return U(n) //Return terminal states utility
                  //(U is specified as part of game)
                  //Apply Player's moves to get
                  //successor states.
ChildList = n.Successors(Player)
If Player == MIN
    return minimum of DFMiniMax(c, MAX) over c ∈ ChildList
Else //Player is MAX
    return maximum of DFMiniMax(c, MIN) over c ∈ ChildList
```

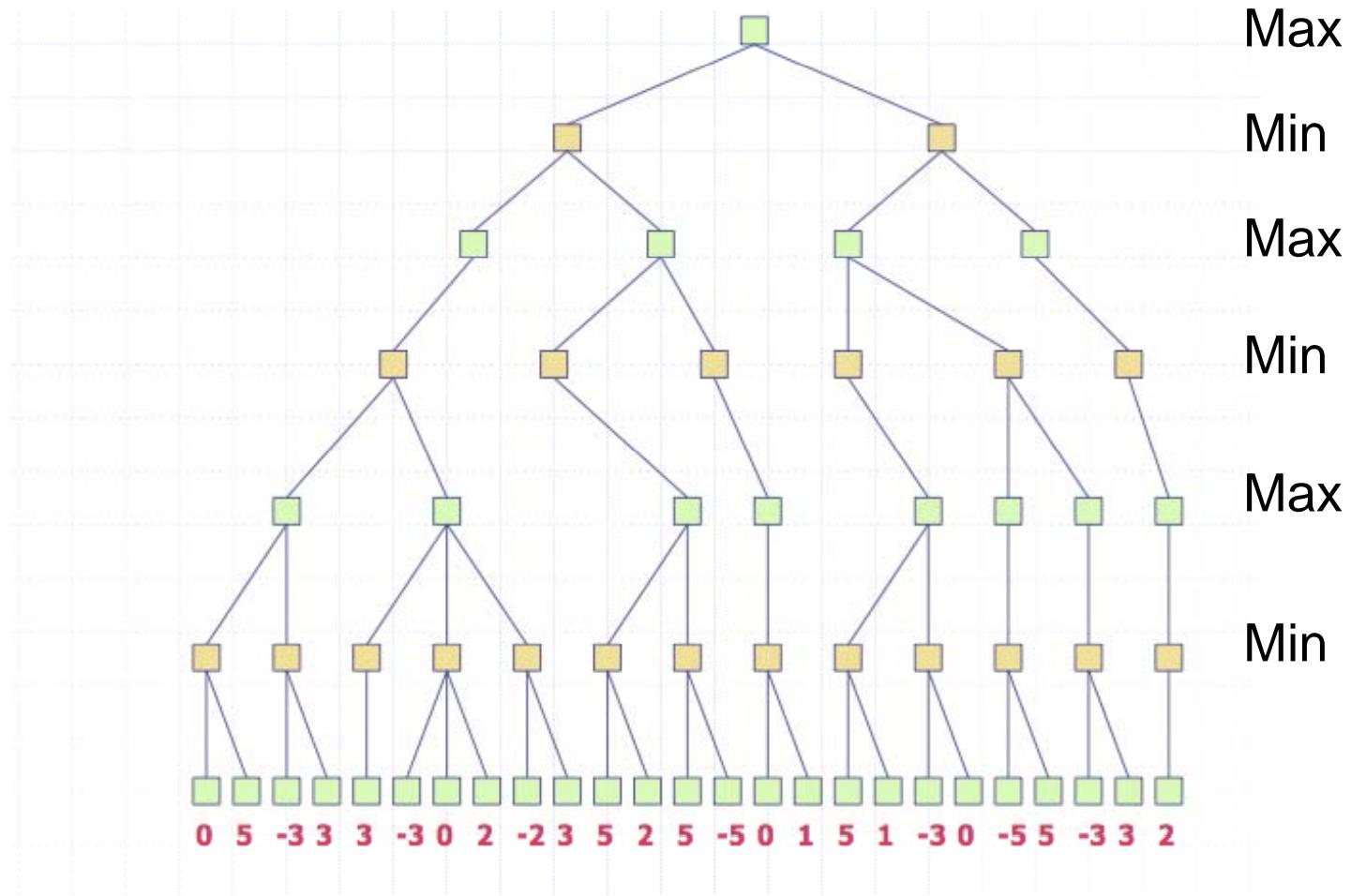
Depth-First Implementation of Minimax

- Notice that the game tree has to have finite depth for this to work
- Advantage of DF implementation: space efficient
- Minimax will expand $O(b^d)$ states, which is both a BEST and WORSE case scenario.
 - We must traverse the entire search tree to evaluate all options
 - We can't be lucky as in regular search and find a path to a goal before searching the entire tree.

Visualization of Depth-First Minimax



Example

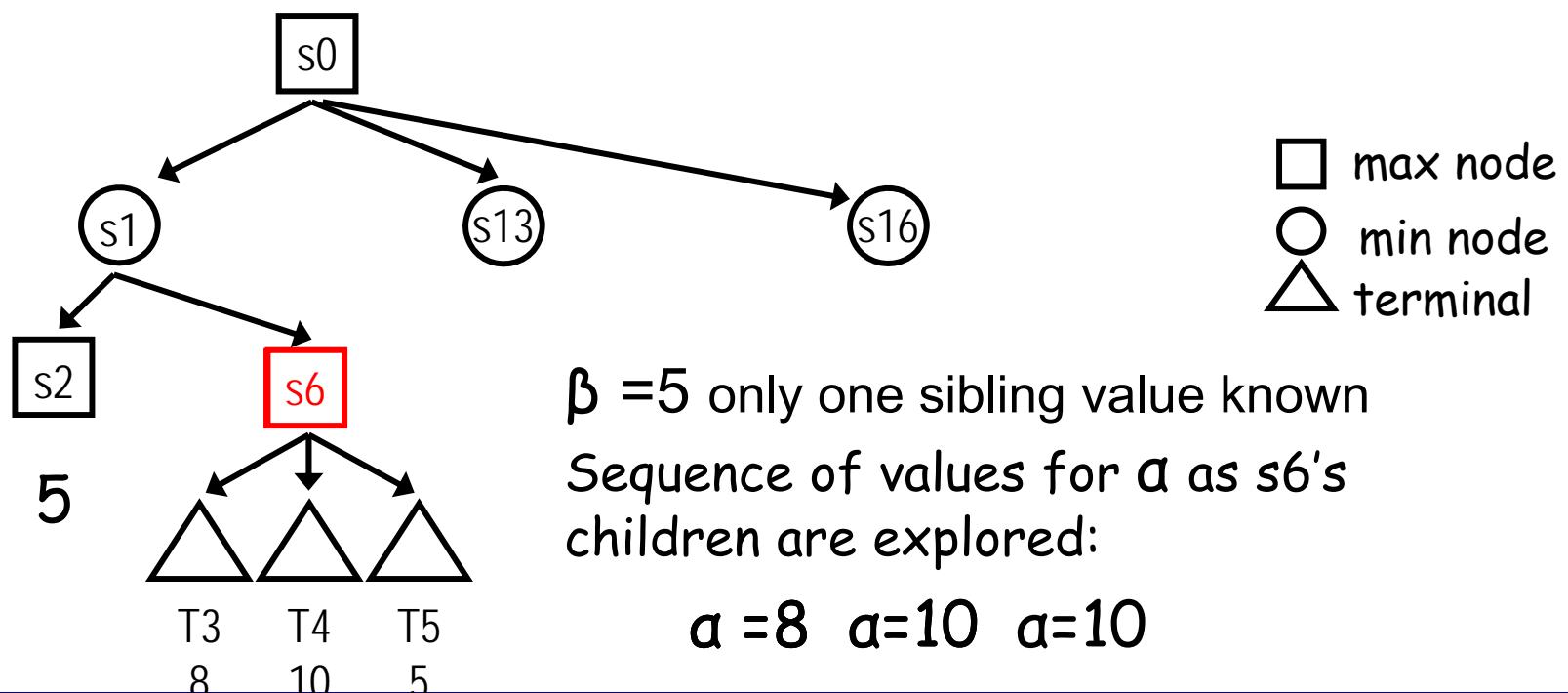


Pruning

- *It is not necessary to examine entire tree to make correct Minimax decision*
- Assume depth-first generation of tree
 - After generating value for only *some* of n 's children we can prove that we'll never reach n in a Minmax strategy.
 - So we needn't generate or evaluate any further children of n !
- Two types of pruning (cuts):
 - pruning of max nodes (*α -cuts*)
 - pruning of min nodes (*β -cuts*)

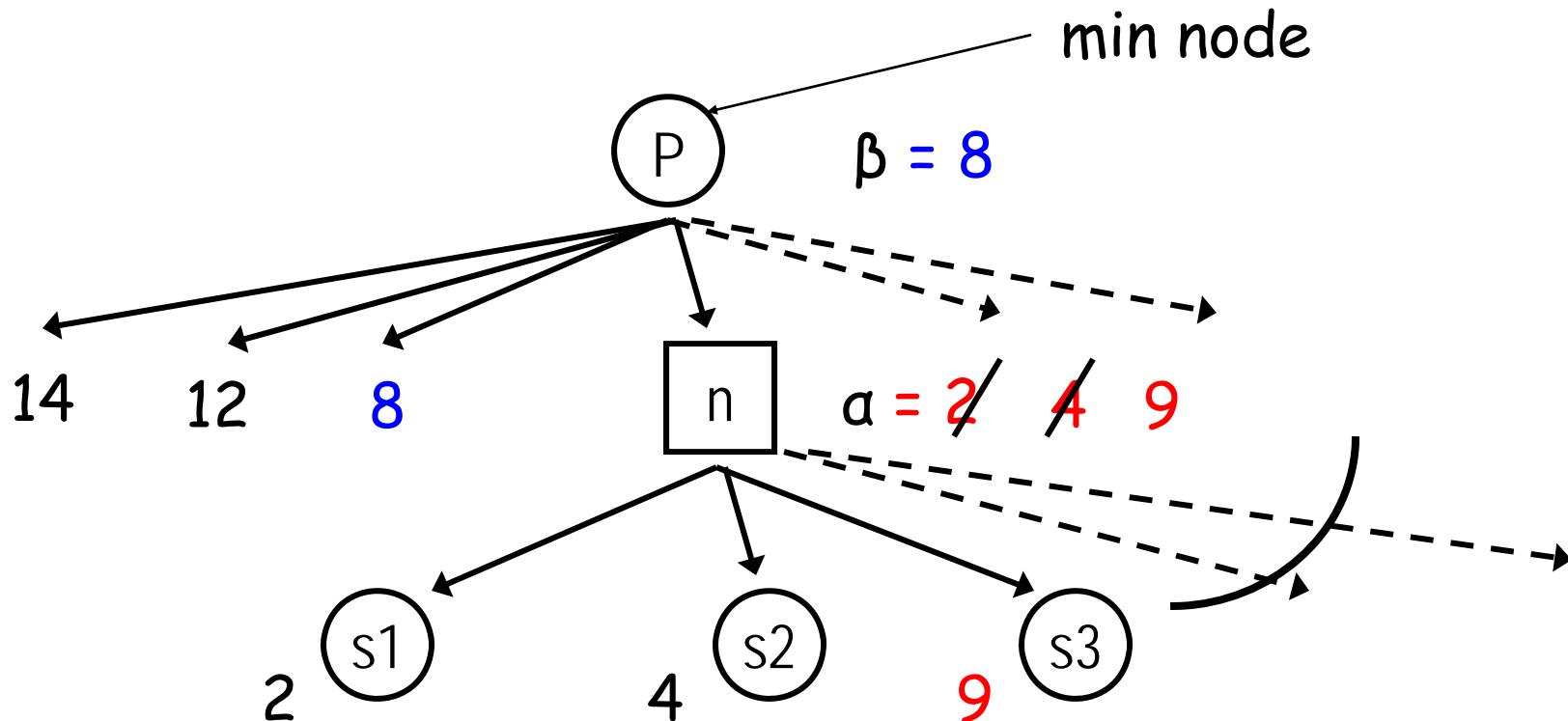
Cutting Max Nodes (Alpha Cuts)

- At a Max node n :
 - Let β be the lowest value of n 's siblings examined so far (siblings to the left of n that have already been searched)
 - Let α be the highest value of n 's children examined so far (changes as children examined)



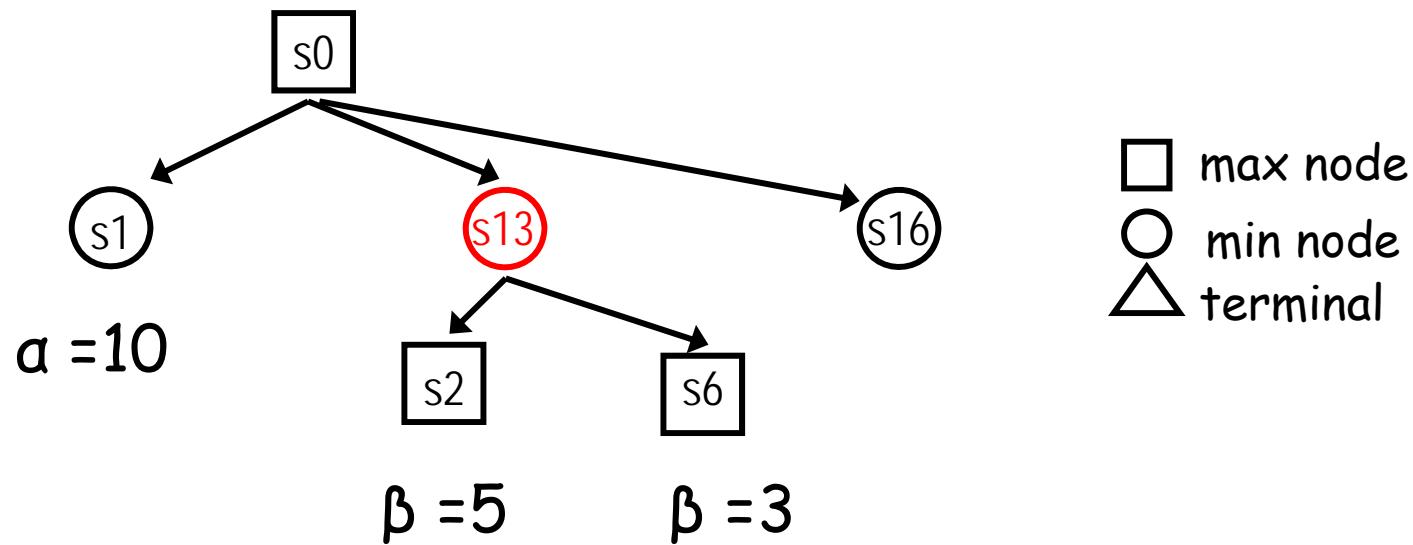
Cutting Max Nodes (Alpha Cuts)

- If α becomes $\geq \beta$ we can stop expanding the children of n
 - Min will never choose to move from n 's parent to n since it would choose one of n 's lower valued siblings first.



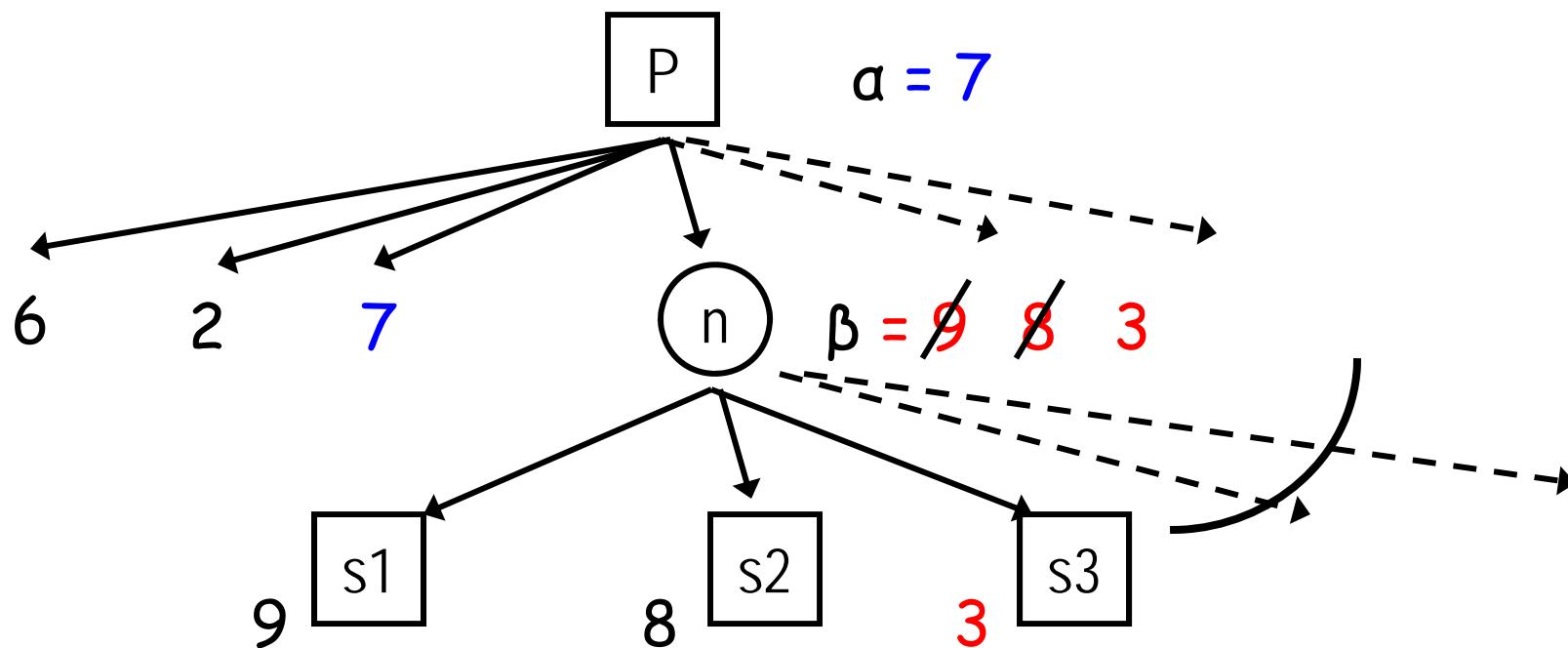
Cutting Min Nodes (Beta Cuts)

- At a Min node n :
 - Let β be the lowest value of n 's children examined so far (changes as children examined)
 - Let a be the highest value of n 's sibling's examined so far (fixed when evaluating n)



Cutting Min Nodes (Beta Cuts)

- If β becomes $\leq \alpha$ we can stop expanding the children of n .
 - Max will never choose to move from n 's parent to n since it would choose one of n 's higher value siblings first.



Implementing Alpha-Beta Pruning

```
AlphaBeta(n,Player,alpha,beta) //return Utility of state
If n is TERMINAL
    return U(n) //Return terminal states utility
ChildList = n.Successors(Player)
If Player == MAX
    for c in ChildList
        alpha = max(alpha, AlphaBeta(c,MIN,alpha,beta))
        If beta <= alpha
            break
    return alpha
Else //Player == MIN
    for c in ChildList
        beta = min(beta, AlphaBeta(c,MAX,alpha,beta))
        If beta <= alpha
            break
    return beta
```

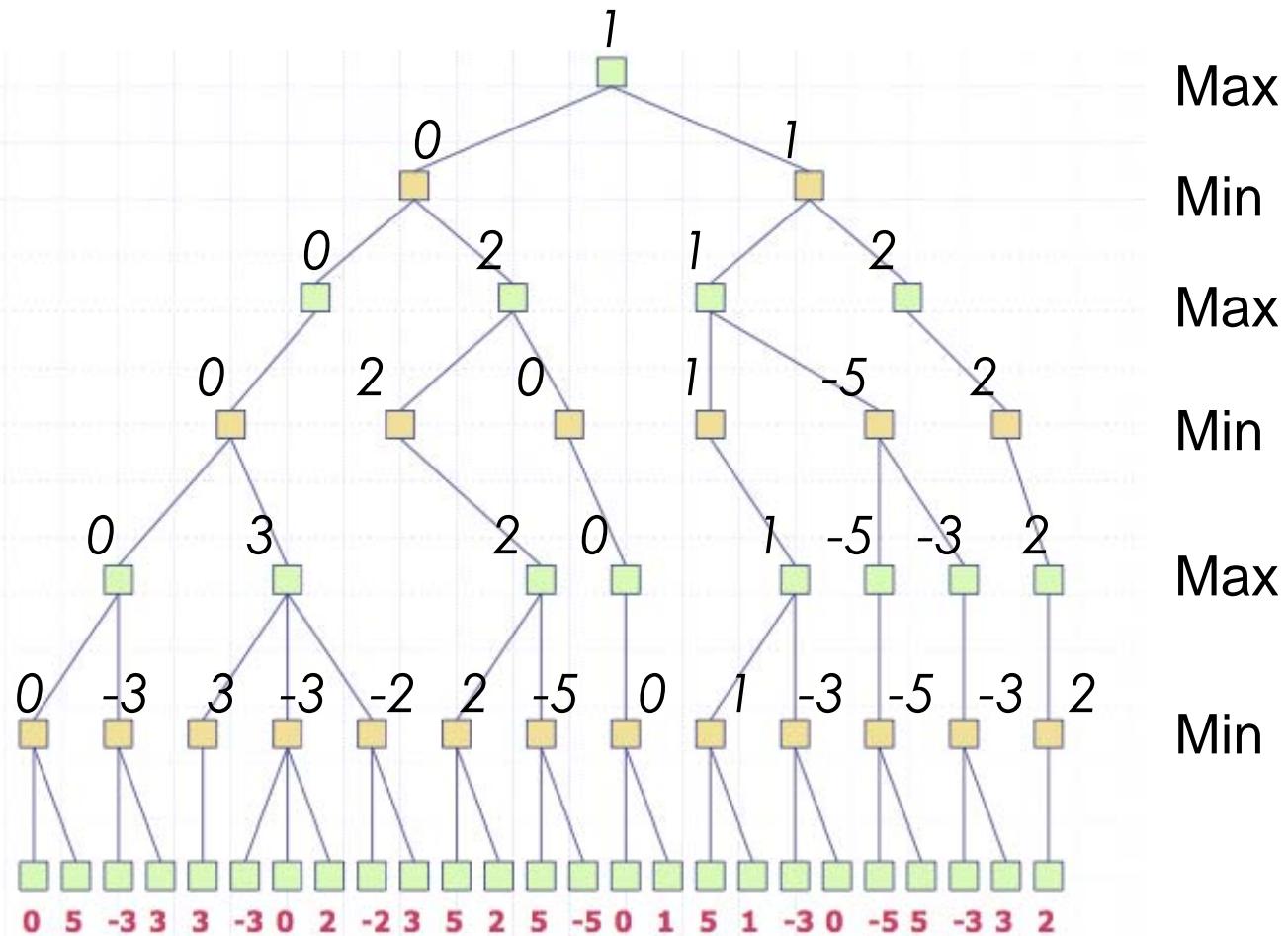
Implementing Alpha-Beta Pruning

Initial call

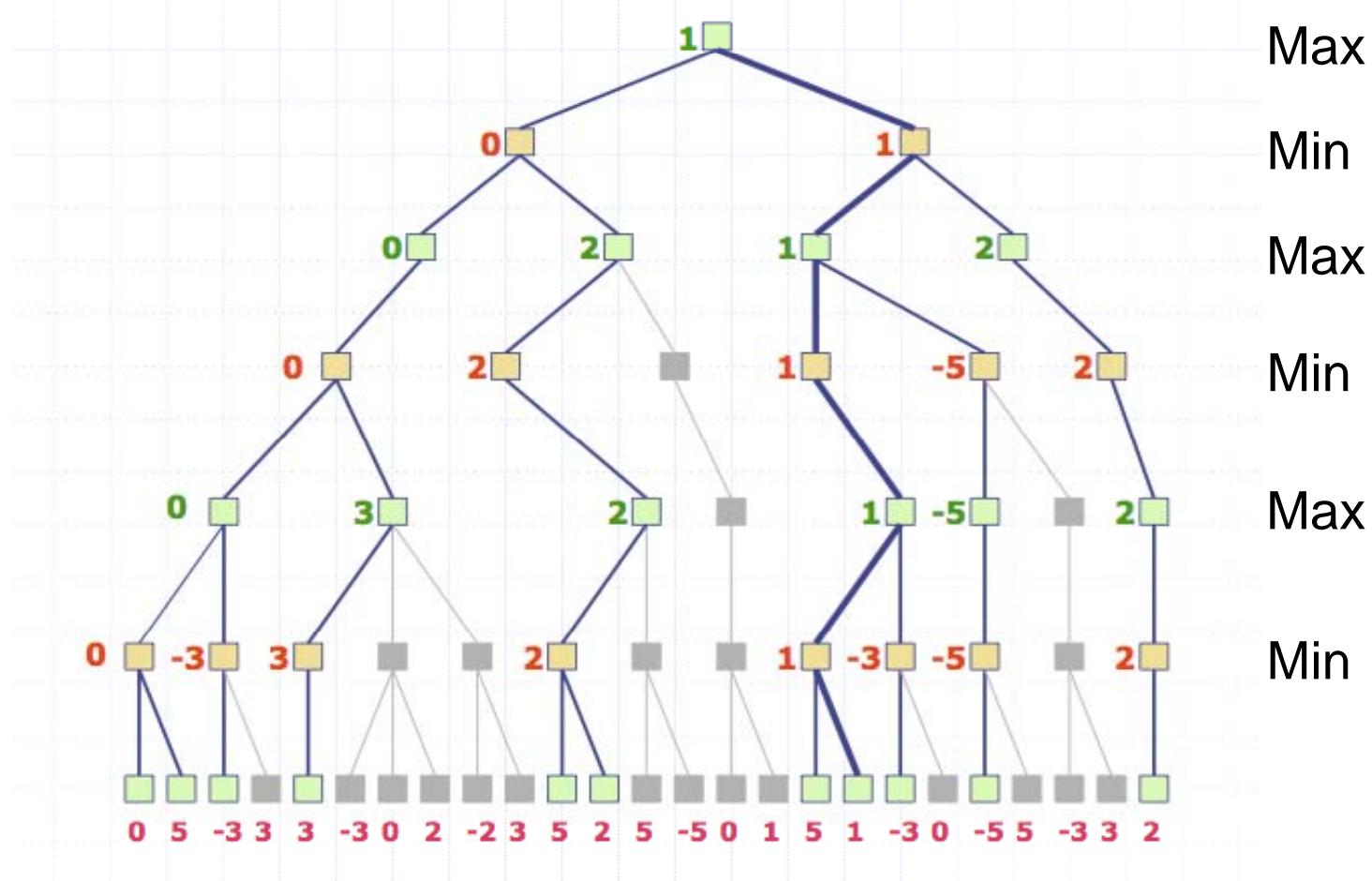
```
AlphaBeta(START_NODE, Player, -infinity, +infinity)
```

Example

Which computations could we have avoided here? Assuming we expand nodes left to right?



Example



Effectiveness of Alpha-Beta Pruning

- With no pruning, you have to explore $O(b^d)$ nodes, which makes the run time of a search with pruning the same as plain Minimax.
- If, however, the move ordering for the search is optimal (meaning the best moves are searched first), the number of nodes we need to search using alpha beta pruning is $O(b^{d/2})$. That means you can, in theory, search twice as deep!
- In Deep Blue, they found that alpha beta pruning meant the average branching factor at each node was about 6 instead of 35.

Rational Opponents

- May want to compute your full strategy ahead of time.
 - you must store “decisions” for each node you can reach by playing optimally
 - if your opponent has unique rational choices, this is a single branch through game tree
 - if there are “ties”, opponent could choose any one of the “tied” moves: must store strategy for each subtree
 - In general space is an issue.
 - Alternatively you compute your next move a fresh at each stage.

Practical Matters

- All “real” games are too large to enumerate tree
 - e.g., chess branching factor is roughly 35
 - Depth 10 tree: $2,700,000,000,000,000$ nodes
 - Even alpha-beta pruning won’t help here!
- We must limit depth of search tree
 - Can’t expand all the way to terminal nodes
 - We must make **heuristic estimates** about the values of the (non-terminal) states at the leaves of the tree
 - These heuristics are often called **evaluation function**
 - evaluation functions are often learned

Heuristics in Games

- Example for tic tac toe: $h(n) = [\# \text{ of 3 lengths that are left open for player A}] - [\# \text{ of 3 lengths that are left open for player B}]$.
- Alan Turing's function for chess: $h(n) = A(n)/B(n)$ where $A(n)$ is the sum of the point value for player A's pieces and $B(n)$ is the sum for player B.
- Most evaluation functions are specified as a weighted sum of features: $h(n) = w_1 * \text{feature}_1(n) + w_2 * \text{feature}_2(n) + \dots + w_i * \text{feature}_i(n)$.
- Deep Blue used about 6000 features in its evaluation function.

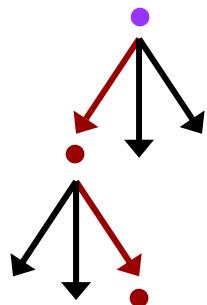
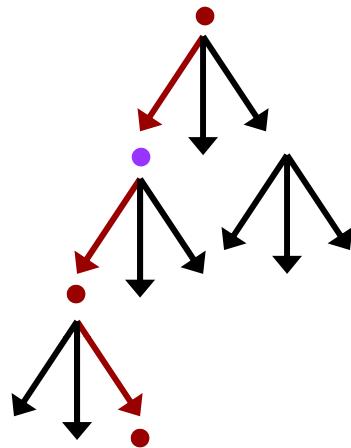
Heuristics in Games

- Think of a few games and suggest some heuristics for estimating the “goodness” of a position
 - Chess?
 - Checkers?
 - Your favorite video game?

An Aside on Large Search Problems

- Issue: *inability to expand tree to terminal nodes is relevant even in standard search*
 - Often we can't expect A* to reach a goal by expanding *full frontier*
 - So we often *limit our look-ahead*, and make moves before we actually know the true path to the goal
 - Sometimes called **online** or **realtime** search
- In this case, we use the heuristic function not just to guide our search, but also to commit to moves we actually make
 - In general, guarantees of optimality are lost, but we reduce computational/memory expense dramatically

Realtime Search Graphically



1. We run A* (or our favorite search algorithm) until we are forced to make a move or run out of memory. Note: no leaves are goals yet.
2. We use evaluation function $f(n)$ to decide which path looks best (let's say it is the red one).
3. We take the first step along the best path (red), by actually making that move.
4. We restart search at the node we reach by making that move. (We may actually cache the results of the relevant part of first search tree if it's hanging around, as it would with A*).

Game Tree Search

- Chapter 5.1, 5.2, 5.3, 5.6 cover some of the material we cover here. Section 5.6 has an interesting overview of State-of-the-Art game playing programs.
- Section 5.5 extends the ideas to games with uncertainty (We won't cover that material but it makes for interesting reading).

Generalizing Search Problem

- So far: our search problems have assumed agent has complete control of environment
 - State does not change unless the agent (robot) changes it.
 - All we need to compute is a single path to a goal state.
- Assumption not always reasonable
 - Stochastic environment (e.g., the weather, traffic accidents).
 - Other agents whose interests conflict with yours
 - Search can find a path to a goal state, but the actions might not lead you to the goal as the state can be changed by other agents (nature or other intelligent agents)

Generalizing Search Problem

- We need to generalize our view of search to handle state changes that are not in the control of the agent.
- One generalization yields game tree search
 - Agent and some other agents.
 - The other agents are acting to maximize their profits
 - this might not have a positive effect on your profits.

General Games

- What makes something a game?
 - There are two (or more) agents making changes to the world (the state)
 - Each agent has their own interests
 - e.g., each agent has a different goal; or assigns different costs to different paths/states
 - Each agent tries to alter the world so as to best benefit itself.

General Games

- What makes games hard?
 - How you should play depends on how you think the other person will play; but how they play depends on how they think you will play; so how you should play depends on how you think they think you will play; but how they play should depend on how they think you think they think you will play; ...

Properties of Games considered here

- Zero-sum games: Fully competitive
 - Competitive: if one play wins, the others lose; e.g. Poker – you win what the other player lose
 - Games can also be cooperative: some outcomes are preferred by both of us, or at least our values aren't diametrically opposed
- Deterministic: no chance involved
 - (no dice, or random deals of cards, or coin flips, etc.)
- Perfect information (all aspects of the state are fully observable, e.g., no hidden cards)

Our Focus: Two-Player Zero-Sum Games

- Fully competitive two player games
 - If you win, the other player (opponent) loses
 - Zero-sum means the sum of your and your opponent's payoff is zero--any thing you gain come at your opponent's cost (and vice-versa).
 - Key insight: How you act depends on how the other agent acts (or how you think they will act)
 - and vice versa (if your opponent acts rational)
- Examples of two-person zero-sum games:
 - Chess, checkers, tic-tac-toe, backgammon, go, Doom, "find the last parking space"
- Most of the ideas extend to multiplayer zero-sum games (cf. Chapter 5.2.2)

Game 1: Rock, Paper, Scissors

- Scissors cut paper, paper covers rock, rock smashes scissors
- Represented as a matrix: Player I chooses a row, Player II chooses a column
- Payoff to each player in each cell (P.I / P.II)
 - 1: win, 0: tie, -1: loss
 - so it's zero-sum

		Player II		
		R	P	S
Player I	R	0/0	-1/1	1/-1
	P	1/-1	0/0	-1/1
	S	-1/1	1/-1	0/0

Game 2: Prisoner's Dilemma

- Two prisoner's in separate cells, sheriff doesn't have enough evidence to convict them. They agree ahead of time to both deny the crime (they will **cooperate**).
 - If one defects (i.e., confesses) and the other doesn't
 - confessor goes free
 - other sentenced to 4 years
 - If both defect (confess)
 - both sentenced to 3 years
 - If both cooperate (neither confesses)
 - both sentenced to 1 year on minor charge
- Payoff: 4 minus sentence

	Coop	Def
Coop	3/3	0/4
Def	4/0	1/1

Extensive Form Two-Player Zero-Sum Games

- Key point of previous games: what you should do depends on what other guy does
- But previous games are simple "one shot" games
 - single move each
 - in game theory: [strategic or normal form games](#)
- Many games extend over multiple moves
 - turn-taking: players act alternatively
 - e.g., chess, checkers, etc.
 - in game theory: [extensive form games](#)
- We'll focus on the extensive form
 - that's where the computational questions emerge

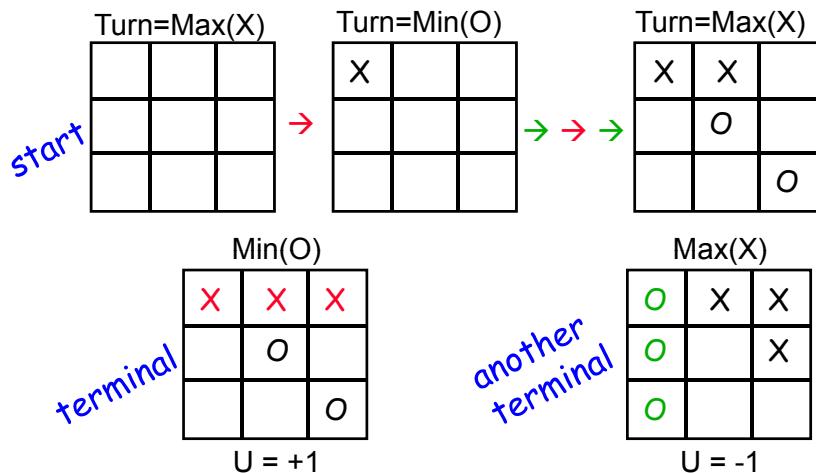
Two-Player Zero-Sum Game – Definition

- Two **players** A (Max) and B (Min)
- Set of **positions** P (states of the game)
- A **starting position** $s \in P$ (where game begins)
- Terminal positions** $T \subseteq P$ (where game can end)
- Set of directed edges E_A between states (A's **moves**)
- set of directed edges E_B between states (B's **moves**)
- Utility or payoff function** $U : T \rightarrow \mathbb{R}$ (how good is each terminal state for player A)
 - Why don't we need a utility function for B?

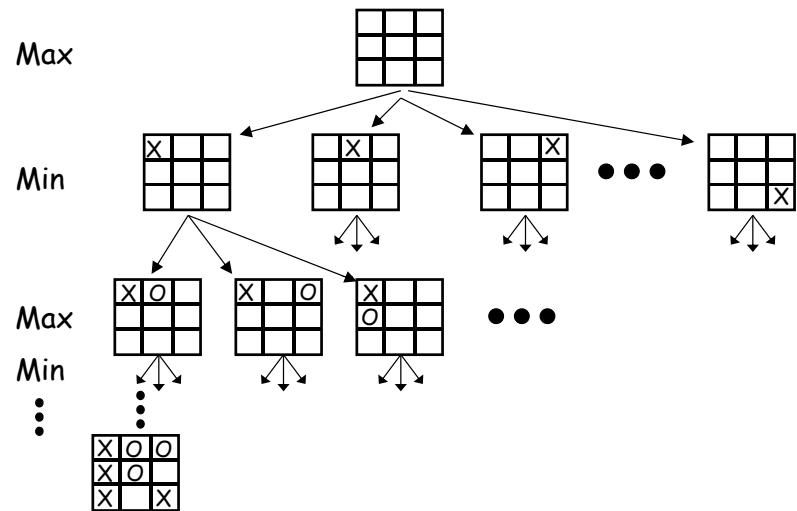
Two-Player Zero-Sum Game – Intuition

- Players alternate moves (starting with Max)
 - Game ends when some terminal $t \in T$ is reached
- A game **state**: a state-player pair
 - Tells us what state we're in and whose move it is
- Utility function and terminals replace goals
 - Max wants to maximize the terminal payoff
 - Min wants to minimize the terminal payoff
- Think of it as:
 - Max gets $U(t)$, Min gets $-U(t)$ for terminal node t
 - This is why it's called zero (or constant) sum

Tic Tac Toe States



Tic Tac Toe Game Tree



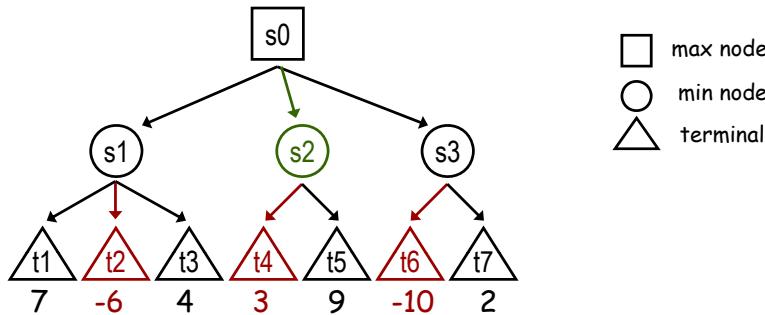
Game Tree

- Game tree looks like a search tree
 - Layers reflect alternating moves between **A** and **B**
 - The search tree in game playing is a subtree of the game tree
- Player **A** doesn't decide where to go alone
 - After **A** moves to a state, **B** decides which of the states children to move to
- Thus **A** must have a **strategy**
 - Must know what to do for each possible move of **B**
 - One sequence of moves will not suffice: "What to do" will depend on how **B** will play
- What is a reasonable strategy?

Minimax Strategy

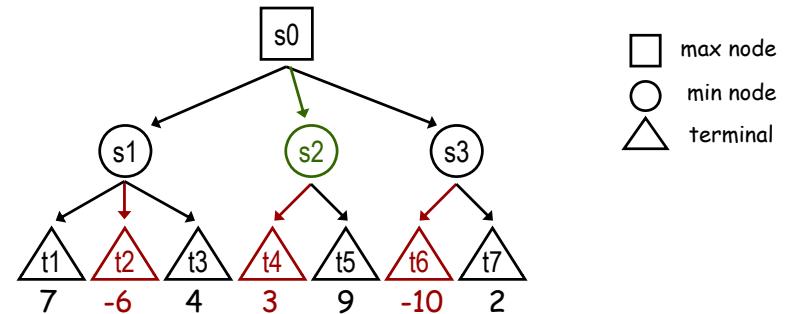
- Assume that the other player will always play their best move,
 - you always play a move that will minimize the payoff that could be gained by the other player.
 - By minimizing the other player's payoff you maximize yours.
- If however you know that Min will play poorly in some circumstances, there might be a better strategy than MiniMax (i.e., a strategy that gives you a better payoff).
- But in the absence of that knowledge minimax "plays it safe"

Minimax Strategy payoffs



The terminal nodes have utilities.
But we can compute a "utility" for the non-terminal states, by assuming both players always play their best move.

Minimax Strategy – Intuitions



If Max goes to s_1 , Min goes to t_2 , $U(s_1) = \min\{U(t_1), U(t_2)\} = -6$
If Max goes to s_2 , Min goes to t_4 , $U(s_2) = \min\{U(t_4), U(t_5)\} = 3$
If Max goes to s_3 , Min goes to t_6 , $U(s_3) = \min\{U(t_6), U(t_7)\} = -10$
So Max goes to s_2 : so $U(s_0) = \max\{U(s_1), U(s_2), U(s_3)\} = 3$

Minimax Strategy

- Build full game tree (all leaves are terminals)
 - Root is start state, edges are possible moves, etc.
 - Label terminal nodes with utilities
- Back values up the tree
 - $U(t)$ is defined for all terminals (part of input)
 - $U(n) = \min \{U(c) : c \text{ is a child of } n\}$ if n is a Min node
 - $U(n) = \max \{U(c) : c \text{ is a child of } n\}$ if n is a Max node

Minimax Strategy

- The values labeling each state are the values that Max will achieve in that state if both Max and Min play their best moves.
 - Max plays a move to change the state to the highest valued min child.
 - Min plays a move to change the state to the lowest valued max child.
- If Min plays poorly, Max could do better, but never worse.
 - If Max, however knows that Min will play poorly, there might be a better strategy of play for Max than Minimax.

Depth-First Implementation of Minimax

- Building the entire game tree and backing up values gives each player their strategy.
- However, the game tree is exponential in size.
- Furthermore, as we will see later it is not necessary to know all of the tree.
- To solve these problems we find a **depth-first** implementation of minimax.

We run the depth-first search after each move to compute what is the next move for the **MAX** player. (We could do the same for the **MIN** player).

- This avoids explicitly representing the exponentially sized game tree: we just compute each move as it is needed.

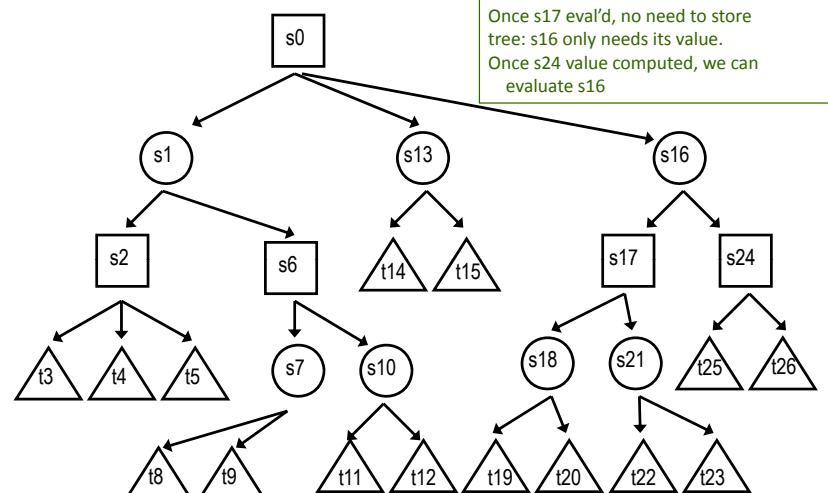
Depth-First Implementation of Minimax

```
DFMinimax(n, Player) //return Utility of state n given that  
//Player is MIN or MAX  
If n is TERMINAL  
    Return U(n) //Return terminal states utility  
    //(U is specified as part of game)  
    //Apply Player's moves to get  
    //successor states.  
ChildList = n.Successors(Player)  
If Player == MIN  
    return minimum of DFMinimax(c, MAX) over c ∈ ChildList  
Else //Player is MAX  
    return maximum of DFMinimax(c, MIN) over c ∈ ChildList
```

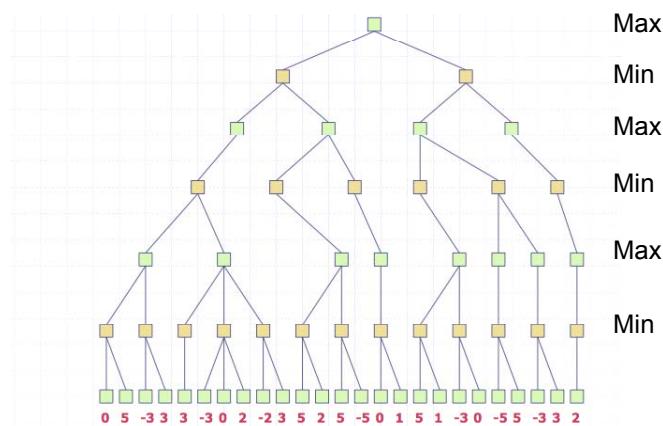
Depth-First Implementation of Minimax

- Notice that the game tree has to have finite depth for this to work
- Advantage of DF implementation: space efficient
- Minimax will expand $O(b^d)$ states, which is both a **BEST** and **WORSE** case scenario.
 - We must traverse the entire search tree to evaluate all options
 - We can't be lucky as in regular search and find a path to a goal before searching the entire tree.

Visualization of Depth-First Minimax



Example



Fahiem Bacchus, CSC384 Introduction to Artificial Intelligence, University of Toronto

25

Pruning

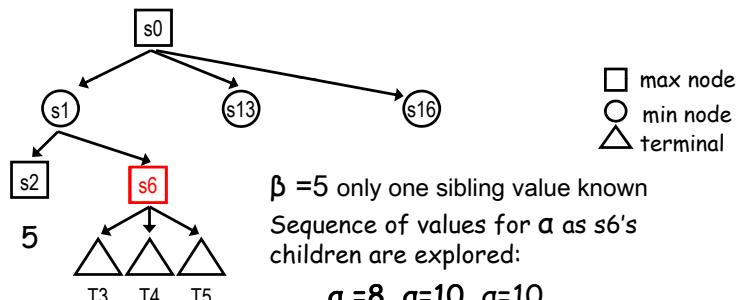
- It is not necessary to examine entire tree to make correct Minimax decision
- Assume depth-first generation of tree
 - After generating value for only *some* of n 's children we can prove that we'll never reach n in a Minmax strategy.
 - So we needn't generate or evaluate any further children of n !
- Two types of pruning (cuts):
 - pruning of max nodes (α -cuts)
 - pruning of min nodes (β -cuts)

Fahiem Bacchus, CSC384 Introduction to Artificial Intelligence, University of Toronto

26

Cutting Max Nodes (Alpha Cuts)

- At a Max node n :
- Let β be the lowest value of n 's siblings examined so far (siblings to the left of n that have already been searched)
- Let α be the highest value of n 's children examined so far (changes as children examined)

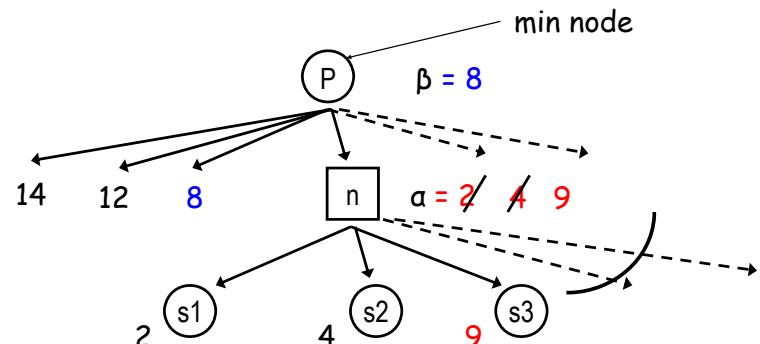


Fahiem Bacchus, CSC384 Introduction to Artificial Intelligence, University of Toronto

27

Cutting Max Nodes (Alpha Cuts)

- If α becomes $\geq \beta$ we can stop expanding the children of n
 - Min will never choose to move from n 's parent to n since it would choose one of n 's lower valued siblings first.

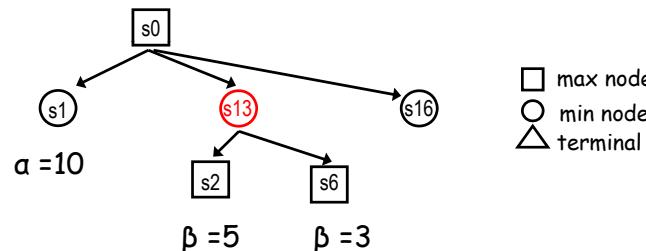


Fahiem Bacchus, CSC384 Introduction to Artificial Intelligence, University of Toronto

28

Cutting Min Nodes (Beta Cuts)

- At a Min node n :
 - Let β be the lowest value of n 's children examined so far (changes as children examined)
 - Let α be the highest value of n 's sibling's examined so far (fixed when evaluating n)

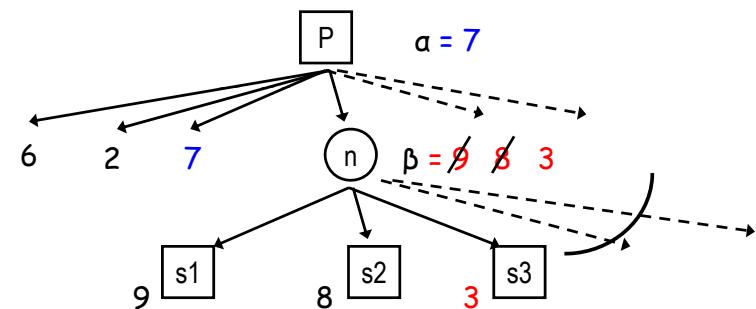


Fahiem Bacchus, CSC384 Introduction to Artificial Intelligence, University of Toronto

29

Cutting Min Nodes (Beta Cuts)

- If β becomes $\leq \alpha$ we can stop expanding the children of n .
 - Max will never choose to move from n 's parent to n since it would choose one of n 's higher value siblings first.



Fahiem Bacchus, CSC384 Introduction to Artificial Intelligence, University of Toronto

30

Implementing Alpha-Beta Pruning

```
AlphaBeta(n,Player,alpha,beta) //return Utility of state
If n is TERMINAL
  return U(n) //Return terminal states utility
ChildList = n.Successors(Player)
If Player == MAX
  for c in ChildList
    alpha = max(alpha, AlphaBeta(c,MIN,alpha,beta))
    If beta <= alpha
      break
  return alpha
Else //Player == MIN
  for c in ChildList
    beta = min(beta, AlphaBeta(c,MAX,alpha,beta))
    If beta <= alpha
      break
  return beta
```

Fahiem Bacchus, CSC384 Introduction to Artificial Intelligence, University of Toronto

31

Implementing Alpha-Beta Pruning

Initial call

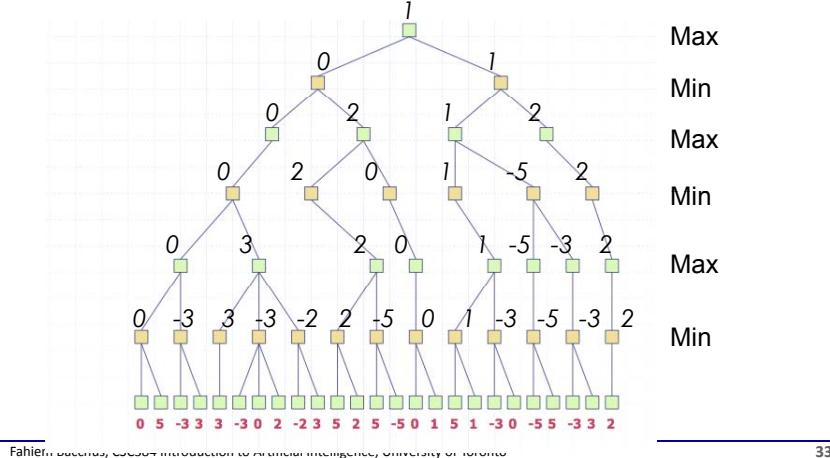
AlphaBeta(START_NODE,Player,-infinity,+infinity)

32

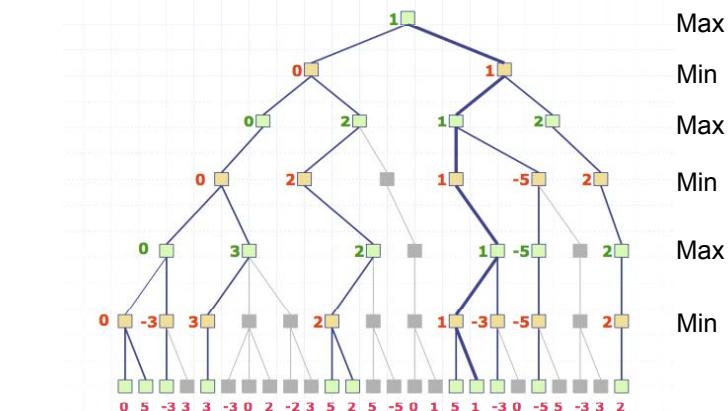
Fahiem Bacchus, CSC384 Introduction to Artificial Intelligence, University of Toronto

Example

Which computations could we have avoided here? Assuming we expand nodes left to right?



Example



Effectiveness of Alpha-Beta Pruning

- With no pruning, you have to explore $O(b^d)$ nodes, which makes the run time of a search with pruning the same as plain Minimax.
- If, however, the move ordering for the search is optimal (meaning the best moves are searched first), the number of nodes we need to search using alpha beta pruning is $O(b^{d/2})$. That means you can, in theory, search twice as deep!
- In Deep Blue, they found that alpha beta pruning meant the average branching factor at each node was about 6 instead of 35.

Rational Opponents

- May want to compute your full strategy ahead of time.
 - you must store “decisions” for each node you can reach by playing optimally
 - if your opponent has unique rational choices, this is a single branch through game tree
 - if there are “ties”, opponent could choose any one of the “tied” moves: must store strategy for each subtree
 - In general space is an issue.
 - Alternatively you compute your next move a fresh at each stage.

Practical Matters

- All “real” games are too large to enumerate tree
 - e.g., chess branching factor is roughly 35
 - Depth 10 tree: $2,700,000,000,000,000$ nodes
 - Even alpha-beta pruning won’t help here!
- We must limit depth of search tree
 - Can’t expand all the way to terminal nodes
 - We must make **heuristic estimates** about the values of the (non-terminal) states at the leaves of the tree
 - These heuristics are often called **evaluation function**
 - evaluation functions are often learned

Heuristics in Games

- Example for tic tac toe: $h(n) = [\# \text{ of 3 lengths that are left open for player A}] - [\# \text{ of 3 lengths that are left open for player B}]$.
- Alan Turing’s function for chess: $h(n) = A(n)/B(n)$ where $A(n)$ is the sum of the point value for player A’s pieces and $B(n)$ is the sum for player B.
- Most evaluation functions are specified as a weighted sum of features: $h(n) = w_1*\text{feature}_1(n) + w_2*\text{feature}_2(n) + \dots w_i*\text{feature}_i(n)$.
- Deep Blue used about 6000 features in its evaluation function.

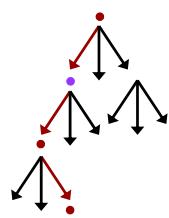
Heuristics in Games

- Think of a few games and suggest some heuristics for estimating the “goodness” of a position
 - Chess?
 - Checkers?
 - Your favorite video game?

An Aside on Large Search Problems

- Issue: inability to expand tree to terminal nodes is relevant even in standard search
 - Often we can’t expect A* to reach a goal by expanding full frontier
 - So we often limit our look-ahead, and make moves before we actually know the true path to the goal
 - Sometimes called **online** or **realtime** search
- In this case, we use the heuristic function not just to guide our search, but also to commit to moves we actually make
 - In general, guarantees of optimality are lost, but we reduce computational/memory expense dramatically

Realtime Search Graphically



1. We run A* (or our favorite search algorithm) until we are forced to make a move or run out of memory. Note: no leaves are goals yet.
2. We use evaluation function $f(n)$ to decide which path looks best (let's say it is the red one).
3. We take the first step along the best path (red), by actually making that move.
4. We restart search at the node we reach by making that move. (We may actually cache the results of the relevant part of first search tree if it's hanging around, as it would with A*).

CSC384h: Intro to Artificial Intelligence

▶ Knowledge Representation

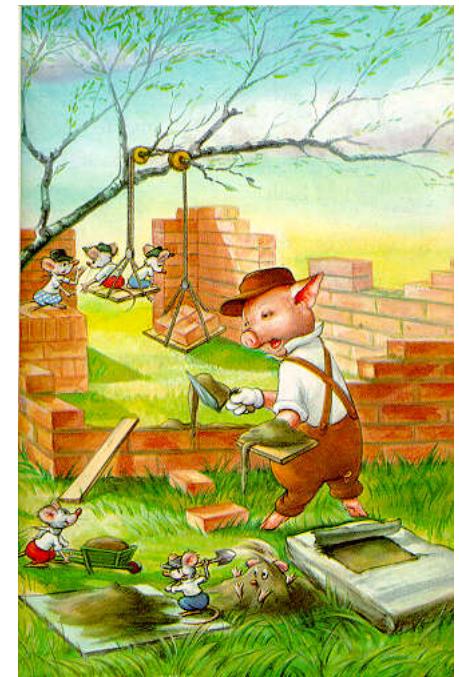
- ▶ This material is covered in chapters 7—10 of the text.
- ▶ Chapter 7 provides a useful motivation for logic, and an introduction to some basic ideas. It also introduces propositional logic, which is a good background for first-order logic.
- ▶ What we cover here is mainly covered in Chapters 8 and 9. However, Chapter 8 contains some additional useful examples of how first-order knowledge bases can be constructed. Chapter 9 covers forward and backward chaining mechanisms for inference, while here we concentrate on resolution.
- ▶ Chapter 10 covers some of the additional notions that have to be dealt with when using knowledge representation in AI.

Knowledge Representation

- ▶ Consider the task of understanding a simple story.
- ▶ How do we test understanding?
- ▶ Not easy, but understanding at least entails some ability to answer simple questions about the story.

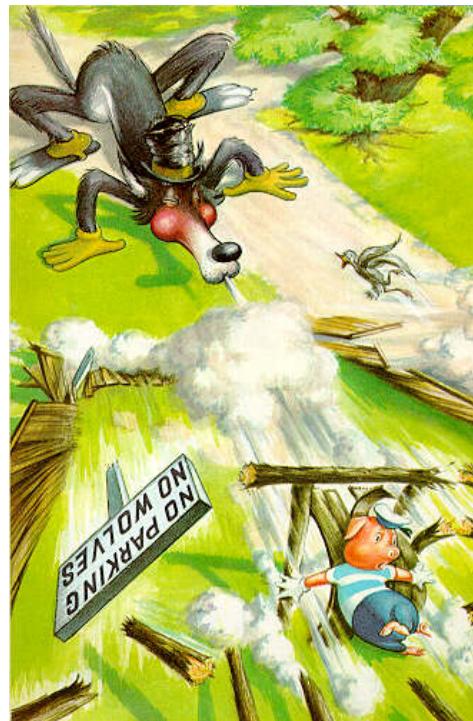
Example.

- ▶ Three little pigs



Example.

- ▶ Three little pigs



Example.

- ▶ Why couldn't the wolf blow down the house made of bricks?
- ▶ What background knowledge are we applying to come to that conclusion?
 - ▶ Brick structures are stronger than straw and stick structures.
 - ▶ Objects, like the wolf, have physical limitations. The wolf can only blow so hard.

Why Knowledge Representation?

- ▶ Large amounts of knowledge are used to understand the world around us, and to communicate with others.
- ▶ We also have to be able to reason with that knowledge.
 - ▶ Our knowledge won't be about the blowing ability of wolfs in particular, it is about physical limits of objects in general.
 - ▶ We have to employ reasoning to make conclusions about the wolf.
 - ▶ More generally, reasoning provides an exponential or more compression in the knowledge we need to store. I.e., without reasoning we would have to store a infeasible amount of information: e.g., Elephants can't fit into teacups.

Logical Representations

- ▶ AI typically employs logical representations of knowledge.
- ▶ Logical representations useful for a number of reasons:

Logical Representations

- ▶ They are mathematically precise, thus we can analyze their limitations, their properties, the complexity of inference etc.
- ▶ They are formal languages, thus computer programs can manipulate sentences in the language.
- ▶ They come with both a formal **syntax** and a formal **semantics**.
- ▶ Typically, have well developed **proof theories**: formal procedures for reasoning at the syntactic level (achieved by manipulating sentences).

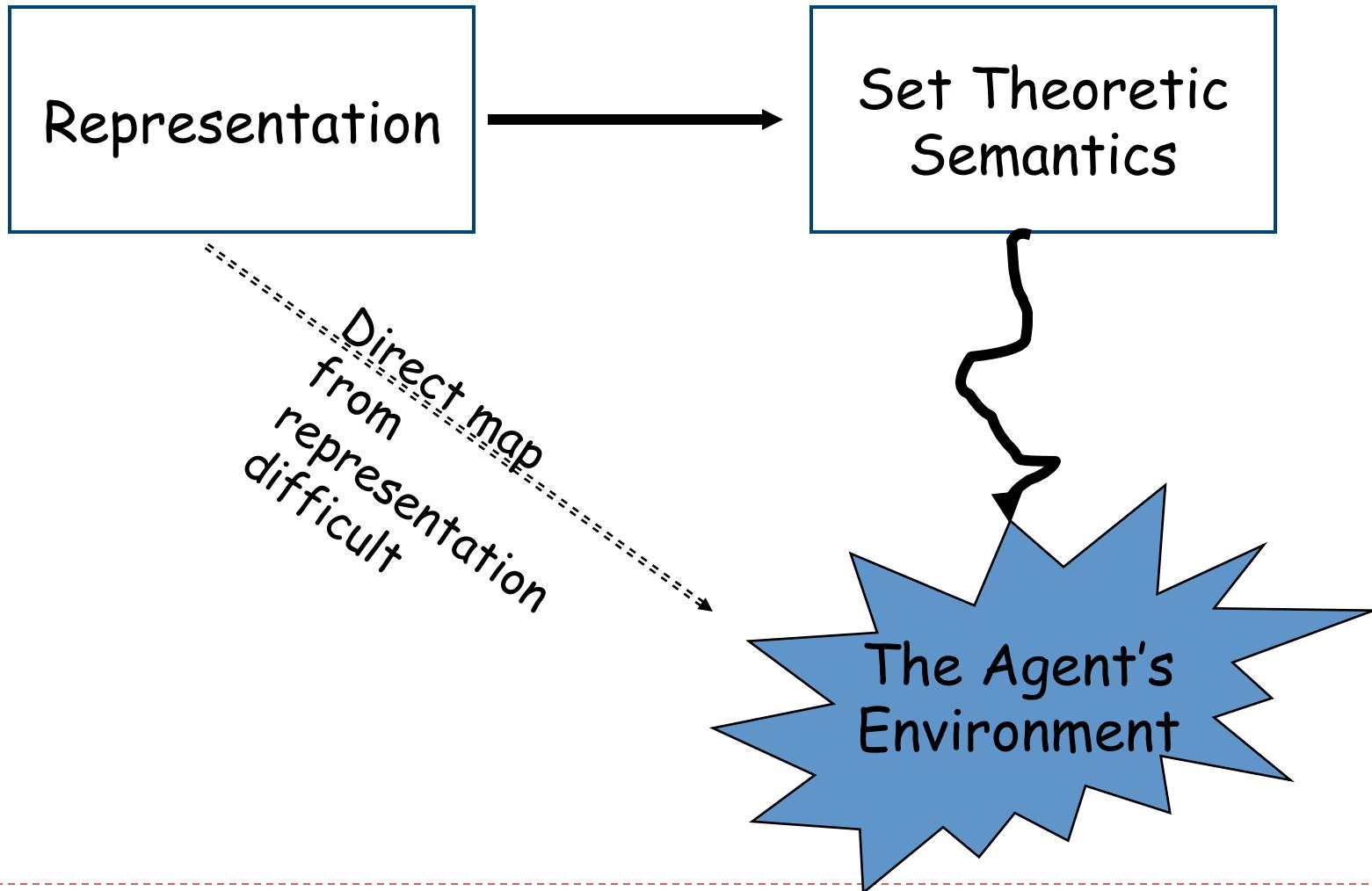
Set theoretic semantics

- ▶ Suppose our knowledge is represented in our program by some collection of data structures. We can think of these as a collection of **strings (sentences)**.
 - ▶ We want a clear mapping from this set of sentences to features of the environment. What are sentences asserting about environment?
-
- ▶ In other words, we want to be able to provide an intuitive interpretation of any piece of our representation.
 - ▶ Similar in spirit to having an intuitive understanding of what individual statements in a program mean. It does not mean that it is easy to understand the whole, but it provides the means to understand the whole by understanding the parts.

Set theoretic semantics

- ▶ Set theoretic semantics facilitates both goals.
 - ▶ It is a formal characterization, and it can be used to prove a wide range of properties of the representation.
 - ▶ It maps arbitrarily complex sentences of the logic down into intuitive assertions about the real world.
 - ▶ It is based on notions that are very close to how we think about the real world. Thus it provides the bridge from the syntax to an intuitive understanding of what is being asserted.

Set theoretic semantics



Semantics Formal Details

- ▶ A set of **objects**. These are objects in the environment that are important for your application.
- ▶ Distinguished subsets of objects. **Properties**.
- ▶ Distinguished sets of tuples of objects. **Relations**.
- ▶ Distinguished functions mapping tuples of objects to objects. **Functions**.

Example

- ▶ Teaching CSC384, want to represent knowledge that would be useful for making the course a successful learning experience.

- ▶ **Objects:**
 - ▶ **students, subjects, assignments, numbers.**
- ▶ **Predicates:**
 - ▶ **difficult(subject), CSMajor(student).**
- ▶ **Relations:**
 - ▶ **handedIn(student, assignment)**
- ▶ **Functions:**
 - ▶ **Grade(student, assignment) → number**

First Order Logic

1. **Syntax:** A grammar specifying what are legal syntactic constructs of the representation.
2. **Semantics:** A formal mapping from syntactic constructs to set theoretic assertions.

First Order Syntax

Start with a set of primitive symbols.

1. *constant symbols.*
 2. *function symbols.*
 3. *predicate symbols (for predicates and relations).*
 4. *variables.*
-
- Each function and predicate symbol has a specific arity (determines the number of arguments it takes).

First Order Syntax—Building up.

- ▶ *terms* are used as names (perhaps complex nested names) for objects in the domain.
- ▶ *Terms* of the language are either:
 - ▶ a variable
 - ▶ a constant
 - ▶ an expression of the form $f(t_1, \dots t_k)$ where
 - ▶ (a) f is a function symbol;
 - ▶ (b) k is its arity;
 - ▶ (c) each t_i is a term
- ▶ 5 is a term—a symbol representing the number 5. John is a term—a symbol representing the person John.
- ▶ $+(5,5)$ is a term—a symbol representing the number 10.

First Order Syntax—Building up.

- ▶ **Note:** constants are the same as functions taking zero arguments.
- ▶ Terms denote objects (things in the world):
 - ▶ constants denote specific objects;
 - ▶ functions map tuples of objects to other objects
 - ▶ bill, jane, father(jane), father(father(jane))
 - ▶ X , father(X), hotel7, rating(hotel7), cost(hotel7)
 - ▶ Variables like X are not yet determined, but they will eventually denote particular objects.

First Order Syntax—Building up.

- ▶ Once we have terms we can build up *formulas*. Terms represent (denote) objects, *formulas* represent true/false assertions about these objects.
- ▶ We start with *atomic formulas* these are
 - ▶ expressions of the form $p(t_1, \dots t_k)$ where
 - ▶ (a) p is a predicate symbol;
 - ▶ (b) k is its arity;
 - ▶ (c) each t_i is a term

Semantic Intuition (formalized later).

- ▶ Atoms denote facts that can be true or false about the world
 - ▶ father_of(jane, bill), female(jane), system_down()
 - ▶ satisfied(client15), satisfied(C)
 - ▶ desires(client15,rome,week29), desires(X,Y,Z)
 - ▶ rating(hotel7, 4), cost(hotel7, 125)

First Order Syntax—Building up.

- ▶ Atomic formulas
- ▶ The negation (NOT) of a formula is a new formula
 - ▶ $\neg f$ ($\neg f$)
Asserts that f is false.
- ▶ The conjunction (AND) of a set of formulas is a formula.
 - ▶ $f_1 \wedge f_2 \wedge \dots \wedge f_n$ where each f_i is formula
Asserts that each formula f_i is true.

First Order Syntax—Building up.

- ▶ The disjunction (OR) of a set of formulas is a formula.
 - ▶ $f_1 \vee f_2 \vee \dots \vee f_n$ where each f_i is formula
 - Asserts that at least one formula f_i is true.
- ▶ Existential Quantification \exists .
 - ▶ $\exists X. f$ where X is a variable and f is a formula.
 - Asserts there is some object such that once X is bound to that object, f will be true.
- ▶ Universal Quantification \forall .
 - ▶ $\forall X. f$ where X is a variable and f is a formula.
 - Asserts that f is true for every object X can be bound to.

First Order Syntax—abbreviations.

- ▶ Implication:

- ▶ $f_1 \rightarrow f_2$

Take this to mean

- ▶ $\neg f_1 \vee f_2$.

Semantics.

- ▶ Formulas (syntax) can be built up recursively, and can become arbitrarily complex.
- ▶ Intuitively, there are various distinct formulas (viewed as strings) that really are asserting the same thing
 - ▶ $\forall X, Y. \text{elephant}(X) \wedge \text{teacup}(Y) \rightarrow \text{largerThan}(X, Y)$
 - ▶ $\forall X, Y. \text{teacup}(Y) \wedge \text{elephant}(X) \rightarrow \text{largerThan}(X, Y)$
- ▶ To capture this equivalence and to make sense of complex formulas we utilize the semantics.

Semantics.

- ▶ A formal mapping from formulas to semantic entities (individuals, sets and relations over individuals, functions over individuals).
- ▶ The mapping is mirrors the recursive structure of the syntax, so we can give any formula, no matter how complex a mapping to semantic entities.

Semantics—Formal Details

- ▶ First, we must fix the particular first-order language we are going to provide semantics for. The primitive symbols included in the syntax defines the particular language.
 $L(F,P,V)$
- ▶ F = set of function (and constant symbols)
 - ▶ Each symbol f in F has a particular arity.
- ▶ P = set of predicate and relation symbols.
 - ▶ Each symbol p in P has a particular arity.
- ▶ V = an infinite set of variables.

Semantics—Formal Details

- ▶ An interpretation (model) is a tuple $\langle D, \Phi, \Psi, v \rangle$
 - ▶ D is a non-empty set (domain of individuals)
 - ▶ Φ is a mapping: $\Phi(f) \rightarrow (D^k \rightarrow D)$
 - ▶ maps k-ary function symbol f , to a function from k-ary tuples of individuals to individuals.
 - ▶ Ψ is a mapping: $\Psi(p) \rightarrow (D^k \rightarrow \text{True/False})$
 - ▶ maps k-ary predicate symbol p , to an indicator function over k-ary tuples of individuals (a subset of D^k)
 - ▶ v is a variable assignment function. $v(X) = d \in D$ (it maps every variable to some individual)

Intuitions: Domain

- ▶ Domain D : $d \in D$ is an *individual*
- ▶ E.g., $\{ \underline{\text{craig}}, \underline{\text{jane}}, \underline{\text{grandhotel}}, \underline{\text{le-fleabag}}, \underline{\text{rome}}, \underline{\text{portofino}}, \underline{100}, \underline{110}, \underline{120} \dots \}$
- ▶ Underlined symbols denote domain individuals
(as opposed to symbols of the first-order language)
- ▶ Domains often infinite, but we'll use finite models to prime our intuitions

Intuitions: Φ

- ▶ $\Phi(f) \rightarrow (D^k \rightarrow D)$

Given k -ary function f , k individuals, what individual does $f(d_1, \dots, d_k)$ denote

- ▶ 0-ary functions (constants) are mapped to specific individuals in D .
 - ▶ $\Phi(\text{client17}) = \text{craig}$, $\Phi(\text{hotel5}) = \text{le-fleabag}$, $\Phi(\text{rome}) = \text{rome}$
- ▶ 1-ary functions are mapped to functions in $D \rightarrow D$
 - ▶ $\Phi(\text{minquality}) = f_{\text{minquality}}$:
 $f_{\text{minquality}}(\text{craig}) = \text{3stars}$
 - ▶ $\Phi(\text{rating}) = f_{\text{rating}}$:
 $f_{\text{rating}}(\text{grandhotel}) = \text{5stars}$
- ▶ 2-ary functions are mapped to functions from $D^2 \rightarrow D$
 - ▶ $\Phi(\text{distance}) = f_{\text{distance}}$:
 $f_{\text{distance}}(\text{toronto}, \text{sienna}) = \text{3256}$
- ▶ n -ary functions are mapped similarly.

Intuitions: Ψ

- ▶ $\Psi(p) \rightarrow (D^k \rightarrow \text{True/False})$
 - ▶ given k -ary predicate, k individuals, does the relation denoted by p hold of these? $\Psi(p)(\langle d_1, \dots, d_k \rangle) = \text{true?}$
- ▶ 0-ary predicates are mapped to true or false.
 $\Psi(\text{rainy}) = \text{True}$ $\Psi(\text{sunny}) = \text{False}$
- ▶ 1-ary predicates are mapped indicator functions of subsets of D .
 - ▶ $\Psi(\text{satisfied}) = p_{\text{satisfied}}$:
 $p_{\text{satisfied}}(\underline{\text{craig}}) = \text{True}$
 - ▶ $\Psi(\text{privatebeach}) = p_{\text{privatebeach}}$:
 $p_{\text{privatebeach}}(\underline{\text{le-fleabag}}) = \text{False}$
- ▶ 2-ary predicates are mapped to indicator functions over D^2
 - ▶ $\Psi(\text{location}) = p_{\text{location}}$: $p_{\text{location}}(\underline{\text{grandhotel}}, \underline{\text{rome}}) = \text{True}$
 $p_{\text{location}}(\underline{\text{grandhotel}}, \underline{\text{sienna}}) = \text{False}$
 - ▶ $\Psi(\text{available}) = p_{\text{available}}$:
 $p_{\text{available}}(\underline{\text{grandhotel}}, \underline{\text{week29}}) = \text{True}$
- ▶ n -ary predicates..

Intuitions: ν

- ▶ ν exists to take care of quantification. As we will see the exact mapping it specifies will not matter.
- ▶ Notation: $\nu[X/d]$ is a **new** variable assignment function.
 - ▶ Exactly like ν , except that it maps the variable X to the individual d .
 - ▶ Maps every other variable exactly like ν :
$$\nu(Y) = \nu[X/d](Y)$$

Semantics—Building up

Given language $L(F,P,V)$, and an interpretation

$$I = \langle D, \Phi, \Psi, v \rangle$$

- a) Constant c (0-ary function) denotes an individual
 $I(c) = \Phi(c) \in D$
- b) Variable X denotes an individual
 $I(X) = v(X) \in D$ (variable assignment function).
- c) Ground term $t = f(t_1, \dots, t_k)$ denotes an individual
 $I(t) = \Phi(f)(I(t_1), \dots, I(t_k)) \in D$

We recursively find the denotation of each term, then we apply the function denoted by f to get a new individual.

Hence terms always denote individuals under an interpretation I

Semantics—Building up

Formulas

- a) Ground atom $a = p(t_1, \dots, t_k)$ has truth value
 $I(a) = \Psi(p)(I(t_1), \dots, I(t_k)) \in \{ \text{True}, \text{False} \}$

We recursively find the individuals denoted by the t_i , then we check to see if this tuple of individuals is in the relation denoted by p .

Semantics—Building up

Formulas

- b) Negated formulas $\neg f$ has truth value

$I(\neg f) = \text{True if } I(f) = \text{False}$

$I(\neg f) = \text{False if } I(f) = \text{True}$

- c) And formulas $f_1 \wedge f_2 \wedge \dots \wedge f_n$ have truth value

$I(f_1 \wedge f_2 \wedge \dots \wedge f_n) = \text{True if every } I(f_i) = \text{True.}$

$I(f_1 \wedge f_2 \wedge \dots \wedge f_n) = \text{False otherwise.}$

- d) Or formulas $f_1 \vee f_2 \vee \dots \vee f_n$ have truth value

$I(f_1 \vee f_2 \vee \dots \vee f_n) = \text{True if any } I(f_i) = \text{True.}$

$I(f_1 \vee f_2 \vee \dots \vee f_n) = \text{False otherwise.}$

Semantics—Building up

Formulas

e) Existential formulas $\exists X. f$ have truth value

$I(\exists X. f) = \text{True}$ if there exists a $d \in D$ such that

$I'(f) = \text{True}$

where $I' = \langle D, \Phi, \Psi, v[X/d] \rangle$

False otherwise.

I' is just like I except that its variable assignment function now maps X to d . " d " is the individual of which " f " is true.

Semantics—Building up

Formulas

f) Universal formulas $\forall X.f$ have truth value

$$I(\forall X.f) = \text{True if for all } d \in D$$

$$I'(f) = \text{True}$$

$$\text{where } I' = \langle D, \Phi, \Psi, v[X/d] \rangle$$

False otherwise.

Now "f" must be true of every individual "d".

Hence formulas are always either True or False under an interpretation I

Example

$D = \{\underline{bob}, \underline{jack}, \underline{fred}\}$

happy is true of all objects.

$I(\forall X.\text{happy}(X))$

1. $\Psi(\text{happy})(v[X/bob](X)) = \Psi(\text{happy})(\text{bob}) = \text{True}$

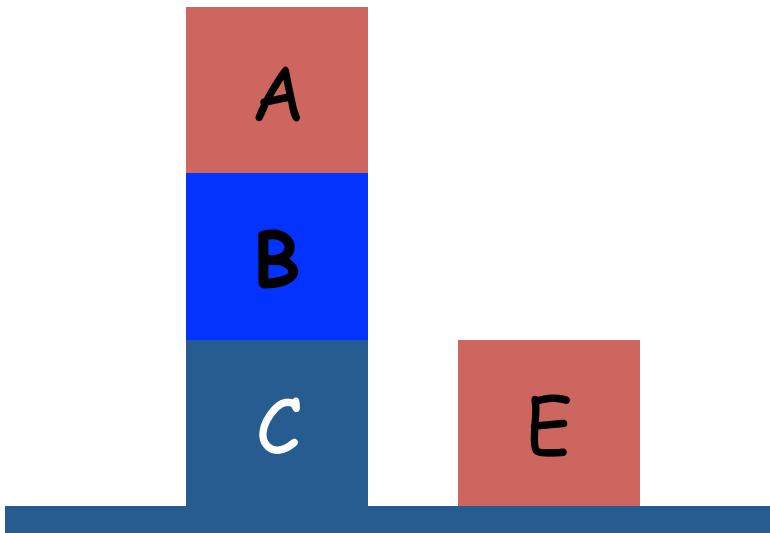
2. $\Psi(\text{happy})(v[X/jack](X)) = \Psi(\text{happy})(\text{jack}) = \text{True}$

3. $\Psi(\text{happy})(v[X/fred](X)) = \Psi(\text{happy})(\text{fred}) = \text{True}$

Therefore $I(\forall X.\text{happy}(X)) = \text{True}.$

Models—Examples.

Environment



Language (Syntax)

- Constants: a,b,c,e
- Functions:
 - No function
- Predicates:
 - on: binary
 - above: binary
 - clear: unary
 - ontable: unary

Models—Examples.

Language (syntax)

- Constants: a, b, c, e
- Predicates:
 - on (binary)
 - above (binary)
 - clear (unary)
 - ontable(unary)

A possible Model I_1 (semantics)

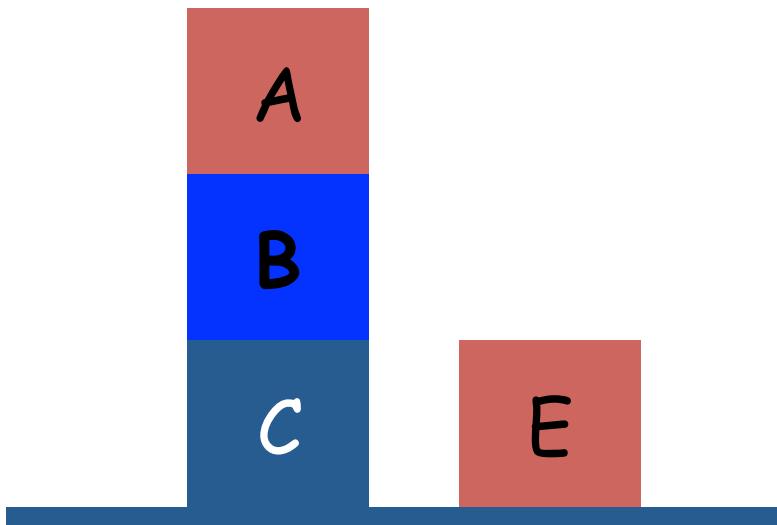
- $D = \{\underline{A}, \underline{B}, \underline{C}, \underline{E}\}$
- $\Phi(a) = \underline{A}, \Phi(b) = \underline{B}, \Phi(c) = \underline{C}, \Phi(e) = \underline{E}.$
- $\Psi(\text{on}) = \{(\underline{A}, \underline{B}), (\underline{B}, \underline{C})\}$
- $\Psi(\text{above}) = \{(\underline{A}, \underline{B}), (\underline{B}, \underline{C}), (\underline{A}, \underline{C})\}$
- $\Psi(\text{clear}) = \{\underline{A}, \underline{E}\}$
- $\Psi(\text{ontable}) = \{\underline{C}, \underline{E}\}$

Models—Examples.

Model I₁

- $D = \{\underline{A}, \underline{B}, \underline{C}, \underline{E}\}$
- $\Phi(a) = \underline{A}, \Phi(b) = \underline{B},$
 $\Phi(c) = \underline{C}, \Phi(e) = \underline{E}.$
- $\Psi(on) = \{(\underline{A}, \underline{B}), (\underline{B}, \underline{C})\}$
- $\Psi(above) = \{(\underline{A}, \underline{B}),$
 $(\underline{B}, \underline{C}), (\underline{A}, \underline{C})\}$
- $\Psi(clear) = \{\underline{A}, \underline{E}\}$
- $\Psi(ontable) = \{\underline{C}, \underline{E}\}$

Environment



Models—Formulas true or false?

Model I₁

- $D = \{\underline{A}, \underline{B}, \underline{C}, \underline{E}\}$
- $\Phi(a) = \underline{A}, \Phi(b) = \underline{B},$
 $\Phi(c) = \underline{C}, \Phi(e) = \underline{E}.$
- $\Psi(\text{on}) = \{(\underline{A}, \underline{B}), (\underline{B}, \underline{C})\}$
- $\Psi(\text{above}) = \{(\underline{A}, \underline{B}),$
 $(\underline{B}, \underline{C}), (\underline{A}, \underline{C})\}$
- $\Psi(\text{clear}) = \{\underline{A}, \underline{E}\}$
- $\Psi(\text{ontable}) = \{\underline{C}, \underline{E}\}$

$\forall X, Y. \text{on}(X, Y) \rightarrow \text{above}(X, Y)$

$X = \underline{A}, Y = \underline{B}$

$X = \underline{C}, Y = \underline{A}$

...

$\forall X, Y. \text{above}(X, Y) \rightarrow \text{on}(X, Y)$

$X = \underline{A}, Y = \underline{B}$

$X = \underline{A}, Y = \underline{C}$

Models—Examples.

Model I₁

- $D = \{\underline{A}, \underline{B}, \underline{C}, \underline{E}\}$
- $\Phi(a) = \underline{A}, \Phi(b) = \underline{B},$
 $\Phi(c) = \underline{C}, \Phi(e) = \underline{E}.$
- $\Psi(\text{on}) = \{(\underline{A}, \underline{B}), (\underline{B}, \underline{C})\}$
- $\Psi(\text{above}) = \{(\underline{A}, \underline{B}),$
 $(\underline{B}, \underline{C}), (\underline{A}, \underline{C})\}$
- $\Psi(\text{clear}) = \{\underline{A}, \underline{E}\}$
- $\Psi(\text{ontable}) = \{\underline{C}, \underline{E}\}$

$\forall X \exists Y. (\text{clear}(X) \vee \text{on}(Y, X))$

$X = \underline{A}$

$X = \underline{C}, Y = \underline{B}$

...

$\exists Y \forall X. (\text{clear}(X) \vee \text{on}(Y, X))$

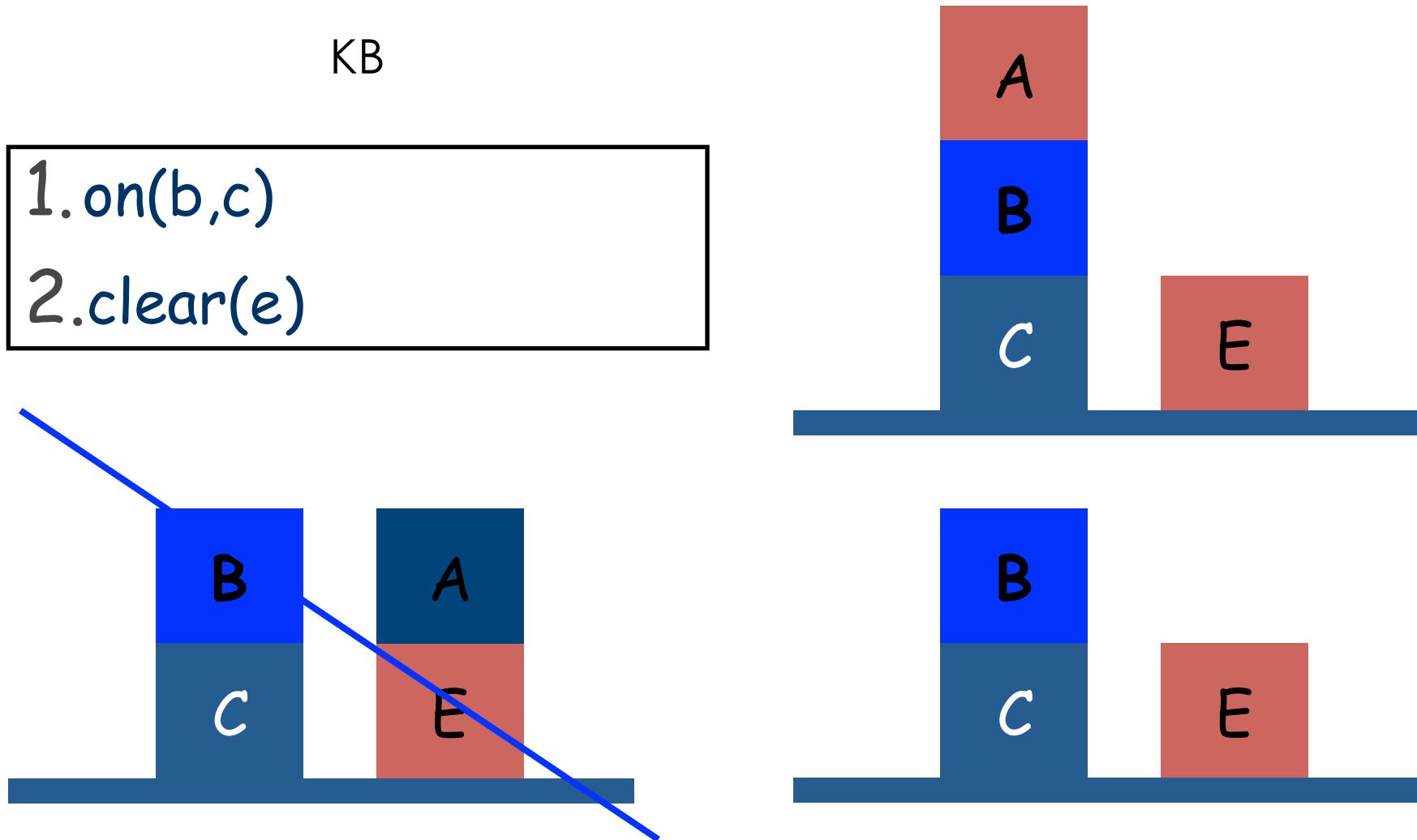
$Y = \underline{A} ? \text{No! } (X = \underline{C})$

$Y = \underline{C} ? \text{No! } (X = \underline{B})$

$Y = \underline{E} ? \text{No! } (X = \underline{B})$

$Y = \underline{B} ? \text{No! } (X = \underline{B})$

KB—many models



Models

- ▶ Let our Knowledge base KB, consist of a set of formulas.
- ▶ We say that \mathbf{I} is a **model** of KB or that \mathbf{I} **satisfies** KB
 - ▶ If, every formula $f \in \text{KB}$ is true under \mathbf{I}
- ▶ We write $\mathbf{I} \models \text{KB}$ if \mathbf{I} satisfies KB, and $\mathbf{I} \models f$ if f is true under \mathbf{I} .

What's Special About Models?

- ▶ When we write KB, we intend that the real world (i.e. our set theoretic abstraction of it) is one of its models.
- ▶ This means that every statement in KB is **true** in the real world.
- ▶ Note however, that not every thing true in the real world need be contained in KB. We might have only incomplete knowledge.

Models support reasoning.

- ▶ Suppose formula f is not mentioned in KB, but is true in every model of KB; i.e.,
 $I \models KB \rightarrow I \models f$.
- ▶ Then we say that f is a **logical consequence** of KB or that KB **entails** f .
- ▶ Since the real world is a model of KB, f must be true in the real world.
- ▶ This means that entailment is a way of finding new true facts that were not explicitly mentioned in KB.

??? If KB doesn't entail f , is f false in the real world?

Logical Consequence Example

- ▶ **elephant(clyde)**
 - ▶ the individual denoted by the symbol *clyde* in the set denoted by *elephant* (has the property that it is an *elephant*).
- ▶ **teacup(cup)**
 - ▶ *cup* is a teacup.
- ▶ Note that in both cases a unary predicate specifies a set of individuals. Asserting a unary predicate to be true of a term means that the individual denoted by that term is in the specified set.
 - ▶ Formally, we map individuals to TRUE/FALSE (this is an indicator function for the set).

Logical Consequence Example

- ▶ $\forall X, Y. \text{elephant}(X) \wedge \text{teacup}(Y) \rightarrow \text{largerThan}(X, Y)$
 - ▶ For all pairs of individuals if the first is an elephant and the second is a teacup, then the pair of objects are related to each other by the *largerThan* relation.
 - ▶ For pairs of individuals who are not elephants and teacups, the formula is immediately true.

Logical Consequence Example

- ▶ $\forall X, Y. \text{largerThan}(X, Y) \rightarrow \neg \text{fitsIn}(X, Y)$
 - ▶ For all pairs of individuals if X is larger than Y (the pair is in the `largerThan` relation) then we cannot have that X fits in Y (the pair cannot be in the `fitsIn` relation).
 - ▶ (The relation `largerThan` has a empty intersection with the `fitsIn` relation).

Logical Consequences

- ▶ $\neg \text{fitsIn}(\text{clyde}, \text{cup})$
- ▶ We know $\text{largerThan}(\text{clyde}, \text{teacup})$ from the first implication. Thus we know this from the second implication.

Logical Consequences

fitsIn

\neg fitsIn

largerThan

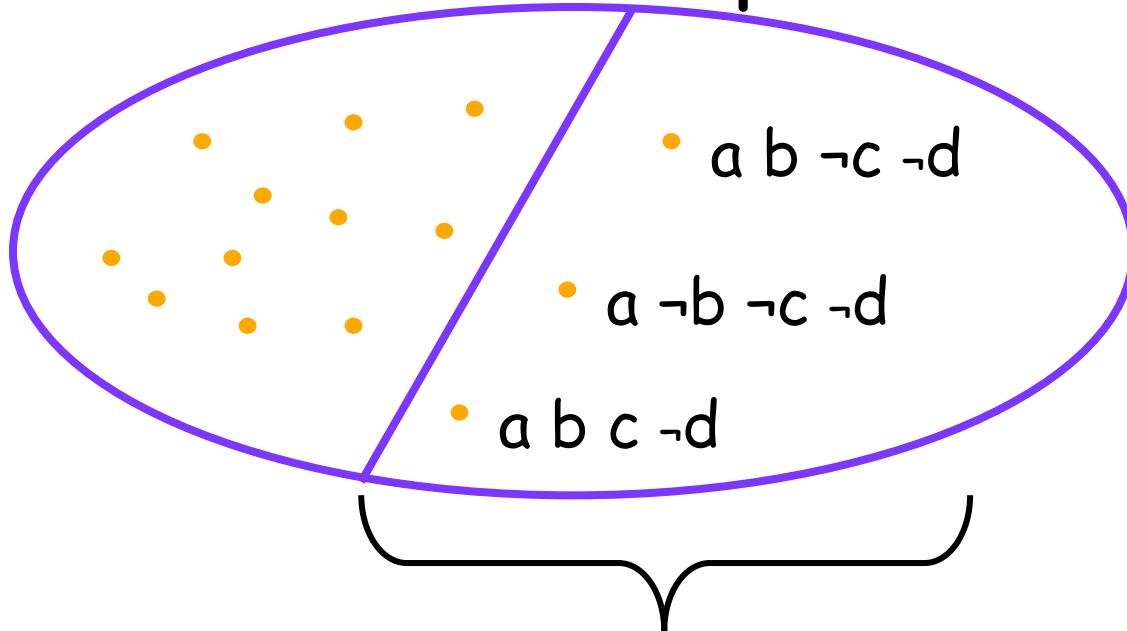
Elephants \times teacups
(clyde , cup)

Logical Consequence Example

- ▶ If an interpretation satisfies KB, then the set of pairs *elephant X teacup* must be a subset of *largerThan*, which is disjoint from *fitsIn*.
- ▶ Therefore, the pair *(clyde,cup)* must be in the complement of the set *fitsIn*.
- ▶ Hence, $\neg \text{fitsIn}(\text{clyde}, \text{cup})$ must be true in every interpretation that satisfies KB.
- ▶ $\neg \text{fitsIn}(\text{clyde}, \text{cup})$ is a logical consequence of KB.

Models Graphically

Set of All Interpretations



Consequences? $a, c \rightarrow b, b \rightarrow c, d \rightarrow b, \neg b \rightarrow \neg c$

Models and Interpretations

- ▶ the more sentences in KB, the fewer models (satisfying interpretations) there are.
- ▶ The more you write down (as long as it's all true!), the “closer” you get to the “real world”! Because Each sentence in KB rules out certain unintended interpretations.
- ▶ This is called **axiomatizing the domain**

Computing logical consequences

- ▶ We want procedures for computing logical consequences that can be implemented in our programs.
- ▶ This would allow us to reason with our knowledge
 - ▶ Represent the knowledge as logical formulas
 - ▶ Apply procedures for generating logical consequences
- ▶ These procedures are called **proof procedures**.

Proof Procedures

- ▶ Interesting, proof procedures work by simply manipulating formulas. They do not know or care anything about interpretations.
- ▶ Nevertheless they respect the semantics of interpretations!
- ▶ We will develop a proof procedure for first-order logic called resolution.
 - ▶ Resolution is the mechanism used by PROLOG

Properties of Proof Procedures

- ▶ Before presenting the details of resolution, we want to look at properties we would like to have in a (any) proof procedure.
- ▶ We write $\text{KB} \vdash f$ to indicate that f can be proved from KB (the proof procedure used is implicit).

Properties of Proof Procedures

- ▶ Soundness
 - ▶ $\text{KB} \vdash f \rightarrow \text{KB} \vDash f$
i.e all conclusions arrived at via the proof procedure are correct: they are logical consequences.
- ▶ Completeness
 - ▶ $\text{KB} \vDash f \rightarrow \text{KB} \vdash f$
i.e. every logical consequence can be generated by the proof procedure.
- ▶ Note proof procedures are computable, but they might have very high complexity in the worst case. So completeness is not necessarily achievable in practice.

Resolution

- ▶ **Clausal form.**
 - ▶ Resolution works with formulas expressed in clausal form.
 - ▶ A **literal** is an atomic formula or the negation of an atomic formula. dog(fido) , $\neg\text{cat(fido)}$
 - ▶ A **clause** is a disjunction of literals:
 - ▶ $\neg\text{owns(fido,fred)} \vee \neg\text{dog(fido)} \vee \text{person(fred)}$
 - ▶ We write
 $(\neg\text{owns(fido,fred)}, \neg\text{dog(fido)}, \text{person(fred)})$
 - ▶ A **clausal theory** is a conjunction of clauses.

Resolution

▶ Prolog Programs

- ▶ Prolog programs are clausal theories.
- ▶ However, each clause in a Prolog program is Horn.
- ▶ A horn clause contains at most one positive literal.

- ▶ The horn clause

$$\neg q_1 \vee \neg q_2 \vee \dots \vee \neg q_n \vee p$$

is equivalent to

$$q_1 \wedge q_2 \wedge \dots \wedge q_n \Rightarrow p$$

and is written as the following rule in Prolog:

$$p :- q_1 , q_2 , \dots , q_n$$

Resolution Rule for Ground Clauses

- ▶ The resolution proof procedure consists of only one simple rule:
 - ▶ From the two clauses
 - ▶ $(P, Q_1, Q_2, \dots, Q_k)$
 - ▶ $(\neg P, R_1, R_2, \dots, R_n)$
 - ▶ We infer the new clause
 - ▶ $(Q_1, Q_2, \dots, Q_k, R_1, R_2, \dots, R_n)$
- ▶ Example:
 - ▶ $(\neg \text{largerThan}(\text{clyde}, \text{cup}), \neg \text{fitsIn}(\text{clyde}, \text{cup}))$
 - ▶ $(\text{fitsIn}(\text{clyde}, \text{cup}))$
 - ⇒ $\neg \text{largerThan}(\text{clyde}, \text{cup})$

Resolution Proof: Forward chaining

- ▶ Logical consequences can be generated from the resolution rule in two ways:
 1. Forward Chaining inference.

- ▶ If we have a sequence of clauses C_1, C_2, \dots, C_k
- ▶ Such that each C_i is either in KB or is the result of a resolution step involving two prior clauses in the sequence.
- ▶ We then have that $KB \vdash C_k$.

Forward chaining is sound so we also have $KB \vDash C_k$

Resolution Proof: Refutation proofs

2. Refutation proofs.

- ▶ We determine if $\text{KB} \vdash f$ by showing that a contradiction can be generated from $\text{KB} \wedge \neg f$.
- ▶ In this case a contradiction is an empty clause () .
- ▶ We employ resolution to construct a sequence of clauses C_1, C_2, \dots, C_m such that
 - C_i is in $\text{KB} \wedge \neg f$, or is the result of resolving two previous clauses in the sequence.
 - $C_m = ()$ i.e. its the empty clause.

Resolution Proof: Refutation proofs

- ▶ If we can find such a sequence $C_1, C_2, \dots, C_m = ()$, we have that
 - ▶ $KB \vdash f$.
 - ▶ Furthermore, this procedure is sound so
 - ▶ $KB \vDash f$
- ▶ And the procedure is also complete so it is capable of finding a proof of any f that is a logical consequence of KB . I.e.
 - ▶ If $KB \vDash f$ then we can generate a refutation from $KB \wedge \neg f$

Resolution Proofs Example

Want to prove $\text{likes}(\text{clyde}, \text{peanuts})$ from:

1. $(\text{elephant}(\text{clyde}), \text{giraffe}(\text{clyde}))$
2. $(\neg \text{elephant}(\text{clyde}), \text{likes}(\text{clyde}, \text{peanuts}))$
3. $(\neg \text{giraffe}(\text{clyde}), \text{likes}(\text{clyde}, \text{leaves}))$
4. $\neg \text{likes}(\text{clyde}, \text{leaves})$

Forward Chaining Proof:

- ▶ 3&4 → $\neg \text{giraffe}(\text{clyde})$ [5.]
- ▶ 5&1 → $\text{elephant}(\text{clyde})$ [6.]
- ▶ 6&2 → $\text{likes}(\text{clyde}, \text{peanuts})$ [7.] ✓

Resolution Proofs Example

1. (elephant(clyde), giraffe(clyde))
2. (\neg elephant(clyde), likes(clyde,peanuts))
3. (\neg giraffe(clyde), likes(clyde,leaves))
4. \neg likes(clyde,leaves)

Refutation Proof:

- ▶ \neg likes(clyde,peanuts) [5.]
- ▶ 5&2 \rightarrow \neg elephant(clyde) [6.]
- ▶ 6&1 \rightarrow giraffe(clyde) [7.]
- ▶ 7&3 \rightarrow likes(clyde,leaves) [8.]
- ▶ 8&4 \rightarrow () ✓

Resolution Proofs

- ▶ Proofs by refutation have the advantage that they are easier to find.
 - ▶ They are more focused to the particular conclusion we are trying to reach.
-
- ▶ To develop a complete resolution proof procedure for First-Order logic we need :
 1. A way of converting KB and f (the query) into clausal form.
 2. A way of doing resolution even when we have variables (unification).

Conversion to Clausal Form

To convert the KB into Clausal form we perform the following 8-step procedure:

1. **Eliminate Implications.**
2. **Move Negations inwards (and simplify $\neg\neg$).**
3. **Standardize Variables.**
4. **Skolemize.**
5. **Convert to Prenix Form.**
6. **Distribute conjunctions over disjunctions.**
7. **Flatten nested conjunctions and disjunctions.**
8. **Convert to Clauses.**

C-T-C-F: Eliminate implications

We use this example to show each step:

$$\begin{aligned} \forall X.p(X) \rightarrow & \left((\forall Y.p(Y) \rightarrow p(f(X,Y))) \right. \\ & \left. \wedge \neg(\forall Y. \neg q(X,Y) \wedge p(Y)) \right) \end{aligned}$$

1. Eliminate implications: $A \rightarrow B \rightarrow \neg A \vee B$

$$\begin{aligned} \forall X. \neg p(X) \vee & \left((\forall Y. \neg p(Y) \vee p(f(X,Y))) \right. \\ & \left. \wedge \neg(\forall Y. \neg q(X,Y) \wedge p(Y)) \right) \end{aligned}$$

C-T-C-F: Move \neg Inwards

$$\begin{aligned} \forall X. \neg p(X) \\ \vee \left((\forall Y. \neg p(Y) \vee p(f(X, Y))) \right. \\ \left. \wedge \neg(\forall Y. \neg q(X, Y) \wedge p(Y)) \right) \end{aligned}$$

2. Move Negations Inwards (and simplify $\neg\neg$)

$$\begin{aligned} \forall X. \neg p(X) \\ \vee \left((\forall Y. \neg p(Y) \vee p(f(X, Y))) \right. \\ \left. \wedge (\exists Y. q(X, Y) \vee \neg p(Y)) \right) \end{aligned}$$

C-T-C-F: : \neg continue...

Rules for moving negations inwards

- ▶ $\neg(A \wedge B) \rightarrow \neg A \vee \neg B$
- ▶ $\neg(A \vee B) \rightarrow \neg A \wedge \neg B$
- ▶ $\neg \forall X. f \rightarrow \exists X. \neg f$
- ▶ $\neg \exists X. f \rightarrow \forall X. \neg f$
- ▶ $\neg \neg A \rightarrow A$

C-T-C-F: Standardize Variables

$$\begin{aligned} \forall X. \neg p(X) \\ \vee \left((\forall Y. \neg p(Y) \vee p(f(X, Y))) \right. \\ \left. \wedge (\exists Y. q(X, Y) \vee \neg p(Y)) \right) \end{aligned}$$

3. Standardize Variables (Rename variables so that each quantified variable is unique)

$$\begin{aligned} \forall X. \neg p(X) \\ \vee \left((\forall Y. (\neg p(Y) \vee p(f(X, Y)))) \right. \\ \left. \wedge (\exists Z. q(X, Z) \vee \neg p(Z)) \right) \end{aligned}$$

C-T-C-F: Skolemize

$$\begin{aligned} \forall X. \neg p(X) \\ \vee \left((\forall Y. \neg p(Y) \vee p(f(X, Y))) \right. \\ \left. \wedge (\exists Z. q(X, Z) \vee \neg p(Z)) \right) \end{aligned}$$

4. Skolemize (Remove existential quantifiers by introducing new function symbols).

$$\begin{aligned} \forall X. \neg p(X) \\ \vee \left((\forall Y. \neg p(Y) \vee p(f(X, Y))) \right. \\ \left. \wedge (q(X, g(X)) \vee \neg p(g(X))) \right) \end{aligned}$$

C-T-C-F: Skolemization

Consider $\exists Y. \text{elephant}(Y) \wedge \text{friendly}(Y)$

- ▶ This asserts that there is some individual (binding for Y) that is both an elephant and friendly.
- ▶ To remove the existential, we **invent** a name for this individual, say **a**. This is a new constant symbol **not equal to any previous constant symbols** to obtain:
 $\text{elephant}(a) \wedge \text{friendly}(a)$
- ▶ This is saying the same thing, since we do not know anything about the new constant **a**.

C-T-C-F: Skolemization

- ▶ It is essential that the introduced symbol “a” is **new**. Else we might know something else about “a” in KB.
- ▶ If we did know something else about “a” we would be asserting more than the existential.
- ▶ In original quantified formula we know nothing about the variable “Y”. Just what was being asserted by the existential formula.

C-T-C-F: Skolemization

Now consider $\forall X \exists Y. \text{loves}(X, Y)$.

- ▶ This formula claims that for every X there is some Y that X loves (perhaps a different Y for each X).
- ▶ Replacing the existential by a new constant won't work
 $\forall X. \text{loves}(X, a)$.

Because this asserts that there is a **particular** individual “ a ” loved by every X .

- ▶ To properly convert existential quantifiers scoped by universal quantifiers we must use **functions** not just constants.

C-T-C-F: Skolemization

- ▶ We must use a function that mentions **every universally quantified variable that scopes the existential.**
- ▶ In this case X scopes Y so we must replace the existential Y by a function of X
 $\forall X. \text{loves}(X, g(X)).$
where g is a **new** function symbol.

- ▶ This formula asserts that for every X there is some individual (given by $g(X)$) that X loves. $g(X)$ can be different for each different binding of X .

C-T-C-F: Skolemization Examples

- ▶ $\forall XYZ \exists W.r(X,Y,Z,W) \rightarrow \forall XYZ.r(X,Y,Z,h1(X,Y,Z))$

- ▶ $\forall XY \exists W.r(X,Y,g(W)) \rightarrow \forall XY.r(X,Y,Z,g(h2(X,Y)))$

- ▶ $\forall XY \exists W \forall Z.r(X,Y,W) \wedge q(Z,W)$
→ $\forall XYZ.r(X,Y,h3(X,Y)) \wedge q(Z,h3(X,Y))$

C-T-C-F: Convert to prenex

$\forall X. \neg p(X)$

$$\vee \left(\forall Y. \neg p(Y) \vee p(f(X, Y)) \wedge q(X, g(X)) \vee \neg p(g(X)) \right)$$

5. Convert to prenex form. (Bring all quantifiers to the front—only universals, each with different name).

$\forall X \forall Y. \neg p(X)$

$$\vee \left(\neg p(Y) \vee p(f(X, Y)) \wedge q(X, g(X)) \vee \neg p(g(X)) \right)$$

C-T-C-F: Conjunctions over disjunctions

$$\begin{aligned} \forall X \forall Y. & \neg p(X) \\ & \vee \left(\left(\neg p(Y) \vee p(f(X,Y)) \right) \right. \\ & \quad \left. \wedge (q(X,g(X)) \vee \neg p(g(X))) \right) \end{aligned}$$

6. Conjunctions over disjunctions

$$A \vee (B \wedge C) \rightarrow (A \vee B) \wedge (A \vee C)$$

$$\begin{aligned} \forall X \forall Y. & (\neg p(X) \vee \neg p(Y) \vee p(f(X,Y))) \\ & \wedge (\neg p(X) \vee q(X,g(X)) \vee \neg p(g(X))) \end{aligned}$$

C-T-C-F: flatten & convert to clauses

7. Flatten nested conjunctions and disjunctions.

$$(A \vee (B \vee C)) \rightarrow (A \vee B \vee C)$$

8. Convert to Clauses (remove quantifiers and break apart conjunctions).

$$\forall X Y. (\neg p(X) \vee \neg p(Y) \vee p(f(X, Y))) \\ \wedge (\neg p(X) \vee q(X, g(X)) \vee \neg p(g(X)))$$

- a) $\neg p(X) \vee \neg p(Y) \vee p(f(X, Y))$
- b) $\neg p(X) \vee q(X, g(X)) \vee \neg p(g(X))$

Unification

- ▶ Ground clauses are clauses with no variables in them. For ground clauses we can use syntactic identity to detect when we have a P and $\neg P$ pair.
- ▶ What about variables can the clauses
 - ▶ $(P(\text{john}), Q(\text{fred}), R(X))$
 - ▶ $(\neg P(Y), R(\text{susan}), R(Y))$Be resolved?

Unification.

- ▶ Intuitively, once reduced to clausal form, all remaining variables are universally quantified. So, implicitly $(\neg P(Y), R(susan), R(Y))$ represents a whole set of ground clauses like
 - ▶ $(\neg P(\text{fred}), R(susan), R(\text{fred}))$
 - ▶ $(\neg P(\text{john}), R(susan), R(\text{john}))$
 - ▶ ...
- ▶ So there is a “specialization” of this clause that can be resolved with $(P(\text{john}), Q(\text{fred}), R(X))$

Unification.

- ▶ We want to be able to match conflicting literals, even when they have variables. This matching process automatically determines whether or not there is a “specialization” that matches.
- ▶ We don’t want to over specialize!

Unification.

- ▶ $(\neg p(X), s(X), q(fred))$
- ▶ $(p(Y), r(Y))$
- ▶ Possible resolvents
 - ▶ $(s(john), q(fred), r(john)) \{Y=X, X=john\}$
 - ▶ $(s(sally), q(fred), r(sally)) \{Y=X, X=sally\}$
 - ▶ $(s(X), q(fred), r(X)) \{Y=X\}$
- ▶ The last resolvent is “**most-general**”, the other two are specializations of it.
- ▶ We want to keep the most general clause so that we can use it future resolution steps.

Unification.

- ▶ unification is a mechanism for finding a “most general” matching.
- ▶ First we consider substitutions.
 - ▶ A substitution is a finite set of equations of the form

$$V = t$$

where V is a variable and t is a term not containing V . (t might contain other variables).

Substitutions.

- ▶ We can apply a substitution σ to a formula f to obtain a new formula $f\sigma$ by simultaneously replacing every variable mentioned in the left hand side of the substitution by the right hand side.

$$p(X, g(Y, Z)) [X=Y, Y=f(a)] \rightarrow p(Y, g(f(a), Z))$$

- ▶ Note that the substitutions are not applied sequentially, i.e., the first Y is not subsequently replaced by $f(a)$.

Substitutions.

- ▶ We can compose two substitutions, θ and σ to obtain a new substitution $\theta\sigma$.

Let $\theta = \{X_1=s_1, X_2=s_2, \dots, X_m=s_m\}$

$\sigma = \{Y_1=t_1, Y_2=t_2, \dots, Y_k=s_k\}$

To compute $\theta\sigma$

1. $S = \{X_1=s_1\sigma, X_2=s_2\sigma, \dots, X_m=s_m\sigma, Y_1=t_1, Y_2=t_2, \dots, Y_k=s_k\}$

we apply σ to each RHS of θ and then add all of the equations of σ .

Substitutions.

1. $S = \{X_1=s_1\sigma, X_2=s_2\sigma, \dots, X_m=s_m\sigma, Y_1=t_1, Y_2=t_2, \dots, Y_k=s_k\}$
2. Delete any identities, i.e., equations of the form $V=V$.
3. Delete any equation $Y_i=s_i$ where Y_i is equal to one of the X_j in θ .

The final set S is the composition $\theta\sigma$.

Composition Example.

$$\theta = \{X=f(Y), Y=Z\}, \sigma = \{X=a, Y=b, Z=Y\}$$

$\theta\sigma$

Substitutions.

- ▶ The empty substitution $\varepsilon = \{\}$ is also a substitution, and it acts as an identity under composition.
- ▶ More importantly substitutions when applied to formulas are associative:

$$(f\theta)\sigma = f(\theta\sigma)$$

- ▶ Composition is simply a way of converting the sequential application of a series of substitutions to a single simultaneous substitution.

Unifiers.

- ▶ A **unifier** of two formulas f and g is a substitution σ that makes f and g **syntactically identical**.
- ▶ Not all formulas can be unified—substitutions only affect variables.

$$p(f(X), a) \quad p(Y, f(w))$$

- ▶ This pair cannot be unified as there is no way of making $a = f(w)$ with a substitution.
- ▶ Note we typically use UPPER CASE to denote variables, lower case for constants.

MGU.

- ▶ A substitution σ of two formulas f and g is a **Most General Unifier (MGU)** if
 1. σ is a unifier.
 2. For every other unifier θ of f and g there must exist a third substitution λ such that
$$\theta = \sigma\lambda$$
- This says that every other unifier is “more specialized than σ . The MGU of a pair of formulas f and g is unique up to renaming.

MGU.

$$p(f(X), Z) \quad p(Y, a)$$

1. $\sigma = \{Y = f(a), X=a, Z=a\}$ is a unifier.

$$\begin{aligned} p(f(X), Z)\sigma &= \\ p(Y, a)\sigma &= \end{aligned}$$

But it is not an MGU.

2. $\theta = \{Y=f(X), Z=a\}$ is an MGU.

$$\begin{aligned} p(f(X), Z)\theta &= \\ p(Y, a)\theta &= \end{aligned}$$

MGU.

$$p(f(X), Z) \quad p(Y, a)$$

3. $\sigma = \theta\lambda$, where $\lambda = \{X=a\}$

$$\sigma = \{Y = f(a), X=a, Z=a\}$$

$$\lambda = \{X=a\}$$

$$\theta\lambda =$$

MGU.

- ▶ The MGU is the “least specialized” way of making clauses with universal variables match.
- ▶ We can compute MGUs.
- ▶ Intuitively we line up the two formulas and find the first sub-expression where they disagree. The pair of subexpressions where they **first** disagree is called the **disagreement set**.
- ▶ The algorithm works by successively fixing disagreement sets until the two formulas become syntactically identical.

MGU.

To find the MGU of two formulas f and g .

1. $k = 0; \sigma_0 = \{\}; S_0 = \{f, g\}$
2. If S_k contains an identical pair of formulas stop, and return σ_k as the MGU of f and g .
3. Else find the disagreement set $D_k = \{e_1, e_2\}$ of S_k
4. If $e_1 = V$ a variable, and $e_2 = t$ a term not containing V (or vice-versa) then let

$$\sigma_{k+1} = \sigma_k \{V=t\} \quad (\text{Compose the additional substitution})$$

$$S_{k+1} = S_k \{V=t\} \quad (\text{Apply the additional substitution})$$

$k = k+1$
GOTO 2

5. Else stop, f and g cannot be unified.

MGU Example 1.

$$S_0 = \{p(f(a), g(X)) : p(Y, Y)\}$$

MGU Example 2.

$$S_0 = \{p(a, X, h(g(Z))) : p(Z, h(Y), h(Y))\}$$

MGU Example 3.

$$S_0 = \{p(X, X) \ ; \ p(Y, f(Y))\}$$

Non-Ground Resolution

- ▶ Resolution of non-ground clauses. From the two clauses

$$(L, Q_1, Q_2, \dots, Q_k)$$
$$(\neg M, R_1, R_2, \dots, R_n)$$

Where there exists σ a MGU for L and M.

We infer the new clause

$$(Q_1\sigma, \dots, Q_k\sigma, R_1\sigma, \dots, R_n\sigma)$$

Non-Ground Resolution E.G.

1. $(p(X), q(g(X)))$
2. $(r(a), q(Z), \neg p(a))$

$L=p(X); M=p(a)$
 $\sigma = \{X=a\}$

3. $R[1a,2c]\{\{X=a\} (q(g(a)), r(a), q(Z))$

The notation is important.

- ▶ “R” means resolution step.
- ▶ “1a” means the **first (a-th)** literal in the first clause i.e. $p(X)$.
- ▶ “2c” means the **third (c-th)** literal in the second clause, $\neg p(a)$.
 - ▶ 1a and 2c are the “clashing” literals.
- ▶ $\{X=a\}$ is the substitution applied to make the clashing literals identical.

Resolution Proof Example

“Some patients like all doctors. No patient likes any quack. Therefore no doctor is a quack.”

Resolution Proof Step 1.

Pick symbols to represent these assertions.

$p(X)$: X is a patient

$d(x)$: X is a doctor

$q(X)$: X is a quack

$I(X,Y)$: X likes Y

Resolution Proof Example

Resolution Proof Step 2.

Convert each assertion to a first-order formula.

1. Some patients like all doctors.

F1.

Resolution Proof Example

2. No patient likes any quack

F2.

3. Therefore no doctor is a quack.

Query.

Resolution Proof Example

Resolution Proof Step 3.

Convert to Clausal form.

F1.

F2.

Negation of Query.

Resolution Proof Example

Resolution Proof Step 4.

Resolution Proof from the Clauses.

1. $p(a)$
2. $(\neg d(Y), I(a,Y))$
3. $(\neg p(Z), \neg q(R), \neg I(Z,R))$
4. $d(b)$
5. $q(b)$

Answer Extraction.

- ▶ The previous example shows how we can answer true-false questions. With a bit more effort we can also answer “fill-in-the-blanks” questions (e.g., what is wrong with the car?).
- ▶ As in Prolog we use free variables in the query where we want the fill in the blanks. We simply need to keep track of the binding that these variables received in proving the query.
 - ▶ `parent(art, jon)` –is art one of jon’s parents?
 - ▶ `parent(X, jon)` -who is one of jon’s parents?

Answer Extraction.

- ▶ A simple bookkeeping device is to use an predicate symbol `answer(X,Y,...)` to keep track of the bindings automatically.
- ▶ To answer the query `parent(X,jon)`, we construct the clause
 $(\neg \text{parent}(X,\text{jon}), \text{answer}(X))$
- ▶ Now we perform resolution until we obtain a clause consisting of only answer literals (previously we stopped at empty clauses).

Answer Extraction: Example 1

1. father(art, jon)
2. father(bob,kim)
3. $(\neg \text{father}(Y,Z), \text{parent}(Y,Z))$
i.e. *all fathers are parents*
4. $(\neg \text{parent}(X,\text{jon}), \text{answer}(X))$
i.e. the query is: who is parent of jon?

Here is a resolution proof:

5. $R[4,3b]\{Y=X, Z=\text{jon}\}$
 $(\neg \text{father}(X,\text{jon}), \text{answer}(X))$
6. $R[5,1]\{X=\text{art}\} \text{ answer}(\text{art})$

so art is parent of jon

Answer Extraction: Example 2

1. $(\text{father}(\text{art}, \text{jon}), \text{father}(\text{bob}, \text{jon}))$ //either bob or art is parent of jon
2. $\text{father}(\text{bob}, \text{kim})$
3. $(\neg \text{father}(Y, Z), \text{parent}(Y, Z))$ //i.e. all fathers are parents
4. $(\neg \text{parent}(X, \text{jon}), \text{answer}(X))$ //i.e. query is parent(X,jon)

Here is a resolution proof:

5. $R[4,3b]\{Y=X, Z=\text{jon}\} (\neg \text{father}(X, \text{jon}), \text{answer}(X))$
6. $R[5,1a]\{X=\text{art}\} (\text{father}(\text{bob}, \text{jon}), \text{answer}(\text{art}))$
7. $R[6,3b]\{Y=\text{bob}, Z=\text{jon}\}$
 $(\text{parent}(\text{bob}, \text{jon}), \text{answer}(\text{art}))$
8. $R[7,4]\{X=\text{bob}\} (\text{answer}(\text{bob}), \text{answer}(\text{art}))$

A disjunctive answer: either bob or art is parent of jon.

Factoring

1. $(p(X), p(Y)) \quad // \forall X. \forall Y. \neg p(X) \rightarrow p(Y)$
2. $(\neg p(V), \neg p(W)) \quad // \forall V. \forall W. p(V) \rightarrow \neg p(W)$

▶ These clauses are intuitively contradictory, but following the strict rules of resolution only we obtain:

3. $R[1a,2a](X=V) (p(Y), \neg p(W))$
Renaming variables: $(p(Q), \neg p(Z))$
4. $R[3b,1a](X=Z) (p(Y), p(Q))$

No way of generating empty clause!

Factoring is needed to make resolution over non-ground clauses complete, without it resolution is incomplete!

Factoring.

- ▶ If two or more literals of a clause C have an mgu θ , then $C\theta$ with all duplicate literals removed is called a **factor** of C.
- ▶ $C = (p(X), p(f(Y)), \neg q(X))$
 $\theta = \{X=f(Y)\}$
 $C\theta = (p(f(Y)), p(f(Y)), \neg q(f(Y))) \rightarrow (p(f(Y)), \neg q(f(Y)))$ is a factor

Adding a factor of a clause can be a step of proof:

1. $(p(X), p(Y))$
2. $(\neg p(V), \neg p(W))$
3. $f[1ab]\{X=Y\} p(Y)$
4. $f[2ab]\{V=W\} \neg p(W)$
5. $R[3,4]\{Y=W\} ()$.

Prolog

- ▶ Prolog search mechanism is simply an instance of resolution, except
 1. Clauses are Horn (only one positive literal)
 2. Prolog uses a specific depth first strategy when searching for a proof. (Rules are used first mentioned first used, literals are resolved away left to right).

Prolog

▶ Append:

1. `append([], Z, Z)`
2. `append([E1 | R1], Y, [E1 | Rest]) :-
append(R1, Y, Rest)`

Note:

- The second is actually the clause
`(append([E1 | R1], Y, [E1 | Rest]), \neg append(R1, Y, Rest))`
- `[]` is a constant (the empty list)
- `[X | Y]` is `cons(X, Y)`.
- So `[a, b, c]` is short hand for `cons(a, cons(b, cons(c, [])))`

Prolog: Example of proof

- ▶ Try to prove : $\text{append}([\text{a},\text{b}], [\text{c},\text{d}], [\text{a},\text{b},\text{c},\text{d}])$:

1. $\text{append}([], \text{Z}, \text{Z})$
2. $(\text{append}([\text{E1} | \text{R1}], \text{Y}, [\text{E1} | \text{Rest}]),$
 $\neg \text{append}(\text{R1}, \text{Y}, \text{Rest}))$
3. $\neg \text{append}([\text{a},\text{b}], [\text{c},\text{d}], [\text{a},\text{b},\text{c},\text{d}])$
4. $\text{R}[3,2\text{a}]\{\text{E1}=\text{a}, \text{R1}=[\text{b}], \text{Y}=[\text{c},\text{d}], \text{Rest}=[\text{b},\text{c},\text{d}]\}$
 $\neg \text{append}([\text{b}], [\text{c},\text{d}], [\text{b},\text{c},\text{d}])$
5. $\text{R}[4,2\text{a}]\{\text{E1}=\text{b}, \text{R1}=[], \text{Y}=[\text{c},\text{d}], \text{Rest}=[\text{c},\text{d}]\}$
 $\neg \text{append}([], [\text{c},\text{d}], [\text{c},\text{d}])$
6. $\text{R}[5,1]\{\text{Z}=[\text{c},\text{d}]\} ()$

Review: One Last Example!

Consider the following English description

- ▶ Whoever can read is literate.
 - ▶ Dolphins are not literate.
 - ▶ Flipper is an intelligent dolphin.
-
- ▶ Who is intelligent but cannot read.

Example: pick symbols & convert to first-order formula

- ▶ Whoever can read is literate.
 $\forall X. \text{read}(X) \rightarrow \text{lit}(X)$
- ▶ Dolphins are not literate.
 $\forall X. \text{dolp}(X) \rightarrow \neg \text{lit}(X)$
- ▶ Flipper is an intelligent dolphin
 $\text{dolp}(\text{flipper}) \wedge \text{intell}(\text{flipper})$

- ▶ Who is intelligent but cannot read?
 $\exists X. \text{intell}(X) \wedge \neg \text{read}(X).$

Example: convert to clausal form

- ▶ $\forall X. \text{read}(X) \rightarrow \text{lit}(X)$
 $(\neg \text{read}(X), \text{lit}(X))$
- ▶ Dolphins are not literate.
 $\forall X. \text{dolp}(X) \rightarrow \neg \text{lit}(X)$
 $(\neg \text{dolp}(X), \neg \text{lit}(X))$
- ▶ Flipper is an intelligent dolphin.
 $\text{dolp}(\text{flipper})$
 $\text{intell}(\text{flipper})$
- ▶ who are intelligent but cannot read?
 $\exists X. \text{intell}(X) \wedge \neg \text{read}(X).$
 $\rightarrow \forall X. \neg \text{intell}(X) \vee \text{read}(X)$
 $\rightarrow (\neg \text{intell}(X), \text{read}(X), \text{answer}(X))$

Example: do the resolution proof

1. $(\neg \text{read}(X), \text{lit}(X))$
2. $(\neg \text{dolp}(X), \neg \text{lit}(X))$
3. $\text{dolp}(\text{flip})$
4. $\text{intell}(\text{flip})$
5. $(\neg \text{intell}(X), \text{read}(X), \text{answer}(X))$

6. R[5a,4] X=flip. $(\text{read}(\text{flip}), \text{answer}(\text{flip}))$
7. R[6,1a] X=flip. $(\text{lit}(\text{flip}), \text{answer}(\text{flip}))$
8. R[7,2b] X=flip. $(\neg \text{dolp}(\text{flip}), \text{answer}(\text{flip}))$
9. R[8,3] **answer(flip)**

so flip is intelligent but cannot read!

▶ Knowledge Representation

- ▶ This material is covered in chapters 7—10 of the text.
- ▶ Chapter 7 provides a useful motivation for logic, and an introduction to some basic ideas. It also introduces propositional logic, which is a good background for first-order logic.
- ▶ What we cover here is mainly covered in Chapters 8 and 9. However, Chapter 8 contains some additional useful examples of how first-order knowledge bases can be constructed. Chapter 9 covers forward and backward chaining mechanisms for inference, while here we concentrate on resolution.
- ▶ Chapter 10 covers some of the additional notions that have to be dealt with when using knowledge representation in AI.

▶ 1

Fahiem Bacchus, University of Toronto

Example.

- ▶ Three little pigs



▶ 3

Fahiem Bacchus, University of Toronto

Knowledge Representation

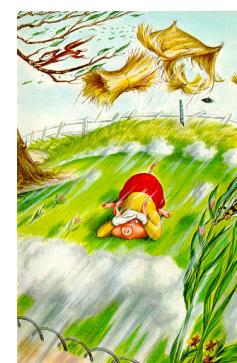
- ▶ Consider the task of understanding a simple story.
- ▶ How do we test understanding?
- ▶ Not easy, but understanding at least entails some ability to answer simple questions about the story.

▶ 2

Fahiem Bacchus, University of Toronto

Example.

- ▶ Three little pigs



▶ 4

Fahiem Bacchus, University of Toronto

Example.

- ▶ Why couldn't the wolf blow down the house made of bricks?
- ▶ What background knowledge are we applying to come to that conclusion?
 - ▶ Brick structures are stronger than straw and stick structures.
 - ▶ Objects, like the wolf, have physical limitations. The wolf can only blow so hard.

▶ 5

Fahiem Bacchus, University of Toronto

Logical Representations

- ▶ AI typically employs logical representations of knowledge.
- ▶ Logical representations useful for a number of reasons:

▶ 7

Fahiem Bacchus, University of Toronto

Why Knowledge Representation?

- ▶ Large amounts of knowledge are used to understand the world around us, and to communicate with others.
- ▶ We also have to be able to reason with that knowledge.
 - ▶ Our knowledge won't be about the blowing ability of wolfs in particular, it is about physical limits of objects in general.
 - ▶ We have to employ reasoning to make conclusions about the wolf.
 - ▶ More generally, reasoning provides an exponential or more compression in the knowledge we need to store. I.e., without reasoning we would have to store a infeasible amount of information: e.g., Elephants can't fit into teacups.

▶ 6

Fahiem Bacchus, University of Toronto

Logical Representations

- ▶ They are mathematically precise, thus we can analyze their limitations, their properties, the complexity of inference etc.
- ▶ They are formal languages, thus computer programs can manipulate sentences in the language.
- ▶ They come with both a formal **syntax** and a formal **semantics**.
- ▶ Typically, have well developed **proof theories**: formal procedures for reasoning at the syntactic level (achieved by manipulating sentences).

▶ 8

Fahiem Bacchus, University of Toronto

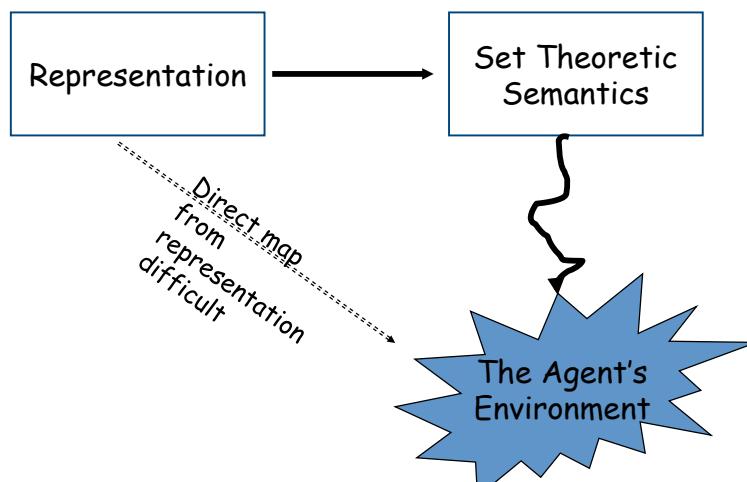
Set theoretic semantics

- ▶ Suppose our knowledge is represented in our program by some collection of data structures. We can think of these as a collection of **strings (sentences)**.
- ▶ We want a clear mapping from this set of sentences to features of the environment. What are sentences asserting about environment?
 - ▶ In other words, we want to be able to provide an intuitive interpretation of any piece of our representation.
 - ▶ Similar in spirit to having an intuitive understanding of what individual statements in a program mean. It does not mean that it is easy to understand the whole, but it provides the means to understand the whole by understanding the parts.

▶ 9

Fahiem Bacchus, University of Toronto

Set theoretic semantics



▶ 11

Fahiem Bacchus, University of Toronto

Set theoretic semantics

- ▶ Set theoretic semantics facilitates both goals.
 - ▶ It is a formal characterization, and it can be used to prove a wide range of properties of the representation.
 - ▶ It maps arbitrarily complex sentences of the logic down into intuitive assertions about the real world.
 - ▶ It is based on notions that are very close to how we think about the real world. Thus it provides the bridge from the syntax to an intuitive understanding of what is being asserted.

▶ 10

Fahiem Bacchus, University of Toronto

Semantics Formal Details

- ▶ A set of **objects**. These are objects in the environment that are important for your application.
- ▶ Distinguished subsets of objects. **Properties**.
- ▶ Distinguished sets of tuples of objects. **Relations**.
- ▶ Distinguished functions mapping tuples of objects to objects. **Functions**.

▶ 12

Fahiem Bacchus, University of Toronto

Example

- ▶ Teaching CSC384, want to represent knowledge that would be useful for making the course a successful learning experience.

- ▶ **Objects:**
 - ▶ **students, subjects, assignments, numbers.**
- ▶ **Predicates:**
 - ▶ **difficult(subject), CSMajor(student).**
- ▶ **Relations:**
 - ▶ **handedIn(student, assignment)**
- ▶ **Functions:**
 - ▶ **Grade(student, assignment) → number**

▶ 13

Fahiem Bacchus, University of Toronto

First Order Syntax

Start with a set of primitive symbols.

1. **constant** symbols.
 2. **function** symbols.
 3. **predicate** symbols (for predicates and relations).
 4. **variables.**
- Each function and predicate symbol has a specific arity (determines the number of arguments it takes).

▶ 15

Fahiem Bacchus, University of Toronto

First Order Logic

1. **Syntax:** A grammar specifying what are legal syntactic constructs of the representation.
2. **Semantics:** A formal mapping from syntactic constructs to set theoretic assertions.

▶ 14

Fahiem Bacchus, University of Toronto

First Order Syntax—Building up.

- ▶ **terms** are used as names (perhaps complex nested names) for objects in the domain.
- ▶ **Terms** of the language are either:
 - ▶ a variable
 - ▶ a constant
 - ▶ an expression of the form $f(t_1, \dots t_k)$ where
 - ▶ (a) f is a function symbol;
 - ▶ (b) k is its arity;
 - ▶ (c) each t_i is a term
- ▶ 5 is a term—a symbol representing the number 5. John is a term—a symbol representing the person John.
- ▶ $+(5,5)$ is a term—a symbol representing the number 10.

▶ 16

Fahiem Bacchus, University of Toronto

First Order Syntax—Building up.

- ▶ **Note:** constants are the same as functions taking zero arguments.
- ▶ Terms denote objects (things in the world):
 - ▶ constants denote specific objects;
 - ▶ functions map tuples of objects to other objects
 - ▶ bill, jane, father(jane), father(father(jane))
 - ▶ X , father(X), hotel7, rating(hotel7), cost(hotel7)
 - ▶ Variables like X are not yet determined, but they will eventually denote particular objects.

▶ 17

Fahiem Bacchus, University of Toronto

Semantic Intuition (formalized later).

- ▶ Atoms denote facts that can be true or false about the world
 - ▶ father_of(jane, bill), female(jane), system_down()
 - ▶ satisfied(client15), satisfied(C)
 - ▶ desires(client15, rome, week29), desires(X, Y, Z)
 - ▶ rating(hotel7, 4), cost(hotel7, 125)

▶ 19

Fahiem Bacchus, University of Toronto

First Order Syntax—Building up.

- ▶ Once we have terms we can build up formulas. Terms represent (denote) objects, formulas represent true/false assertions about these objects.
- ▶ We start with atomic formulas these are
 - ▶ expressions of the form $p(t_1, \dots, t_k)$ where
 - ▶ (a) p is a predicate symbol;
 - ▶ (b) k is its arity;
 - ▶ (c) each t_i is a term

▶ 18

Fahiem Bacchus, University of Toronto

First Order Syntax—Building up.

- ▶ Atomic formulas
- ▶ The negation (NOT) of a formula is a new formula
 - ▶ $\neg f$ ($\neg f$)
Asserts that f is false.
- ▶ The conjunction (AND) of a set of formulas is a formula.
 - ▶ $f_1 \wedge f_2 \wedge \dots \wedge f_n$ where each f_i is formula
Asserts that each formula f_i is true.

▶ 20

Fahiem Bacchus, University of Toronto

First Order Syntax—Building up.

- ▶ The disjunction (**OR**) of a set of formulas is a formula.
 - ▶ $f_1 \vee f_2 \vee \dots \vee f_n$ where each f_i is formula

Asserts that at least one formula f_i is true.
- ▶ Existential Quantification **\exists** .
 - ▶ $\exists X. f$ where X is a variable and f is a formula.

Asserts there is some object such that once X is bound to that object, f will be true.
- ▶ Universal Quantification **\forall** .
 - ▶ $\forall X. f$ where X is a variable and f is a formula.

Asserts that f is true for every object X can be bound to.

▶ 21

Fahiem Bacchus, University of Toronto

Semantics.

- ▶ Formulas (syntax) can be built up recursively, and can become arbitrarily complex.
- ▶ Intuitively, there are various distinct formulas (viewed as strings) that really are asserting the same thing
 - ▶ $\forall X, Y. \text{elephant}(X) \wedge \text{teacup}(Y) \rightarrow \text{largerThan}(X, Y)$
 - ▶ $\forall X, Y. \text{teacup}(Y) \wedge \text{elephant}(X) \rightarrow \text{largerThan}(X, Y)$
- ▶ To capture this equivalence and to make sense of complex formulas we utilize the semantics.

▶ 23

Fahiem Bacchus, University of Toronto

First Order Syntax—abbreviations.

- ▶ Implication:
 - ▶ $f_1 \rightarrow f_2$
- Take this to mean
 - ▶ $\neg f_1 \vee f_2$.

▶ 22

Fahiem Bacchus, University of Toronto

Semantics.

- ▶ A formal mapping from formulas to semantic entities (individuals, sets and relations over individuals, functions over individuals).
- ▶ The mapping mirrors the recursive structure of the syntax, so we can give any formula, no matter how complex a mapping to semantic entities.

▶ 24

Fahiem Bacchus, University of Toronto

Semantics—Formal Details

- ▶ First, we must fix the particular first-order language we are going to provide semantics for. The primitive symbols included in the syntax defines the particular language.
 $L(F,P,V)$
- ▶ F = set of function (and constant symbols)
 - ▶ Each symbol f in F has a particular arity.
- ▶ P = set of predicate and relation symbols.
 - ▶ Each symbol p in P has a particular arity.
- ▶ V = an infinite set of variables.

▶ 25

Fahiem Bacchus, University of Toronto

Intuitions: Domain

- ▶ Domain D : $d \in D$ is an *individual*
- ▶ E.g., $\{ \underline{\text{craig}}, \underline{\text{jane}}, \underline{\text{grandhotel}}, \underline{\text{le-fleabag}}, \underline{\text{rome}}, \underline{\text{portofino}}, \underline{100}, \underline{110}, \underline{120} \dots \}$
- ▶ Underlined symbols denote domain individuals (as opposed to symbols of the first-order language)
- ▶ Domains often infinite, but we'll use finite models to prime our intuitions

▶ 27

Fahiem Bacchus, University of Toronto

Semantics—Formal Details

- ▶ An **interpretation** (model) is a tuple $\langle D, \Phi, \Psi, v \rangle$
 - ▶ D is a non-empty set (domain of individuals)
- ▶ Φ is a mapping: $\Phi(f) \rightarrow (D^k \rightarrow D)$
 - ▶ maps k-ary function symbol f , to a function from k-ary tuples of individuals to individuals.
- ▶ Ψ is a mapping: $\Psi(p) \rightarrow (D^k \rightarrow \text{True/False})$
 - ▶ maps k-ary predicate symbol p , to an indicator function over k-ary tuples of individuals (a subset of D^k)
- ▶ v is a variable assignment function. $v(X) = d \in D$ (it maps every variable to some individual)

▶ 26

Fahiem Bacchus, University of Toronto

Intuitions: Φ

- ▶ $\Phi(f) \rightarrow (D^k \rightarrow D)$
Given k-ary function f , k individuals, what individual does $f(d_1, \dots, d_k)$ denote
 - ▶ 0-ary functions (constants) are mapped to specific individuals in D .
 $\Phi(\text{client17}) = \underline{\text{craig}}$, $\Phi(\text{hotel5}) = \underline{\text{le-fleabag}}$, $\Phi(\text{rome}) = \underline{\text{rome}}$
 - ▶ 1-ary functions are mapped to functions in $D \rightarrow D$
 - ▶ $\Phi(\text{minquality}) = f_{\text{minquality}}$:
 $f_{\text{minquality}}(\underline{\text{craig}}) = \underline{\text{3stars}}$
 - ▶ $\Phi(\text{rating}) = f_{\text{rating}}$:
 $f_{\text{rating}}(\underline{\text{grandhotel}}) = \underline{\text{5stars}}$
 - ▶ 2-ary functions are mapped to functions from $D^2 \rightarrow D$
 - ▶ $\Phi(\text{distance}) = f_{\text{distance}}$:
 $f_{\text{distance}}(\underline{\text{toronto}}, \underline{\text{sienna}}) = \underline{\text{3256}}$
 - ▶ n-ary functions are mapped similarly.

▶ 28

Fahiem Bacchus, University of Toronto

Intuitions: Ψ

- ▶ $\Psi(p) \rightarrow (D^k \rightarrow \text{True/False})$
 - ▶ given k-ary predicate, k individuals, does the relation denoted by p hold of these? $\Psi(p)(d_1, \dots, d_k) = \text{true?}$
- ▶ 0-ary predicates are mapped to true or false.
 $\Psi(\text{rainy}) = \text{True}$ $\Psi(\text{sunny}) = \text{False}$
- ▶ 1-ary predicates are mapped indicator functions of subsets of D.
 - ▶ $\Psi(\text{satisfied}) = p_{\text{satisfied}}$:
 $p_{\text{satisfied}}(\text{craig}) = \text{True}$
 - ▶ $\Psi(\text{privatebeach}) = p_{\text{privatebeach}}$:
 $p_{\text{privatebeach}}(\text{le-fleabag}) = \text{False}$
- ▶ 2-ary predicates are mapped to indicator functions over D^2
 - ▶ $\Psi(\text{location}) = p_{\text{location}}$:
 $p_{\text{location}}(\text{grandhotel}, \text{rome}) = \text{True}$
 $p_{\text{location}}(\text{grandhotel}, \text{sienna}) = \text{False}$
 - ▶ $\Psi(\text{available}) = p_{\text{available}}$:
 $p_{\text{available}}(\text{grandhotel}, \text{week29}) = \text{True}$
- ▶ n-ary predicates..

▶ 29

Fahiem Bacchus, University of Toronto

Semantics—Building up

Given language $L(F,P,V)$, and an interpretation
 $I = \langle D, \Phi, \Psi, v \rangle$

- a) Constant c (0-ary function) denotes an individual
 $I(c) = \Phi(c) \in D$
- b) Variable X denotes an individual
 $I(X) = v(X) \in D$ (variable assignment function).
- c) Ground term $t = f(t_1, \dots, t_k)$ denotes an individual
 $I(t) = \Phi(f)(I(t_1), \dots, I(t_k)) \in D$

We recursively find the denotation of each term, then we apply the function denoted by f to get a new individual.

Hence terms always denote individuals under an interpretation I

▶ 31

Fahiem Bacchus, University of Toronto

Intuitions: v

- ▶ v exists to take care of quantification. As we will see the exact mapping it specifies will not matter.
- ▶ Notation: $v[X/d]$ is a new variable assignment function.
 - ▶ Exactly like v , except that it maps the variable X to the individual d .
 - ▶ Maps every other variable exactly like v :
 $v(Y) = v[X/d](Y)$

▶ 30

Fahiem Bacchus, University of Toronto

Semantics—Building up

Formulas

- a) Ground atom $a = p(t_1, \dots, t_k)$ has truth value
 $I(a) = \Psi(p)(I(t_1), \dots, I(t_k)) \in \{\text{True, False}\}$

We recursively find the individuals denoted by the t_i , then we check to see if this tuple of individuals is in the relation denoted by p .

▶ 32

Fahiem Bacchus, University of Toronto

Semantics—Building up

Formulas

- b) Negated formulas $\neg f$ has truth value

$I(\neg f) = \text{True}$ if $I(f) = \text{False}$

$I(\neg f) = \text{False}$ if $I(f) = \text{True}$

- c) And formulas $f_1 \wedge f_2 \wedge \dots \wedge f_n$ have truth value

$I(f_1 \wedge f_2 \wedge \dots \wedge f_n) = \text{True}$ if every $I(f_i) = \text{True}$.

$I(f_1 \wedge f_2 \wedge \dots \wedge f_n) = \text{False}$ otherwise.

- d) Or formulas $f_1 \vee f_2 \vee \dots \vee f_n$ have truth value

$I(f_1 \vee f_2 \vee \dots \vee f_n) = \text{True}$ if any $I(f_i) = \text{True}$.

$I(f_1 \vee f_2 \vee \dots \vee f_n) = \text{False}$ otherwise.

▶ 33

Fahiem Bacchus, University of Toronto

Semantics—Building up

Formulas

- f) Universal formulas $\forall X.f$ have truth value

$I(\forall X.f) = \text{True}$ if for all $d \in D$

$I'(f) = \text{True}$

where $I' = \langle D, \Phi, \Psi, v[X/d] \rangle$

False otherwise.

Now "f" must be true of every individual "d".

Hence formulas are always either True or False under an interpretation I

▶ 35

Fahiem Bacchus, University of Toronto

Semantics—Building up

Formulas

- e) Existential formulas $\exists X. f$ have truth value

$I(\exists X. f) = \text{True}$ if there exists a $d \in D$ such that

$I'(f) = \text{True}$

where $I' = \langle D, \Phi, \Psi, v[X/d] \rangle$

False otherwise.

I' is just like I except that its variable assignment function now maps X to d . "d" is the individual of which "f" is true.

▶ 34

Fahiem Bacchus, University of Toronto

Example

$D = \{\underline{\text{bob}}, \underline{\text{jack}}, \underline{\text{fred}}\}$

happy is true of all objects.

$I(\forall X.\text{happy}(X))$

1. $\Psi(\text{happy})(v[X/\text{bob}](X)) = \Psi(\text{happy})(\text{bob}) = \text{True}$

2. $\Psi(\text{happy})(v[X/\text{jack}](X)) = \Psi(\text{happy})(\text{jack}) = \text{True}$

3. $\Psi(\text{happy})(v[X/\text{fred}](X)) = \Psi(\text{happy})(\text{fred}) = \text{True}$

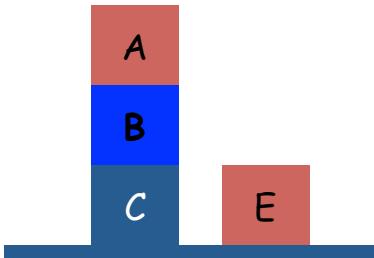
Therefore $I(\forall X.\text{happy}(X)) = \text{True}$.

▶ 36

Fahiem Bacchus, University of Toronto

Models—Examples.

Environment



Language (Syntax)

- **Constants:** a, b, c, e
- **Functions:**
 - No function
- **Predicates:**
 - on : binary
 - $above$: binary
 - $clear$: unary
 - $ontable$: unary

▶ 37

Fahiem Bacchus, University of Toronto

Models—Examples.

Language (syntax)

- **Constants:** a, b, c, e
- **Predicates:**
 - on (binary)
 - $above$ (binary)
 - $clear$ (unary)
 - $ontable$ (unary)

▶ 38

Fahiem Bacchus, University of Toronto

A possible Model I₁ (semantics)

- $D = \{\underline{A}, \underline{B}, \underline{C}, \underline{E}\}$
- $\Phi(a) = \underline{A}, \Phi(b) = \underline{B}, \Phi(c) = \underline{C}, \Phi(e) = \underline{E}$.
- $\Psi(on) = \{(\underline{A}, \underline{B}), (\underline{B}, \underline{C})\}$
- $\Psi(above) = \{(\underline{A}, \underline{B}), (\underline{B}, \underline{C}), (\underline{A}, \underline{C})\}$
- $\Psi clear) = \{\underline{A}, \underline{E}\}$
- $\Psi(ontable) = \{\underline{C}, \underline{E}\}$

Models—Examples.

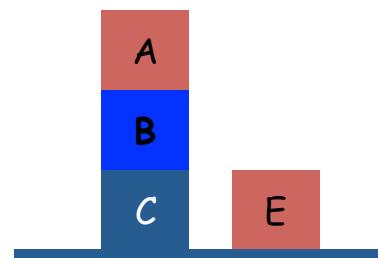
Model I₁

- $D = \{\underline{A}, \underline{B}, \underline{C}, \underline{E}\}$
- $\Phi(a) = \underline{A}, \Phi(b) = \underline{B}, \Phi(c) = \underline{C}, \Phi(e) = \underline{E}$.
- $\Psi(on) = \{(\underline{A}, \underline{B}), (\underline{B}, \underline{C})\}$
- $\Psi(above) = \{(\underline{A}, \underline{B}), (\underline{B}, \underline{C}), (\underline{A}, \underline{C})\}$
- $\Psi clear) = \{\underline{A}, \underline{E}\}$
- $\Psi(ontable) = \{\underline{C}, \underline{E}\}$

▶ 39

Fahiem Bacchus, University of Toronto

Environment



Models—Formulas true or false?

Model I₁

- $D = \{\underline{A}, \underline{B}, \underline{C}, \underline{E}\}$
- $\Phi(a) = \underline{A}, \Phi(b) = \underline{B}, \Phi(c) = \underline{C}, \Phi(e) = \underline{E}$.
- $\Psi(on) = \{(\underline{A}, \underline{B}), (\underline{B}, \underline{C})\}$
- $\Psi(above) = \{(\underline{A}, \underline{B}), (\underline{B}, \underline{C}), (\underline{A}, \underline{C})\}$
- $\Psi clear) = \{\underline{A}, \underline{E}\}$
- $\Psi(ontable) = \{\underline{C}, \underline{E}\}$

▶ 40

Fahiem Bacchus, University of Toronto

$$\forall X, Y. \text{on}(X, Y) \rightarrow \text{above}(X, Y)$$

$$X = \underline{A}, Y = \underline{B}$$

$$X = \underline{C}, Y = \underline{A}$$

...

$$\forall X, Y. \text{above}(X, Y) \rightarrow \text{on}(X, Y)$$

$$X = \underline{A}, Y = \underline{B}$$

$$X = \underline{A}, Y = \underline{C}$$

Models—Examples.

Model I₁

- $D = \{A, B, C, E\}$
- $\Phi(a) = A, \Phi(b) = B,$
 $\Phi(c) = C, \Phi(e) = E.$
- $\Psi(on) = \{(A,B), (B,C)\}$
- $\Psi(above) = \{(A,B),$
 $(B,C), (A,C)\}$
- $\Psi(clear) = \{A, E\}$
- $\Psi(ontable) = \{C, E\}$

$\forall X \exists Y. (\text{clear}(X) \vee \text{on}(Y,X))$

$X=A$

$X=C, Y=B$

...

$\exists Y \forall X. (\text{clear}(X) \vee \text{on}(Y,X))$

$Y=A ? \text{No! } (X=C)$

$Y=C ? \text{No! } (X=B)$

$Y=E ? \text{No! } (X=B)$

$Y=B ? \text{No! } (X=B)$

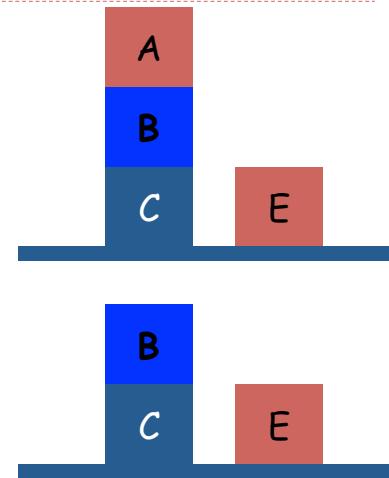
▶ 41

Fahiem Bacchus, University of Toronto

KB

1. $\text{on}(b,c)$

2. $\text{clear}(e)$



▶ 42

Fahiem Bacchus, University of Toronto

Models

- ▶ Let our Knowledge base KB, consist of a set of formulas.
- ▶ We say that I is a **model** of KB or that I **satisfies** KB
 - ▶ If, every formula $f \in \text{KB}$ is true under I
- ▶ We write $I \models \text{KB}$ if I satisfies KB, and $I \models f$ if f is true under I .

▶ 43

Fahiem Bacchus, University of Toronto

What's Special About Models?

- ▶ When we write KB, we intend that the real world (i.e. our set theoretic abstraction of it) is one of its models.
- ▶ This means that every statement in KB is **true** in the real world.
- ▶ Note however, that not every thing true in the real world need be contained in KB. We might have only incomplete knowledge.

▶ 44

Fahiem Bacchus, University of Toronto

Models support reasoning.

- ▶ Suppose formula f is not mentioned in KB, but is true in every model of KB; i.e.,
 $I \models KB \rightarrow I \models f$.
- ▶ Then we say that f is a **logical consequence** of KB or that KB **entails** f .
- ▶ Since the real world is a model of KB, f must be true in the real world.
- ▶ This means that entailment is a way of finding new true facts that were not explicitly mentioned in KB.

??? If KB doesn't entail f , is f false in the real world?

▶ 45

Fahiem Bacchus, University of Toronto

Logical Consequence Example

- ▶ $\forall X, Y. \text{elephant}(X) \wedge \text{teacup}(Y) \rightarrow \text{largerThan}(X, Y)$

- ▶ For all pairs of individuals if the first is an elephant and the second is a teacup, then the pair of objects are related to each other by the *largerThan* relation.
- ▶ For pairs of individuals who are not elephants and teacups, the formula is immediately true.

▶ 47

Fahiem Bacchus, University of Toronto

Logical Consequence Example

- ▶ **elephant(clyde)**
 - ▶ the individual denoted by the symbol *clyde* in the set denoted by *elephant* (has the property that it is an elephant).
- ▶ **teacup(cup)**
 - ▶ *cup* is a teacup.
- ▶ Note that in both cases a unary predicate specifies a set of individuals. Asserting a unary predicate to be true of a term means that the individual denoted by that term is in the specified set.
 - ▶ Formally, we map individuals to TRUE/FALSE (this is an indicator function for the set).

▶ 46

Fahiem Bacchus, University of Toronto

Logical Consequence Example

- ▶ $\forall X, Y. \text{largerThan}(X, Y) \rightarrow \neg \text{fitsIn}(X, Y)$

- ▶ For all pairs of individuals if X is larger than Y (the pair is in the *largerThan* relation) then we cannot have that X fits in Y (the pair cannot be in the *fitsIn* relation).
- ▶ (The relation *largerThan* has an empty intersection with the *fitsIn* relation).

▶ 48

Fahiem Bacchus, University of Toronto

Logical Consequences

- ▶ $\neg \text{fitsIn}(\text{clyde}, \text{cup})$
- ▶ We know $\text{largerThan}(\text{clyde}, \text{teacup})$ from the first implication. Thus we know this from the second implication.

▶ 49

Fahiem Bacchus, University of Toronto

Logical Consequence Example

- ▶ If an interpretation satisfies KB, then the set of pairs *elephant X teacup* must be a subset of *largerThan*, which is disjoint from *fitsIn*.
- ▶ Therefore, the pair $(\text{clyde}, \text{cup})$ must be in the complement of the set *fitsIn*.
- ▶ Hence, $\neg \text{fitsIn}(\text{clyde}, \text{cup})$ must be true in every interpretation that satisfies KB.
- ▶ $\neg \text{fitsIn}(\text{clyde}, \text{cup})$ is a logical consequence of KB.

▶ 51

Fahiem Bacchus, University of Toronto

Logical Consequences

fitsIn

$\neg \text{fitsIn}$

largerThan

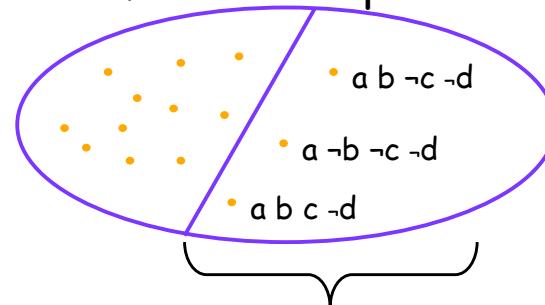
Elephants \times teacups
 $(\text{clyde}, \text{cup})$

▶ 50

Fahiem Bacchus, University of Toronto

Models Graphically

Set of All Interpretations



Models of KB

Consequences? $a, c \rightarrow b, b \rightarrow c, d \rightarrow b, \neg b \rightarrow \neg c$

▶ 52

Fahiem Bacchus, University of Toronto

Models and Interpretations

- ▶ the more sentences in KB, the fewer models (satisfying interpretations) there are.
- ▶ The more you write down (as long as it's all true!), the “closer” you get to the “real world”! Because Each sentence in KB rules out certain unintended interpretations.
- ▶ This is called **axiomatizing the domain**

▶ 53

Fahiem Bacchus, University of Toronto

Proof Procedures

- ▶ Interesting, proof procedures work by simply manipulating formulas. They do not know or care anything about interpretations.
- ▶ Nevertheless **they respect the semantics of interpretations!**
- ▶ We will develop a proof procedure for first-order logic called resolution.
 - ▶ **Resolution** is the mechanism used by PROLOG

▶ 55

Fahiem Bacchus, University of Toronto

Computing logical consequences

- ▶ We want procedures for computing logical consequences that can be implemented in our programs.
- ▶ This would allow us to reason with our knowledge
 - ▶ Represent the knowledge as logical formulas
 - ▶ Apply procedures for generating logical consequences
- ▶ These procedures are called **proof procedures**.

▶ 54

Fahiem Bacchus, University of Toronto

Properties of Proof Procedures

- ▶ Before presenting the details of resolution, we want to look at properties we would like to have in a (any) proof procedure.
- ▶ We write **$KB \vdash f$** to indicate that f can be proved from KB (the proof procedure used is implicit).

▶ 56

Fahiem Bacchus, University of Toronto

Properties of Proof Procedures

▶ Soundness

- ▶ $\text{KB} \vdash f \rightarrow \text{KB} \vDash f$

i.e. all conclusions arrived at via the proof procedure are correct: they are logical consequences.

▶ Completeness

- ▶ $\text{KB} \vDash f \rightarrow \text{KB} \vdash f$

i.e. every logical consequence can be generated by the proof procedure.

- ▶ Note proof procedures are computable, but they might have very high complexity in the worst case. So completeness is not necessarily achievable in practice.

▶ 57

Fahiem Bacchus, University of Toronto

Resolution

▶ Prolog Programs

- ▶ Prolog programs are clausal theories.
- ▶ However, each clause in a Prolog program is Horn.
- ▶ A horn clause contains at most one positive literal.

- ▶ The horn clause

$$\neg q_1 \vee \neg q_2 \vee \dots \vee \neg q_n \vee p$$

is equivalent to

$$q_1 \wedge q_2 \wedge \dots \wedge q_n \Rightarrow p$$

and is written as the following rule in Prolog:

$p :- q_1, q_2, \dots, q_n$

▶ 59

Fahiem Bacchus, University of Toronto

Resolution

▶ Clausal form.

- ▶ Resolution works with formulas expressed in clausal form.

- ▶ A **literal** is an atomic formula or the negation of an atomic formula. dog(fido) , $\neg\text{cat(fido)}$

- ▶ A **clause** is a disjunction of literals:

- ▶ $\neg\text{owns(fido,fred)} \vee \neg\text{dog(fido)} \vee \text{person(fred)}$

- ▶ We write
 $(\neg\text{owns(fido,fred)}, \neg\text{dog(fido)}, \text{person(fred)})$

- ▶ A **clausal theory** is a conjunction of clauses.

▶ 58

Fahiem Bacchus, University of Toronto

Resolution Rule for Ground Clauses

- ▶ The resolution proof procedure consists of only one simple rule:

▶ From the two clauses

- ▶ $(P, Q_1, Q_2, \dots, Q_k)$

- ▶ $(\neg P, R_1, R_2, \dots, R_n)$

▶ We infer the new clause

- ▶ $(Q_1, Q_2, \dots, Q_k, R_1, R_2, \dots, R_n)$

▶ Example:

- ▶ $(\neg\text{largerThan(clyde,cup)}, \neg\text{fitsIn(clyde,cup)})$

- ▶ $(\text{fitsIn(clyde,cup)})$

$$\Rightarrow \neg\text{largerThan(clyde,cup)}$$

▶ 60

Fahiem Bacchus, University of Toronto

Resolution Proof: Forward chaining

- ▶ Logical consequences can be generated from the resolution rule in two ways:

1. Forward Chaining inference.

- ▶ If we have a sequence of clauses C_1, C_2, \dots, C_k
- ▶ Such that each C_i is either in KB or is the result of a resolution step involving two prior clauses in the sequence.
- ▶ We then have that $KB \vdash C_k$.

Forward chaining is sound so we also have $KB \vDash C_k$

▶ 61

Fahiem Bacchus, University of Toronto

Resolution Proof: Refutation proofs

- ▶ If we can find such a sequence $C_1, C_2, \dots, C_m = ()$, we have that
 - ▶ $KB \vdash f$.
 - ▶ Furthermore, this procedure is sound so
 - ▶ $KB \vDash f$
- ▶ And the procedure is also complete so it is capable of finding a proof of any f that is a logical consequence of KB. i.e.
 - ▶ If $KB \vDash f$ then we can generate a refutation from $KB \wedge \neg f$

▶ 63

Fahiem Bacchus, University of Toronto

Resolution Proof: Refutation proofs

2. Refutation proofs.

- ▶ We determine if $KB \vdash f$ by showing that a contradiction can be generated from $KB \wedge \neg f$.
- ▶ In this case a contradiction is an empty clause $()$.
- ▶ We employ resolution to construct a sequence of clauses C_1, C_2, \dots, C_m such that
 - C_i is in $KB \wedge \neg f$, or is the result of resolving two previous clauses in the sequence.
 - $C_m = ()$ i.e. its the empty clause.

▶ 62

Fahiem Bacchus, University of Toronto

Resolution Proofs Example

Want to prove `likes(clyde,peanuts)` from:

1. `(elephant(clyde), giraffe(clyde))`
2. `(\neg elephant(clyde), likes(clyde,peanuts))`
3. `(\neg giraffe(clyde), likes(clyde,leaves))`
4. `\neg likes(clyde,leaves)`

Forward Chaining Proof:

- ▶ 3&4 → `\neg giraffe(clyde)` [5.]
- ▶ 5&1 → `elephant(clyde)` [6.]
- ▶ 6&2 → `likes(clyde,peanuts)` [7.] ✓

▶ 64

Fahiem Bacchus, University of Toronto

Resolution Proofs Example

1. (elephant(clyde), giraffe(clyde))
2. (\neg elephant(clyde), likes(clyde,peanuts))
3. (\neg giraffe(clyde), likes(clyde,leaves))
4. \neg likes(clyde,leaves)

Refutation Proof:

- ▶ \neg likes(clyde,peanuts) [5.]
- ▶ 5&2 \rightarrow \neg elephant(clyde) [6.]
- ▶ 6&1 \rightarrow giraffe(clyde) [7.]
- ▶ 7&3 \rightarrow likes(clyde,leaves) [8.]
- ▶ 8&4 \rightarrow () ✓

▶ 65

Fahiem Bacchus, University of Toronto

Conversion to Clausal Form

To convert the KB into Clausal form we perform the following 8-step procedure:

1. **Eliminate Implications.**
2. **Move Negations inwards (and simplify $\neg\neg$).**
3. **Standardize Variables.**
4. **Skolemize.**
5. **Convert to Prenex Form.**
6. **Distribute conjunctions over disjunctions.**
7. **Flatten nested conjunctions and disjunctions.**
8. **Convert to Clauses.**

▶ 67

Fahiem Bacchus, University of Toronto

Resolution Proofs

- ▶ Proofs by refutation have the advantage that they are easier to find.
 - ▶ They are more focused to the particular conclusion we are trying to reach.
- ▶ To develop a complete resolution proof procedure for First-Order logic we need :
 1. A way of converting KB and f (the query) into clausal form.
 2. A way of doing resolution even when we have variables (unification).

▶ 66

Fahiem Bacchus, University of Toronto

C-T-C-F: Eliminate implications

We use this example to show each step:

$$\forall X.p(X) \rightarrow ((\forall Y.p(Y) \rightarrow p(f(X,Y))) \wedge \neg(\forall Y.\neg q(X,Y) \wedge p(Y)))$$

1. Eliminate implications: $A \rightarrow B \rightarrow \neg A \vee B$

$$\forall X. \neg p(X) \vee ((\forall Y. \neg p(Y) \vee p(f(X,Y))) \wedge \neg(\forall Y. \neg q(X,Y) \wedge p(Y)))$$

▶ 68

Fahiem Bacchus, University of Toronto

C-T-C-F: Move \neg Inwards

$$\forall X. \neg p(X) \\ \vee \left((\forall Y. \neg p(Y) \vee p(f(X,Y))) \wedge \neg(\exists Y. q(X,Y) \wedge p(Y)) \right)$$

2. Move Negations Inwards (and simplify $\neg\neg$)

$$\forall X. \neg p(X) \\ \vee \left((\forall Y. \neg p(Y) \vee p(f(X,Y))) \wedge (\exists Y. q(X,Y) \vee \neg p(Y)) \right)$$

▶ 69

Fahiem Bacchus, University of Toronto

C-T-C-F: Standardize Variables

$$\forall X. \neg p(X) \\ \vee \left((\forall Y. \neg p(Y) \vee p(f(X,Y))) \wedge (\exists Y. q(X,Y) \vee \neg p(Y)) \right)$$

3. Standardize Variables (Rename variables so that each quantified variable is unique)

$$\forall X. \neg p(X) \\ \vee \left((\forall Y. (\neg p(Y) \vee p(f(X,Y))) \wedge (\exists Z. q(X,Z) \vee \neg p(Z)) \right)$$

▶ 71

Fahiem Bacchus, University of Toronto

C-T-C-F: : \neg continue...

Rules for moving negations inwards

- ▶ $\neg(A \wedge B) \rightarrow \neg A \vee \neg B$
- ▶ $\neg(A \vee B) \rightarrow \neg A \wedge \neg B$
- ▶ $\neg \forall X. f \rightarrow \exists X. \neg f$
- ▶ $\neg \exists X. f \rightarrow \forall X. \neg f$
- ▶ $\neg \neg A \rightarrow A$

▶ 70

Fahiem Bacchus, University of Toronto

C-T-C-F: Skolemize

$$\forall X. \neg p(X) \\ \vee \left((\forall Y. \neg p(Y) \vee p(f(X,Y))) \wedge (\exists Z. q(X,Z) \vee \neg p(Z)) \right)$$

4. Skolemize (Remove existential quantifiers by introducing new function symbols).

$$\forall X. \neg p(X) \\ \vee \left((\forall Y. \neg p(Y) \vee p(f(X,Y))) \wedge (q(X, g(X)) \vee \neg p(g(X))) \right)$$

▶ 72

Fahiem Bacchus, University of Toronto

C-T-C-F: Skolemization

Consider $\exists Y.\text{elephant}(Y) \wedge \text{friendly}(Y)$

- ▶ This asserts that there is some individual (binding for Y) that is both an elephant and friendly.
- ▶ To remove the existential, we **invent** a name for this individual, say **a**. This is a new constant symbol **not equal to any previous constant symbols** to obtain:
 $\text{elephant}(a) \wedge \text{friendly}(a)$
- ▶ This is saying the same thing, since we do not know anything about the new constant **a**.

▶ 73

Fahiem Bacchus, University of Toronto

C-T-C-F: Skolemization

Now consider $\forall X \exists Y. \text{loves}(X, Y)$.

- ▶ This formula claims that for every X there is some Y that X loves (perhaps a different Y for each X).
- ▶ Replacing the existential by a new constant won't work
 $\forall X. \text{loves}(X, a)$.

Because this asserts that there is a **particular** individual "a" loved by every X.

- ▶ To properly convert existential quantifiers scoped by universal quantifiers we must use **functions** not just constants.

▶ 75

Fahiem Bacchus, University of Toronto

C-T-C-F: Skolemization

- ▶ It is essential that the introduced symbol "a" is **new**. Else we might know something else about "a" in KB.
- ▶ If we did know something else about "a" we would be asserting more than the existential.
- ▶ In original quantified formula we know nothing about the variable "Y". Just what was being asserted by the existential formula.

▶ 74

Fahiem Bacchus, University of Toronto

C-T-C-F: Skolemization

- ▶ We must use a function that mentions **every universally quantified variable that scopes the existential**.
- ▶ In this case X scopes Y so we must replace the existential Y by a function of X
 $\forall X. \text{loves}(X, g(X))$,
where g is a **new** function symbol.
- ▶ This formula asserts that for every X there is some individual (given by g(X)) that X loves. g(X) can be different for each different binding of X.

▶ 76

Fahiem Bacchus, University of Toronto

C-T-C-F: Skolemization Examples

- ▶ $\forall XYZ \exists W.r(X,Y,Z,W) \rightarrow \forall XYZ.r(X,Y,Z,h1(X,Y,Z))$

- ▶ $\forall XY \exists W.r(X,Y,g(W)) \rightarrow \forall XY.r(X,Y,Z,g(h2(X,Y)))$

- ▶ $\forall XY \exists W \forall Z.r(X,Y,W) \wedge q(Z,W)$
 $\rightarrow \forall XYZ.r(X,Y,h3(X,Y)) \wedge q(Z,h3(X,Y))$

▶ 77

Fahiem Bacchus, University of Toronto

C-T-C-F: Conjunctions over disjunctions

$$\begin{aligned} & \forall X \forall Y. \neg p(X) \\ & \quad \vee ((\neg p(Y) \vee p(f(X,Y))) \\ & \quad \quad \wedge (q(X,g(X)) \vee \neg p(g(X)))) \end{aligned}$$

6. Conjunctions over disjunctions

$$A \vee (B \wedge C) \rightarrow (A \vee B) \wedge (A \vee C)$$

$$\begin{aligned} & \forall X \forall Y. (\neg p(X) \vee \neg p(Y) \vee p(f(X,Y))) \\ & \quad \wedge (\neg p(X) \vee q(X,g(X)) \vee \neg p(g(X))) \end{aligned}$$

▶ 79

Fahiem Bacchus, University of Toronto

C-T-C-F: Convert to prefix

$$\begin{aligned} & \forall X. \neg p(X) \\ & \quad \vee ((\forall Y. \neg p(Y) \vee p(f(X,Y))) \\ & \quad \quad \wedge q(X,g(X)) \vee \neg p(g(X))) \end{aligned}$$

5. Convert to prefix form. (Bring all quantifiers to the front—only universals, each with different name).

$$\begin{aligned} & \forall X \forall Y. \neg p(X) \\ & \quad \vee ((\neg p(Y) \vee p(f(X,Y))) \\ & \quad \quad \wedge q(X,g(X)) \vee \neg p(g(X))) \end{aligned}$$

▶ 78

Fahiem Bacchus, University of Toronto

C-T-C-F: flatten & convert to clauses

7. Flatten nested conjunctions and disjunctions.

$$(A \vee (B \vee C)) \rightarrow (A \vee B \vee C)$$

8. Convert to Clauses

(remove quantifiers and break apart conjunctions).

$$\begin{aligned} & \forall X \forall Y. (\neg p(X) \vee \neg p(Y) \vee p(f(X,Y))) \\ & \quad \wedge (\neg p(X) \vee q(X,g(X)) \vee \neg p(g(X))) \end{aligned}$$

- a) $\neg p(X) \vee \neg p(Y) \vee p(f(X,Y))$
- b) $\neg p(X) \vee q(X,g(X)) \vee \neg p(g(X))$

▶ 80

Fahiem Bacchus, University of Toronto

Unification

- ▶ Ground clauses are clauses with no variables in them. For ground clauses we can use syntactic identity to detect when we have a P and $\neg P$ pair.
- ▶ What about variables can the clauses
 - ▶ $(P(john), Q(fred), R(X))$
 - ▶ $(\neg P(Y), R(susan), R(Y))$Be resolved?

▶ 81

Fahiem Bacchus, University of Toronto

Unification.

- ▶ We want to be able to match conflicting literals, even when they have variables. This matching process automatically determines whether or not there is a “specialization” that matches.
- ▶ We don’t want to over specialize!

▶ 83

Fahiem Bacchus, University of Toronto

Unification.

- ▶ Intuitively, once reduced to clausal form, all remaining variables are universally quantified. So, implicitly $(\neg P(Y), R(susan), R(Y))$ represents a whole set of ground clauses like
 - ▶ $(\neg P(fred), R(susan), R(fred))$
 - ▶ $(\neg P(john), R(susan), R(john))$
 - ▶ ...
- ▶ So there is a “specialization” of this clause that can be resolved with $(P(john), Q(fred), R(X))$

▶ 82

Fahiem Bacchus, University of Toronto

Unification.

- ▶ $(\neg p(X), s(X), q(fred))$
- ▶ $(p(Y), r(Y))$
- ▶ Possible resolvants
 - ▶ $(s(john), q(fred), r(john)) \{Y=X, X=john\}$
 - ▶ $(s(sally), q(fred), r(sally)) \{Y=X, X=sally\}$
 - ▶ $(s(X), q(fred), r(X)) \{Y=X\}$
- ▶ The last resolvent is “most-general”, the other two are specializations of it.
- ▶ We want to keep the most general clause so that we can use it future resolution steps.

▶ 84

Fahiem Bacchus, University of Toronto

Unification.

- ▶ **unification** is a mechanism for finding a “most general” matching.
- ▶ First we consider **substitutions**.
 - ▶ A substitution is a finite set of equations of the form

$$V = t$$

where V is a variable and t is a term not containing V . (t might contain other variables).

▶ 85

Fahiem Bacchus, University of Toronto

Substitutions.

- ▶ We can compose two substitutions. θ and σ to obtain a new substitution $\theta\sigma$.

Let $\theta = \{X_1=s_1, X_2=s_2, \dots, X_m=s_m\}$
 $\sigma = \{Y_1=t_1, Y_2=t_2, \dots, Y_k=s_k\}$

To compute $\theta\sigma$

1. $S = \{X_1=s_1\sigma, X_2=s_2\sigma, \dots, X_m=s_m\sigma, Y_1=t_1, Y_2=t_2, \dots, Y_k=s_k\}$

we apply σ to each RHS of θ and then add all of the equations of σ .

▶ 87

Fahiem Bacchus, University of Toronto

Substitutions.

- ▶ We can **apply a substitution σ** to a formula f to obtain a new formula $f\sigma$ by simultaneously replacing every variable mentioned in the left hand side of the substitution by the right hand side.

$$p(X, g(Y, Z)) [X=Y, Y=f(a)] \rightarrow p(Y, g(f(a), Z))$$

- ▶ Note that the substitutions are not applied sequentially, i.e., the first Y is not subsequently replaced by $f(a)$.

▶ 86

Fahiem Bacchus, University of Toronto

Substitutions.

1. $S = \{X_1=s_1\sigma, X_2=s_2\sigma, \dots, X_m=s_m\sigma, Y_1=t_1, Y_2=t_2, \dots, Y_k=s_k\}$
2. Delete any identities, i.e., equations of the form $V=V$.
3. Delete any equation $Y_i=s_i$ where Y_i is equal to one of the X_j in θ .

The final set S is the composition $\theta\sigma$.

▶ 88

Fahiem Bacchus, University of Toronto

Composition Example.

$$\theta = \{X=f(Y), Y=Z\}, \sigma = \{X=a, Y=b, Z=Y\}$$

$\theta\sigma$

▶ 89

Fahiem Bacchus, University of Toronto

Unifiers.

- ▶ A **unifier** of two formulas f and g is a substitution σ that makes f and g **syntactically identical**.
- ▶ Not all formulas can be unified—substitutions only affect variables.

$$p(f(X),a) \quad p(Y,f(w))$$

- ▶ This pair cannot be unified as there is no way of making $a = f(w)$ with a substitution.
- ▶ Note we typically use UPPER CASE to denote variables, lower case for constants.

▶ 91

Fahiem Bacchus, University of Toronto

Substitutions.

- ▶ The empty substitution $\varepsilon = \{\}$ is also a substitution, and it acts as an identity under composition.
- ▶ More importantly substitutions when applied to formulas are associative:

$$(f\theta)\sigma = f(\theta\sigma)$$

- ▶ Composition is simply a way of converting the sequential application of a series of substitutions to a single simultaneous substitution.

▶ 90

Fahiem Bacchus, University of Toronto

MGU.

- ▶ A substitution σ of two formulas f and g is a **Most General Unifier (MGU)** if
 1. σ is a unifier.
 2. For every other unifier θ of f and g there must exist a third substitution λ such that

$$\theta = \sigma\lambda$$

- ▶ This says that every other unifier is “more specialized than σ . The MGU of a pair of formulas f and g is unique up to renaming.

▶ 92

Fahiem Bacchus, University of Toronto

MGU.

$$p(f(X), Z) \quad p(Y, a)$$

1. $\sigma = \{Y = f(a), X=a, Z=a\}$ is a unifier.

$$\begin{aligned} p(f(X), Z)\sigma &= \\ p(Y, a)\sigma &= \end{aligned}$$

But it is not an MGU.

2. $\theta = \{Y=f(X), Z=a\}$ is an MGU.

$$\begin{aligned} p(f(X), Z) \theta &= \\ p(Y, a) \theta &= \end{aligned}$$

▶ 93

Fahiem Bacchus, University of Toronto

MGU.

$$p(f(X), Z) \quad p(Y, a)$$

3. $\sigma = \theta\lambda$, where $\lambda=\{X=a\}$

$$\begin{aligned} \sigma &= \{Y = f(a), X=a, Z=a\} \\ \lambda &= \{X=a\} \\ \theta\lambda &= \end{aligned}$$

▶ 94

Fahiem Bacchus, University of Toronto

MGU.

- ▶ The MGU is the “least specialized” way of making clauses with universal variables match.
- ▶ We can compute MGUs.
- ▶ Intuitively we line up the two formulas and find the first sub-expression where they disagree. The pair of subexpressions where they **first** disagree is called the **disagreement set**.
- ▶ The algorithm works by successively fixing disagreement sets until the two formulas become syntactically identical.

▶ 95

Fahiem Bacchus, University of Toronto

MGU.

To find the MGU of two formulas f and g .

1. $k = 0; \sigma_0 = \{\}; S_0 = \{f, g\}$
2. If S_k contains an identical pair of formulas stop, and return σ_k as the MGU of f and g .
3. Else find the disagreement set $D_k = \{e_1, e_2\}$ of S_k
4. If $e_1 = V$ a variable, and $e_2 = t$ a term not containing V (or vice-versa) then let

$$\sigma_{k+1} = \sigma_k \{V=t\} \quad (\text{Compose the additional substitution})$$

$$S_{k+1} = S_k \{V=t\} \quad (\text{Apply the additional substitution})$$

$k = k+1$
GOTO 2

5. Else stop, f and g cannot be unified.

▶ 96

Fahiem Bacchus, University of Toronto

MGU Example 1.

$S_0 = \{p(f(a), g(X)) ; p(Y, Y)\}$

▶ 97

Fahiem Bacchus, University of Toronto

MGU Example 2.

$S_0 = \{p(a, X, h(g(Z))) ; p(Z, h(Y), h(Y))\}$

▶ 98

Fahiem Bacchus, University of Toronto

MGU Example 3.

$S_0 = \{p(X, X) ; p(Y, f(Y))\}$

▶ 99

Fahiem Bacchus, University of Toronto

Non-Ground Resolution

- ▶ Resolution of non-ground clauses. From the two clauses

$(L, Q_1, Q_2, \dots, Q_k)$
 $(\neg M, R_1, R_2, \dots, R_n)$

Where there exists σ a MGU for L and M.

We infer the new clause

$(Q_1\sigma, \dots, Q_k\sigma, R_1\sigma, \dots, R_n\sigma)$

▶ 100

Fahiem Bacchus, University of Toronto

Non-Ground Resolution E.G.

1. $(p(X), q(g(X)))$
2. $(r(a), q(Z), \neg p(a))$

$L = p(X); M = p(a)$
 $\sigma = \{X = a\}$

3. $R[1a, 2c]\{\{X = a\} (q(g(a)), r(a), q(Z))$

The notation is important.

- ▶ “R” means resolution step.
- ▶ “1a” means the **first** (**a**-th) literal in the first clause i.e. $p(X)$.
- ▶ “2c” means the **third** (**c**-th) literal in the second clause, $\neg p(a)$.
 - ▶ 1a and 2c are the “clashing” literals.
- ▶ $\{X = a\}$ is the substitution applied to make the clashing literals identical.

▶ 101

Fahiem Bacchus, University of Toronto

Resolution Proof Example

“Some patients like all doctors. No patient likes any quack. Therefore no doctor is a quack.”

Resolution Proof Step 1.

Pick symbols to represent these assertions.

$p(X)$: X is a patient
 $d(x)$: X is a doctor
 $q(X)$: X is a quack
 $I(X, Y)$: X likes Y

▶ 102

Fahiem Bacchus, University of Toronto

Resolution Proof Example

Resolution Proof Step 2.

Convert each assertion to a first-order formula.

1. Some patients like all doctors.

F1.

Resolution Proof Example

2. No patient likes any quack

F2.

3. Therefore no doctor is a quack.
Query.

▶ 103

Fahiem Bacchus, University of Toronto

▶ 104

Fahiem Bacchus, University of Toronto

Resolution Proof Example

Resolution Proof Step 3.

Convert to Clausal form.

F1.

F2.

Negation of Query.

▶ 105

Fahiem Bacchus, University of Toronto

Resolution Proof Example

Resolution Proof Step 4.

Resolution Proof from the Clauses.

1. $p(a)$
2. $(\neg d(Y), l(a,Y))$
3. $(\neg p(Z), \neg q(R), \neg l(Z,R))$
4. $d(b)$
5. $q(b)$

▶ 106

Fahiem Bacchus, University of Toronto

Answer Extraction.

- ▶ The previous example shows how we can answer true-false questions. With a bit more effort we can also answer “fill-in-the-blanks” questions (e.g., what is wrong with the car?).
- ▶ As in Prolog we use free variables in the query where we want the fill in the blanks. We simply need to keep track of the binding that these variables received in proving the query.
 - ▶ $\text{parent}(\text{art}, \text{jon})$ – is art one of jon's parents?
 - ▶ $\text{parent}(X, \text{jon})$ – who is one of jon's parents?

▶ 107

Fahiem Bacchus, University of Toronto

Answer Extraction.

- ▶ A simple bookkeeping device is to use an predicate symbol $\text{answer}(X, Y, \dots)$ to keep track of the bindings automatically.
- ▶ To answer the query $\text{parent}(X, \text{jon})$, we construct the clause
 $(\neg \text{parent}(X, \text{jon}), \text{answer}(X))$
- ▶ Now we perform resolution until we obtain a clause consisting of only answer literals (previously we stopped at empty clauses).

▶ 108

Fahiem Bacchus, University of Toronto

Answer Extraction: Example 1

1. father(art, jon)
2. father(bob,kim)
3. $(\neg \text{father}(Y,Z), \text{parent}(Y,Z))$
i.e. all fathers are parents
4. $(\neg \text{parent}(X,jon), \text{answer}(X))$
i.e. the query is: who is parent of jon?

Here is a resolution proof:

5. $R[4,3b]\{Y=X, Z=jon\} (\neg \text{father}(X,jon), \text{answer}(X))$
6. $R[5,1]\{X=art\} \text{answer}(art)$
so art is parent of jon

▶ 109

Fahiem Bacchus, University of Toronto

Factoring

1. $(p(X), p(Y))$ // $\forall X. \forall Y. \neg p(X) \rightarrow p(Y)$
2. $(\neg p(V), \neg p(W))$ // $\forall V. \forall W. p(V) \rightarrow \neg p(W)$

▶ These clauses are intuitively contradictory, but following the strict rules of resolution only we obtain:

3. $R[1a,2a](X=V) (p(Y), \neg p(W))$
Renaming variables: $(p(Q), \neg p(Z))$
4. $R[3b,1a](X=Z) (p(Y), p(Q))$

No way of generating empty clause!

Factoring is needed to make resolution over non-ground clauses complete, without it resolution is incomplete!

▶ 111

Fahiem Bacchus, University of Toronto

Answer Extraction: Example 2

1. $(\text{father}(art, jon), \text{father}(bob, jon))$ // either bob or art is parent of jon
2. $\text{father}(bob, kim)$
3. $(\neg \text{father}(Y,Z), \text{parent}(Y,Z))$ // i.e. all fathers are parents
4. $(\neg \text{parent}(X,jon), \text{answer}(X))$ // i.e. query is parent(X,jon)

Here is a resolution proof:

5. $R[4,3b]\{Y=X, Z=jon\} (\neg \text{father}(X,jon), \text{answer}(X))$
6. $R[5,1a]\{X=art\} (\text{father}(bob, jon), \text{answer}(art))$
7. $R[6,3b]\{Y=bob, Z=jon\} (\text{parent}(bob, jon), \text{answer}(art))$
8. $R[7,4]\{X=bob\} (\text{answer}(bob), \text{answer}(art))$

A disjunctive answer: either bob or art is parent of jon.

▶ 110

Fahiem Bacchus, University of Toronto

Factoring.

- ▶ If two or more literals of a clause C have an mgu θ , then $C\theta$ with all duplicate literals removed is called a **factor** of C.
- ▶ $C = (p(X), p(f(Y)), \neg q(X))$
 $\theta = \{X=f(Y)\}$
 $C\theta = (p(f(Y)), p(f(Y)), \neg q(f(Y))) \rightarrow (p(f(Y)), \neg q(f(Y)))$ is a factor

Adding a factor of a clause can be a step of proof:

1. $(p(X), p(Y))$
2. $(\neg p(V), \neg p(W))$
3. $f[1ab]\{X=Y\} p(Y)$
4. $f[2ab]\{V=W\} \neg p(W)$
5. $R[3,4]\{Y=W\} ()$.

▶ 112

Fahiem Bacchus, University of Toronto

Prolog

- ▶ Prolog search mechanism is simply an instance of resolution, except
 1. Clauses are Horn (only one positive literal)
 2. Prolog uses a specific depth first strategy when searching for a proof. (Rules are used first mentioned first used, literals are resolved away left to right).

▶ 113

Fahiem Bacchus, University of Toronto

Prolog: Example of proof

- ▶ Try to prove : append([a,b], [c,d], [a,b,c,d]):
 1. append([], Z, Z)
 2. (append([E1 | R1], Y, [E1 | Rest]),
 ¬append(R1, Y, Rest))
 3. ¬append([a,b], [c,d], [a,b,c,d])
 4. R[3,2a]{E1=a, R1=[b], Y=[c,d], Rest=[b,c,d]}
 ¬append([b], [c,d], [b,c,d])
 5. R[4,2a]{E1=b, R1=[], Y=[c,d], Rest=[c,d]}
 ¬append([], [c,d], [c,d])
 6. R[5,1]{Z=[c,d]} ()

▶ 115

Fahiem Bacchus, University of Toronto

Prolog

- ▶ Append:
 1. append([], Z, Z)
 2. append([E1 | R1], Y, [E1 | Rest]) :-
 append(R1, Y, Rest)

Note:

- The second is actually the clause
(append([E1 | R1], Y, [E1 | Rest]) , ¬append(R1, Y, Rest))
- [] is a constant (the empty list)
- [X | Y] is cons(X,Y).
- So [a,b,c] is short hand for cons(a,cons(b,cons(c,[])))

▶ 114

Fahiem Bacchus, University of Toronto

Review: One Last Example!

Consider the following English description

- ▶ Whoever can read is literate.
 - ▶ Dolphins are not literate.
 - ▶ Flipper is an intelligent dolphin.
-
- ▶ Who is intelligent but cannot read.

▶ 116

Fahiem Bacchus, University of Toronto

Example: pick symbols & convert to first-order formula

- ▶ Whoever can read is literate.
 $\forall X. \text{read}(X) \rightarrow \text{lit}(X)$
- ▶ Dolphins are not literate.
 $\forall X. \text{dolph}(X) \rightarrow \neg \text{lit}(X)$
- ▶ Flipper is an intelligent dolphin
 $\text{dolph}(\text{flipper}) \wedge \text{intell}(\text{flipper})$
- ▶ Who is intelligent but cannot read?
 $\exists X. \text{intell}(X) \wedge \neg \text{read}(X).$

▶ 117

Fahiem Bacchus, University of Toronto

Example: convert to clausal form

- ▶ $\forall X. \text{read}(X) \rightarrow \text{lit}(X)$
 $(\neg \text{read}(X), \text{lit}(X))$
- ▶ Dolphins are not literate.
 $\forall X. \text{dolph}(X) \rightarrow \neg \text{lit}(X)$
 $(\neg \text{dolph}(X), \neg \text{lit}(X))$
- ▶ Flipper is an intelligent dolphin.
 $\text{dolph}(\text{flipper})$
 $\text{intell}(\text{flipper})$
- ▶ who are intelligent but cannot read?
 $\exists X. \text{intell}(X) \wedge \neg \text{read}(X).$
→ $\forall X. \neg \text{intell}(X) \vee \text{read}(X)$
→ $(\neg \text{intell}(X), \text{read}(X), \text{answer}(X))$

▶ 118

Fahiem Bacchus, University of Toronto

Example: do the resolution proof

1. $(\neg \text{read}(X), \text{lit}(X))$
2. $(\neg \text{dolph}(X), \neg \text{lit}(X))$
3. $\text{dolph}(\text{flip})$
4. $\text{intell}(\text{flip})$
5. $(\neg \text{intell}(X), \text{read}(X), \text{answer}(X))$
6. R[5a,4] $X=\text{flip}$. $(\text{read}(\text{flip}), \text{answer}(\text{flip}))$
7. R[6,1a] $X=\text{flip}$. $(\text{lit}(\text{flip}), \text{answer}(\text{flip}))$
8. R[7,2b] $X=\text{flip}$. $(\neg \text{dolph}(\text{flip}), \text{answer}(\text{flip}))$
9. R[8,3] $\text{answer}(\text{flip})$

so flip is intelligent but cannot read!

▶ 119

Fahiem Bacchus, University of Toronto

CSC384h: Intro to Artificial Intelligence

► Reasoning Under Uncertainty

► D-Separation

More generally...

- ▶ Many conditional independencies hold in a given BN.
 - ▶ These independencies are useful in computation, explanation, etc.
- ▶ How do we determine if two variables X, Y are independent given a set of variables E ?

Simple graphical property: **D-separation**

- ▶ A set of variables E d-separates X and Y if it blocks every undirected path in the BN between X and Y . (We'll define blocks next.)
- ▶ X and Y are conditionally independent given evidence E if E d-separates X and Y
 - ▶ thus BN gives us an easy way to tell if two variables are independent (set $E = \emptyset$) or cond. independent given E .

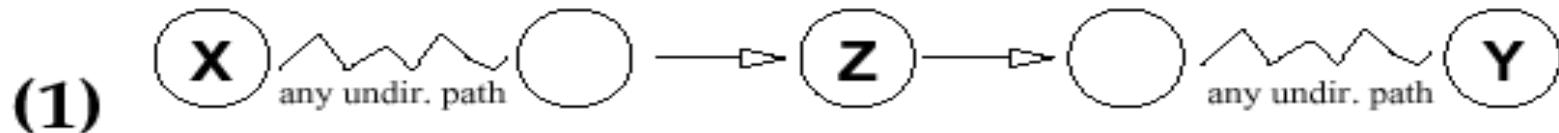
Blocking in D-Separation

- ▶ Let P be an **undirected path** from X to Y in a BN.
Let E (evidence) be a set of variables.

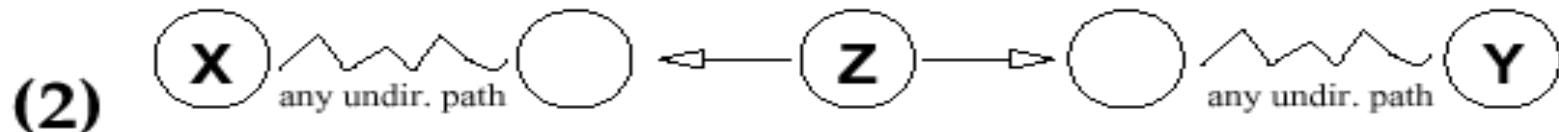
We say E blocks path P
iff **there is some** node Z on the path P such that:

- ▶ **Case 1:** $Z \in E$ and one arc on P enters (goes into) Z and one leaves (goes out of) Z ; or
- ▶ **Case 2:** $Z \in E$ and both arcs on P leave Z ; or
- ▶ **Case 3:** both arcs on P enter Z and *neither Z , nor any of its descendants, are in E .*

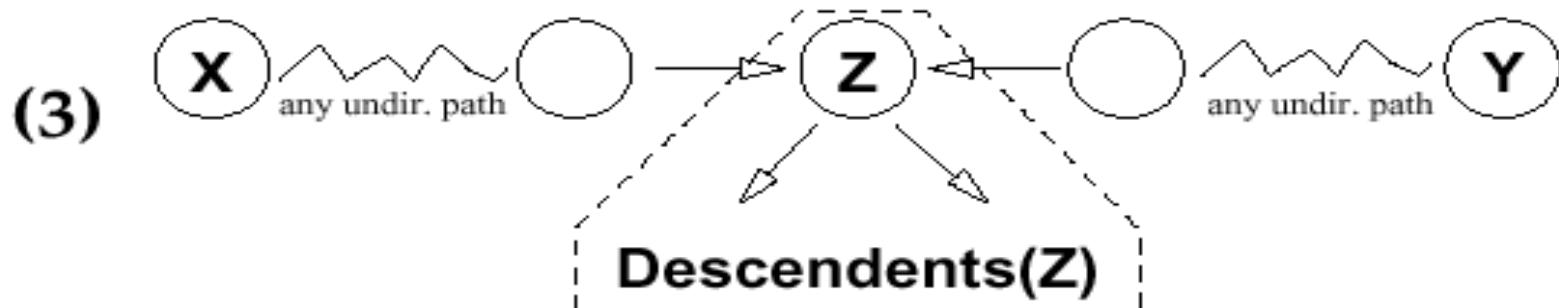
Blocking: Graphical View



If Z in evidence, the path between X and Y blocked



If Z in evidence, the path between X and Y blocked



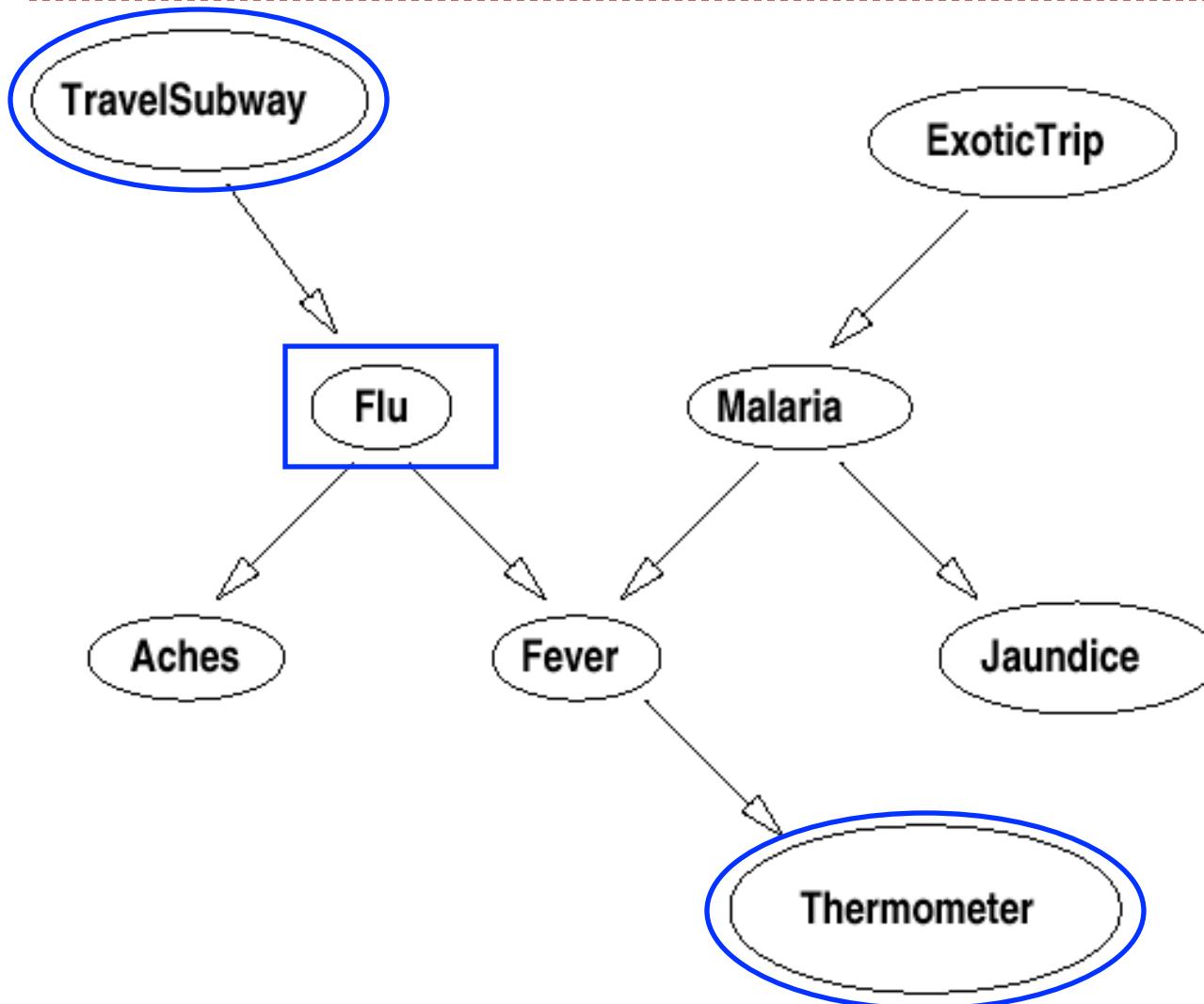
If Z is **not** in evidence and **no** descendent of Z is in evidence,
then the path between X and Y is blocked

Recall: D-Separation

D-separation:

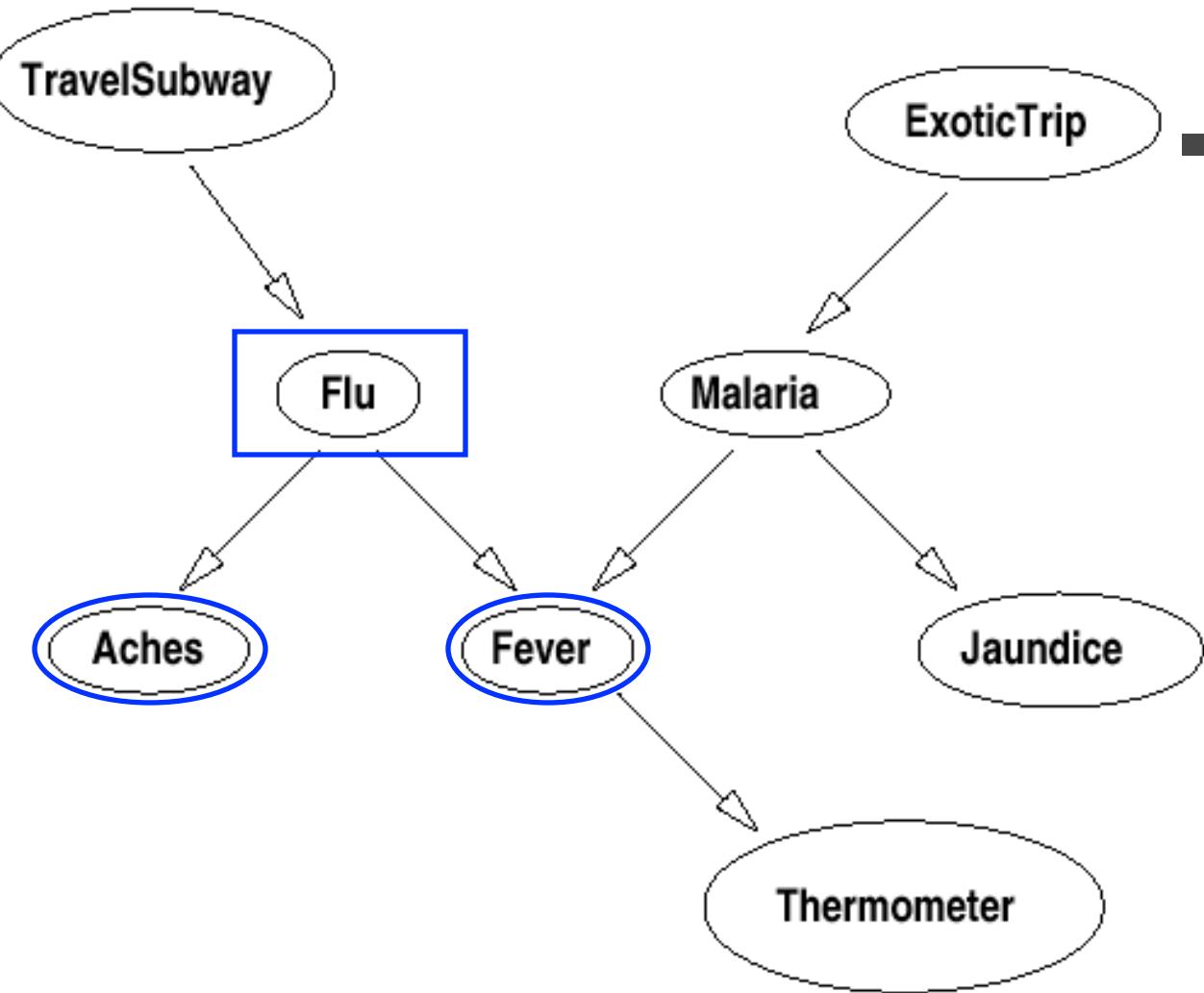
A set of variables E *d-separates* X and Y
if it *blocks every undirected path*
in the BN between X and Y .

D-Separation: Intuitions



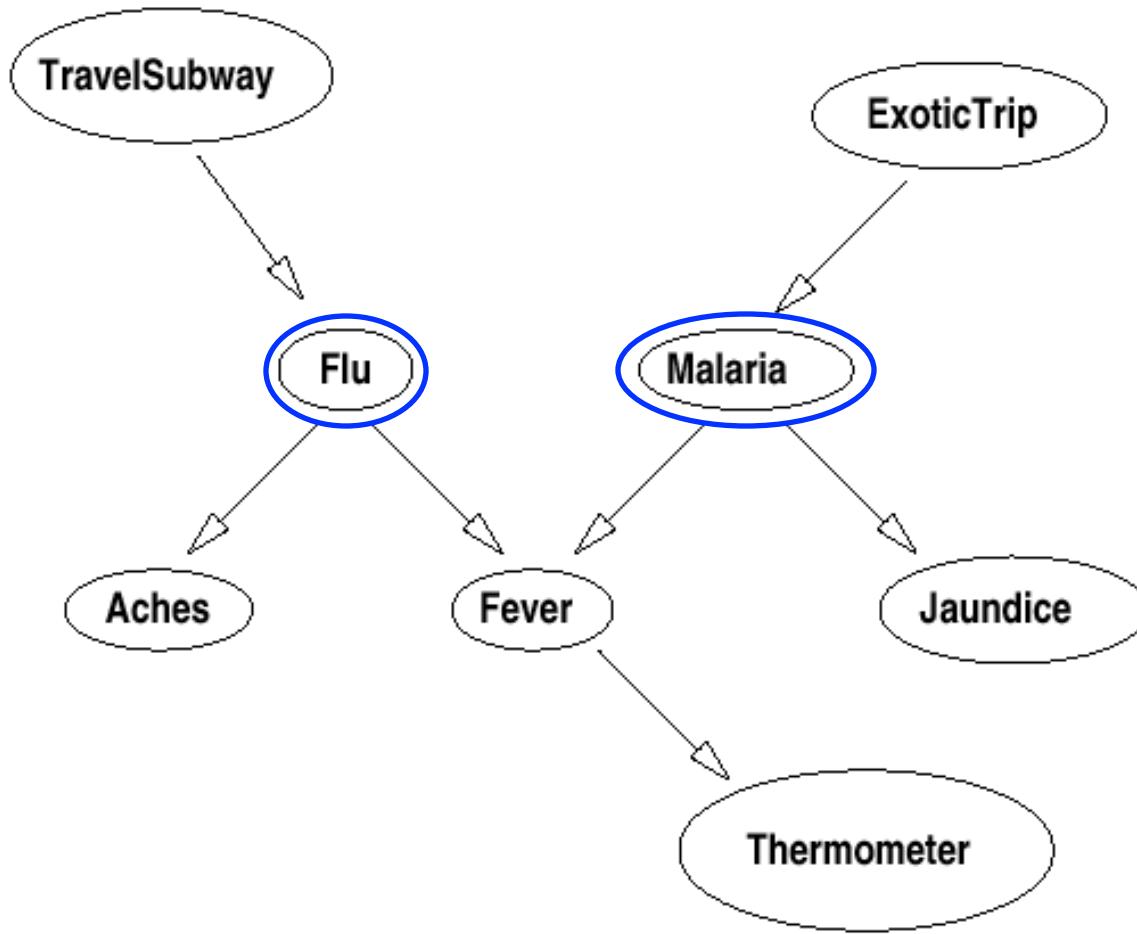
- Subway and Thermometer are dependent; but are independent given Flu (since Flu blocks the only path)

D-Separation: Intuitions



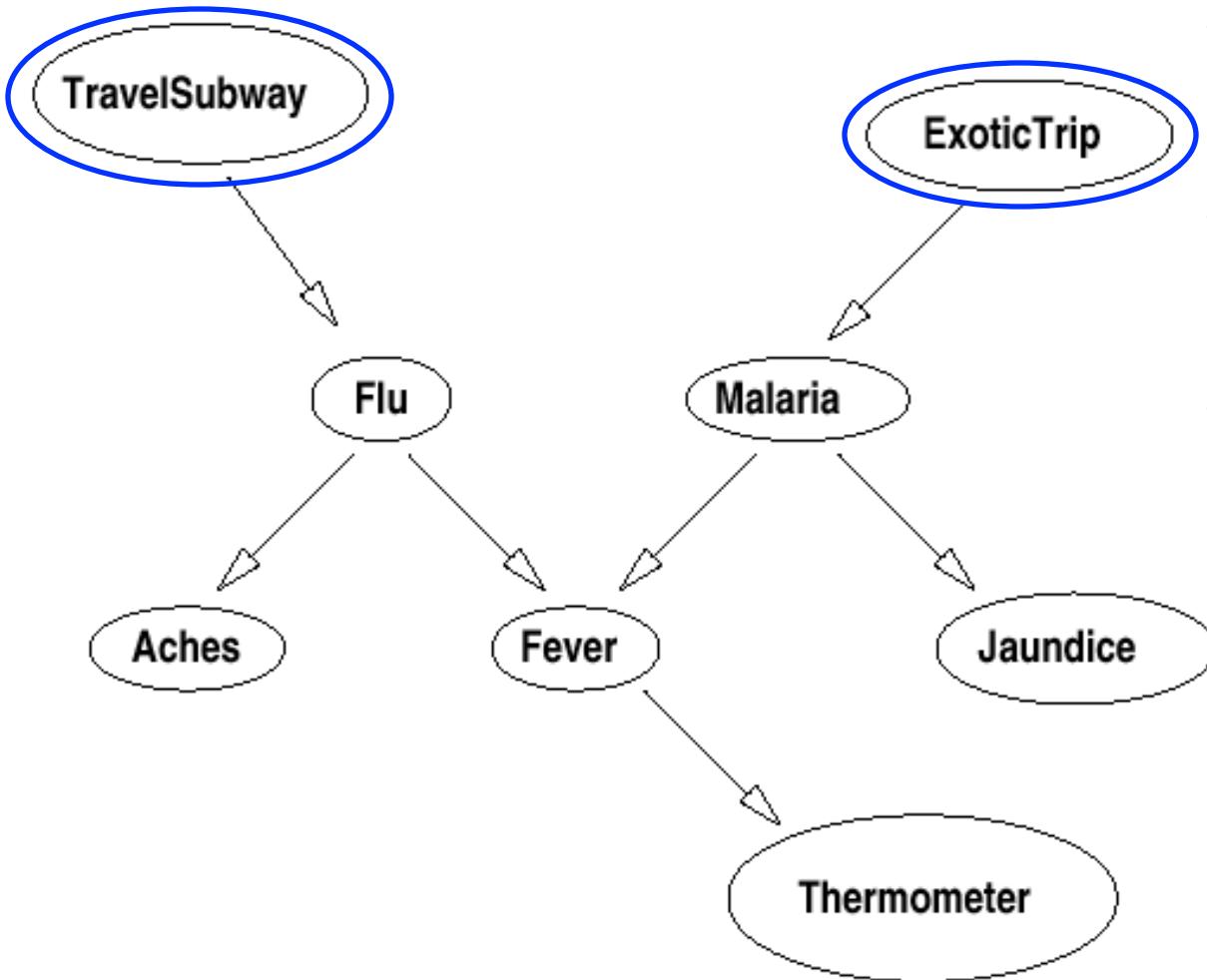
- Aches and Fever are dependent; but are independent given Flu (since Flu blocks the only path). Similarly for Aches and Therm (dependent, but indep. given Flu).

D-Separation: Intuitions



- Flu and Mal are independent (given no evidence): Fever blocks the path, since it is *not in evidence*, nor is its descendant Therm.
- Flu and Mal are dependent given Fever (or given Therm): nothing blocks path now.
What's the intuition?

D-Separation: Intuitions

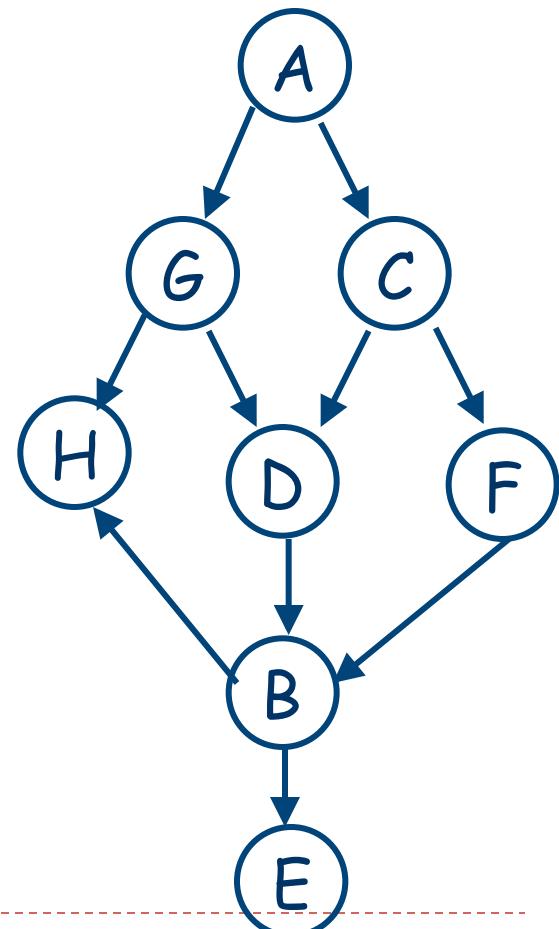


- Subway, ExoticTrip are independent;
- They are dependent given Therm;
- They are independent given Therm and Malaria. This for exactly the same reasons for Flu/Mal above.

D-Separation Example

- ▶ In the following network determine if A and E are independent given the evidence:

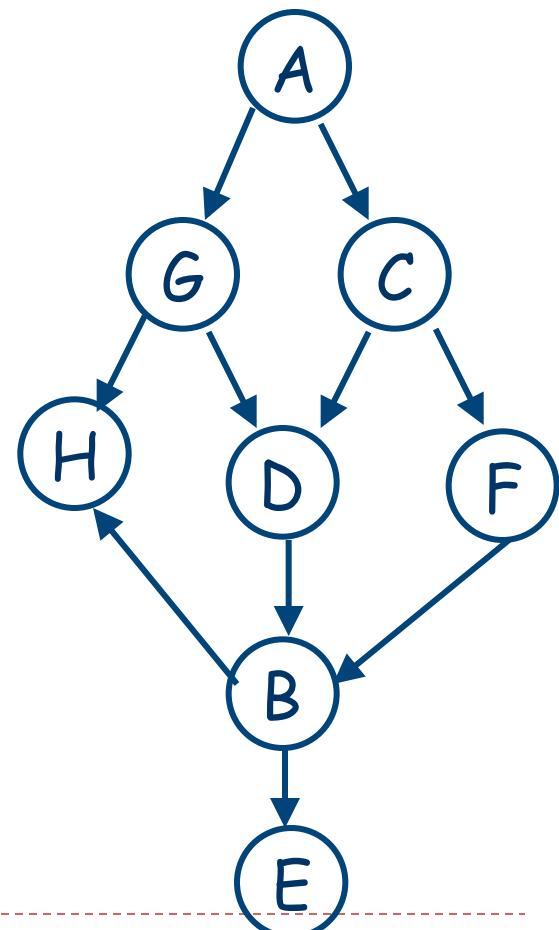
1. A and E given no evidence?
2. A and E given {C}?
3. A and E given {G,C}?
4. A and E given {G,C,H}?
5. A and E given {G,F}?
6. A and E given {F,D}?
7. A and E given {F,D,H}?
8. A and E given {B}?
9. A and E given {H,B}?
10. A and E given {G,C,D,H,D,F,B}?



D-Separation Example

- In the following network determine if A and E are independent given the evidence:

1. A and E given no evidence? **No**
2. A and E given {C}? **No**
3. A and E given {G,C}? **Yes**
4. A and E given {G,C,H}? **Yes**
5. A and E given {G,F}? **No**
6. A and E given {F,D}? **Yes**
7. A and E given {F,D,H}? **No**
8. A and E given {B}? **Yes**
9. A and E given {H,B}? **Yes**
10. A and E given {G,C,D,H,F,B}? **Yes**



CSC384h: Intro to Artificial Intelligence

► Reasoning Under Uncertainty

- This material is covered in chapters 13 and 14. Chapter 13 gives some basic background on probability from the point of view of AI. Chapter 14 talks about Bayesian Networks, exact reasoning in Bayes Nets as well as approximate reasoning, which will be main topics for us.

Uncertainty

- ▶ In search we viewed actions as being deterministic.
 - ▶ If you are in state S_1 and you execute action A you arrive at state S_2 .
- ▶ Furthermore, there was a fixed initial state S_0 . So with deterministic actions after executing any sequence of actions we know exactly what state we have arrived at.
 - ▶ Always know what state one is in.
- ▶ These assumptions are sensible in some domains
- ▶ But in many domains they are not true.

Uncertainty

- ▶ We might not know exactly what state we start off in
 - ▶ E.g., we can't see our opponents cards in a poker game
 - ▶ We don't know what a patient's ailment is.
- ▶ We might not know all of the effects of an action
 - ▶ The action might have a random component, like rolling dice.
 - ▶ We might not know all of the long term effects of a drug.
 - ▶ We might not know the status of a road when we choose the action of driving down it.

Uncertainty

- ▶ In such domains we still need to act, but we can't act solely on the basis of known true facts. We have to "gamble".
- ▶ E.g., we don't know for certain what the traffic will be like on a trip to the airport.

Uncertainty

- ▶ But how do we gamble **rationally**?
 - ▶ If we must arrive at the airport at 9pm on a week night we could “safely” leave for the airport $\frac{1}{2}$ hour before.
 - ▶ Some probability of the trip taking longer, but the probability is low.
 - ▶ If we must arrive at the airport at 4:30pm on Friday we most likely need 1 hour or more to get to the airport.
 - ▶ Relatively high probability of it taking 1.5 hours.

Uncertainty

- ▶ To act rationally under uncertainty we must be able to evaluate how likely certain things are.
- ▶ By weighing likelihoods of events (probabilities) we can develop mechanisms for acting rationally under uncertainty.

Probability over Finite Sets. (Review)

- ▶ Probability is a function defined over a set of **atomic events** U .
 - ▶ **The universe of events.**
- ▶ It assigns a value $\Pr(e)$ to each event $e \in U$, in the range $[0,1]$.
- ▶ It assigns a value to every set of events F by summing the probabilities of the members of that set.

$$\Pr(F) = \sum_{e \in F} \Pr(e)$$

- ▶ $\Pr(U) = 1$, i.e., sum over all events is 1.
- ▶ Therefore: $\Pr(\{\}) = 0$ and
 $\Pr(A \cup B) = \Pr(A) + \Pr(B) - \Pr(A \cap B)$

Probability in General (Review)

- Given a set \mathbf{U} (universe), a probability function is a function defined over the subsets of \mathbf{U} that maps each subset to the real numbers and that satisfies the Axioms of Probability

$$1. \Pr(\mathbf{U}) = 1$$

$$2. \Pr(A) \in [0,1]$$

$$3. \Pr(A \cup B) = \Pr(A) + \Pr(B) - \Pr(A \cap B)$$

Note if $A \cap B = \emptyset$ then $\Pr(A \cup B) = \Pr(A) + \Pr(B)$

Probability over Feature Vectors

- ▶ We will work with universes of events each of which is a vectors of feature values.
- ▶ Like CSPs, we have
 1. a set of variables V_1, V_2, \dots, V_n
 2. a finite domain of values for each variable,
 $\text{Dom}[V_1], \text{Dom}[V_2], \dots, \text{Dom}[V_n]$.
- ▶ The universe of events U is the set of all vectors of values for the variables
 $\langle d_1, d_2, \dots, d_n \rangle: d_i \in \text{Dom}[V_i]$

Probability over Feature Vectors

- ▶ This event space has size
$$\prod_i |\text{Dom}[V_i]|$$
i.e., the product of the domain sizes.
- ▶ E.g., if $|\text{Dom}[V_i]| = 2$ we have 2^n distinct atomic events. (Exponential!)

Probability over Feature Vectors

- ▶ Asserting that some subset of variables have particular values allows us to specify a useful collection of subsets of U .
- ▶ E.g.
 - ▶ $\{V_1 = a\}$ = set of all events where $V_1 = a$
 - ▶ $\{V_1 = a, V_3 = d\}$ = set of all events where $V_1 = a$ and $V_3 = d$.
 - ▶ ...
- ▶ E.g.
 - ▶ $\Pr(\{V_1 = a\}) = \sum_{x \in \text{Dom}[V_3]} \Pr(\{V_1 = a, V_3 = x\}).$

Probability over Feature Vectors

- ▶ If we had \Pr of every atomic event (full instantiation of the variables) we could compute the probability of **any** other set (including sets can cannot be specified some set of variable values).
- ▶ E.g.
 - ▶ $\{V_1 = a\} = \text{set of all events where } V_1 = a$
 - ▶ $\Pr(\{V_1 = a\}) =$

$$\sum_{x_2 \in \text{Dom}[v_2]} \sum_{x_3 \in \text{Dom}[v_3]} \cdots \sum_{x_n \in \text{Dom}[v_n]}$$

$$\Pr(V_1=a, V_2=x_2, V_3=x_3, \dots, V_n=x_n)$$

Probability over Feature Vectors

- ▶ Problem:
 - ▶ This is an exponential number of atomic probabilities to specify.
 - ▶ Requires summing up an exponential number of items.
- ▶ **For evaluating the probability of sets containing a particular subset of variable assignments we can do much better. Improvements come from the use of probabilistic independence, especially conditional independence.**

Conditional Probabilities. (Review)

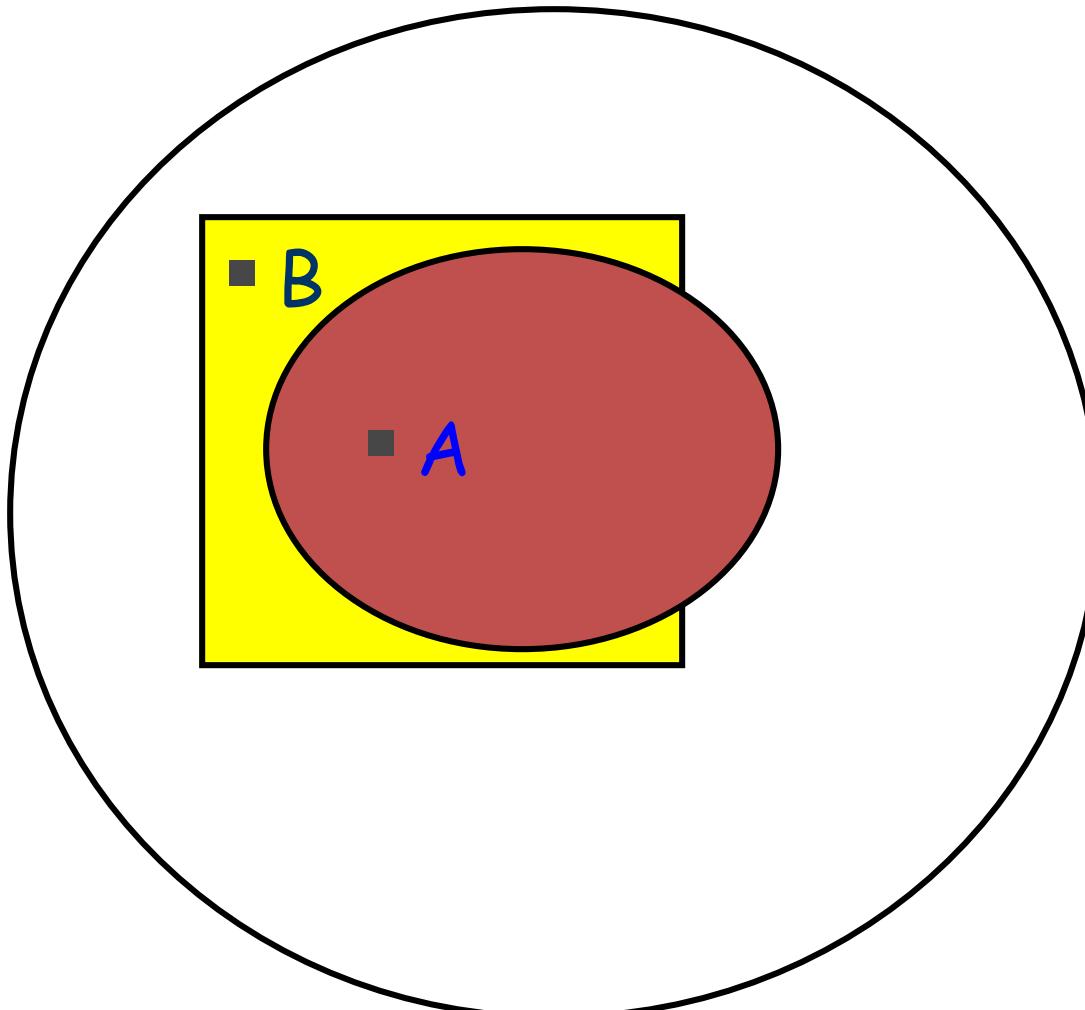
- ▶ With probabilities one can capture conditional information by using **conditional probabilities**.
- ▶ Conditional probabilities are essential for both representing and reasoning with probabilistic information.

Conditional Probabilities (Review)

- ▶ Say that A is a set of events such that $\Pr(A) > 0$.
- ▶ Then one can define a conditional probability wrt the event A:

$$\Pr(B | A) = \Pr(B \cap A) / \Pr(A)$$

Conditional Probabilities (Review)

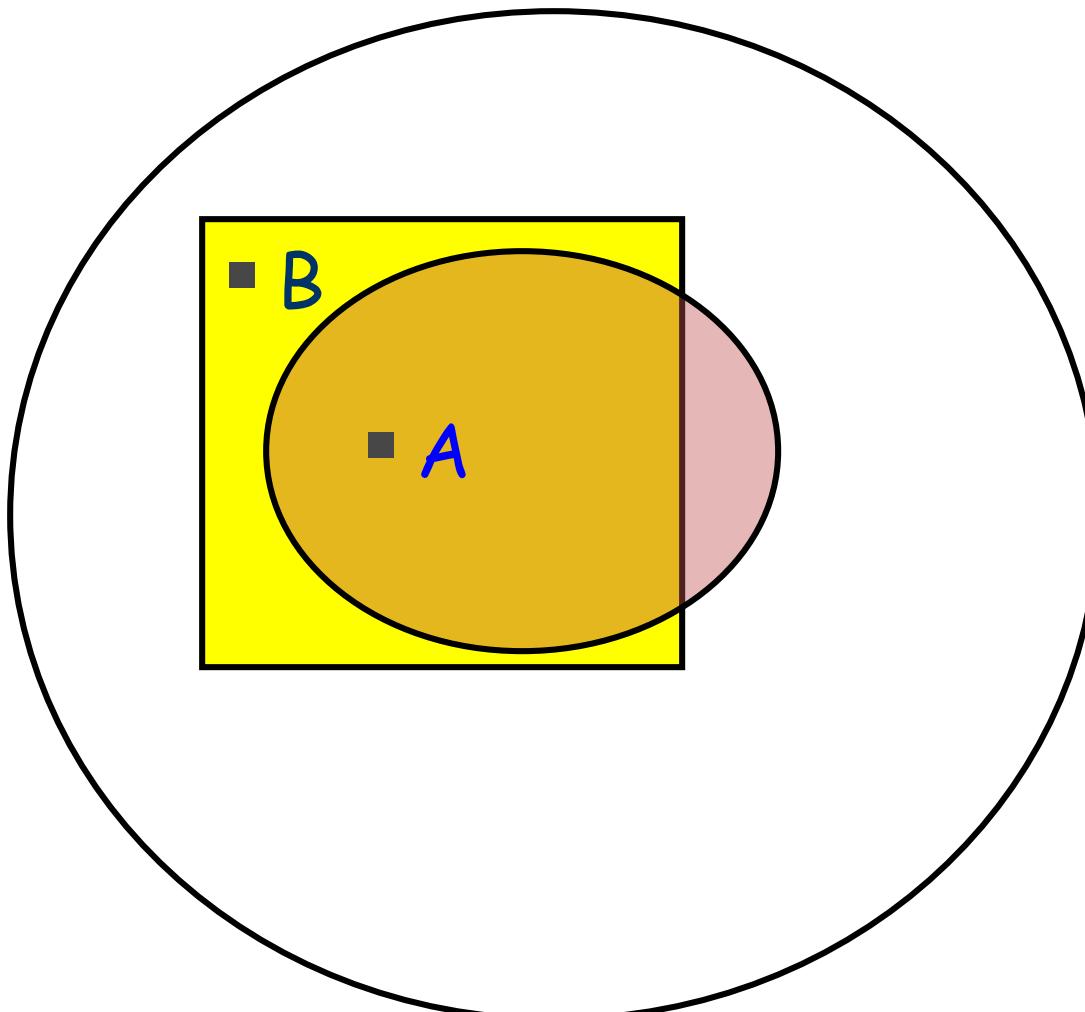


- B covers about 30% of the entire space, but covers over 80% of A.

Conditional Probabilities (Review)

- ▶ Conditioning on A, corresponds to restricting one's attention to the events in A.
- ▶ We now consider A to be the whole set of events (a new universe of events):
$$\Pr(A | A) = 1.$$
- ▶ Then we assign all other sets a probability by taking the probability mass that “lives” in A ($\Pr(B \wedge A)$), and normalizing it to the range [0,1] by dividing by $\Pr(A)$.

Conditional Probabilities (Review)



- B's probability in the new universe A is 0.8.

Conditional Probabilities (Review)

- ▶ A conditional probability is a probability function, but now over A instead of over the entire space.
 - ▶ $\Pr(A | A) = 1$
 - ▶ $\Pr(B | A) \in [0,1]$
 - ▶ $\Pr(C \cup B | A) = \Pr(C | A) + \Pr(B | A) - \Pr(C \cap B | A)$



Summing out rule

- ▶ Useful fact about probabilities
- ▶ Say that B_1, B_2, \dots, B_k form a **partition** of the universe **U**.
 1. $B_i \cap B_j = \emptyset$ $i \neq j$ (mutually exclusive)
 2. $B_1 \cup B_2 \cup B_3 \dots \cup B_k = U$ (exhaustive)
- ▶ In probabilities:
 1. $\Pr(B_i \cap B_j) = 0$
 2. $\Pr(B_1 \cup B_2 \cup B_3 \dots \cup B_k) = 1$



Summing out rule

- ▶ Given any other set of events A we have that

$$\Pr(A) = \Pr(A \cap B_1) + \Pr(A \cap B_2) + \dots + \Pr(A \cap B_k)$$

- ▶ In conditional probabilities:

$$\begin{aligned}\Pr(A) &= \Pr(A | B_1)\Pr(B_1) + \Pr(A | B_2)\Pr(B_2) + \dots \\ &\quad + \Pr(A | B_k)\Pr(B_k)\end{aligned}$$

$$\begin{aligned}\Pr(A | B_i)\Pr(B_i) &= \Pr(A \cap B_i)/\Pr(B_i) * \Pr(B_i) \\ &= \Pr(A \cap B_i)\end{aligned}$$

- ▶ Often we know $\Pr(A | B_i)$, so we can compute $\Pr(A)$ this way.



Properties and Sets

- ▶ Any set of events A can be interpreted as a property: the set of events with property A.
- ▶ Hence, we often write
 - ▶ $A \vee B$ to represent the set of events with either property A or B: the set $A \cup B$
 - ▶ $A \wedge B$ to represent the set of events both property A and B: the set $A \cap B$
 - ▶ $\neg A$ to represent the set of events that do not have property A: the set $U - A$ (i.e., the complement of A wrt the universe of events U)

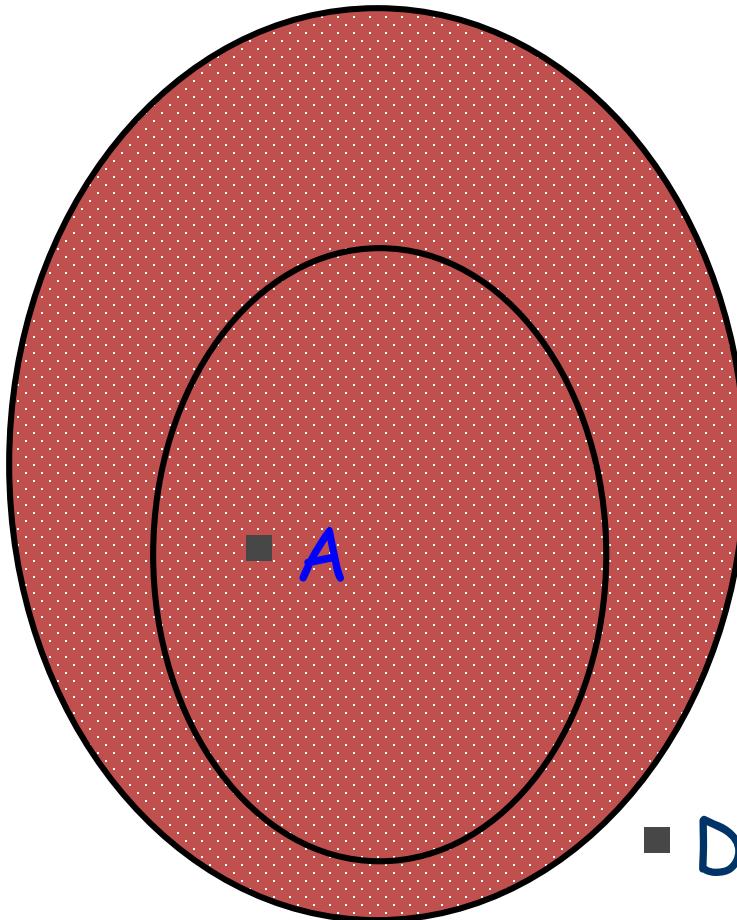


Independence (Review)

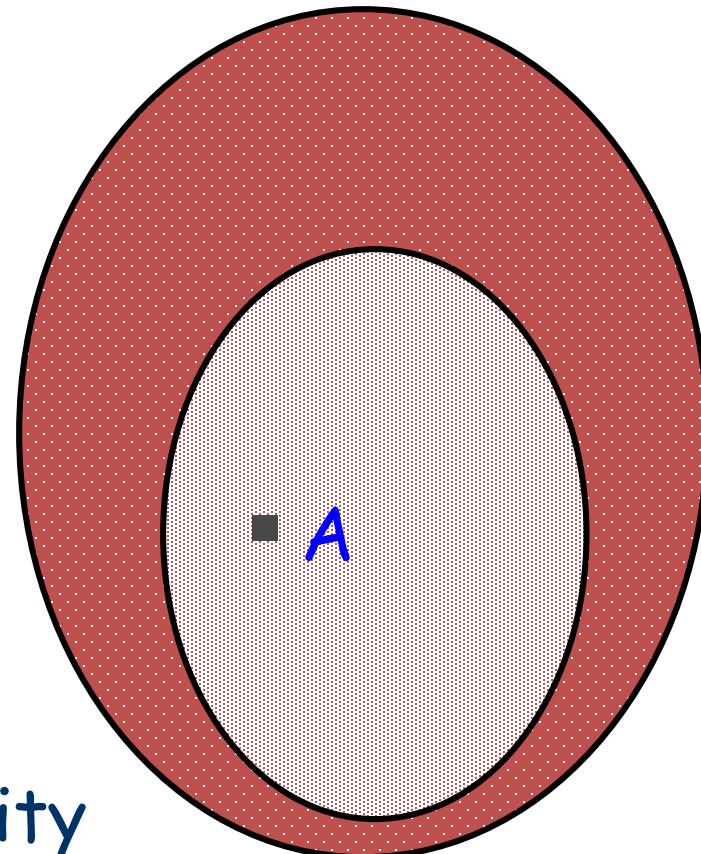
- ▶ It could be that the density of B on A is identical to its density on the entire set.
 - ▶ Density: pick an element at random from the entire set.
How likely is it that the picked element is in the set B?
- ▶ Alternately the density of B on A could be much different than its density on the whole space.
- ▶ In the first case we say that B is **independent** of A. While in the second case B is dependent on A.

Independence (Review)

- Independent



- Dependent



- Density of B

Independence Definition (**Review**)

A and B are independent properties

$$\Pr(B | A) = \Pr(B)$$

A and B are dependent.

$$\Pr(B | A) \neq \Pr(B)$$

Implications for Knowledge

- ▶ Say that we have picked an element from the entire set. Then we find out that this element has property A (i.e., is a member of the set A).
 - ▶ Does this tell us anything more about how likely it is that the element also has property B?
 - ▶ If B is independent of A then we have learned nothing new about the likelihood of the element being a member of B.

Independence

- ▶ E.g., we have a feature vector, we don't know which one. We then find out that it contains the feature $V_1=a$.
 - ▶ I.e., we know that the vector is a member of the set $\{V_1 = a\}$.
 - ▶ Does this tell us anything about whether or not $V_2=a$, $V_3=c$, ..., etc?
 - ▶ This depends on whether or not these features are independent/dependent of $V_1=a$.

Conditional Independence

- ▶ Say we have already learned that the randomly picked element has property A.
- ▶ We want to know whether or not the element has property B:
 $\Pr(B | A)$ expresses the probability of this being true.
- ▶ Now we learn that the element also has property C. Does this give us more information about B-ness?

$\Pr(B | A \wedge C)$ expresses the probability of this being true under the additional information.

Conditional Independence

- ▶ If

$$\Pr(B | A \wedge C) = \Pr(B | A)$$

then we have not gained any additional information from knowing that the element is also a member of the set C.

- ▶ In this case we say that B is conditionally independent of C given A.
- ▶ That is, once we know A, additionally knowing C is irrelevant (wrt to whether or not B is true).
 - ▶ Conditional independence is independence in the conditional probability space $\Pr(\bullet | A)$.
 - ▶ Note we could have $\Pr(B | C) \neq \Pr(B)$. But once we learn A, C becomes irrelevant.

Computational Impact of Independence

- ▶ We will see in more detail how independence allows us to speed up computation. But the fundamental insight is that

If A and B are independent properties then

$$\Pr(A \wedge B) = \Pr(B) * \Pr(A)$$

Proof:

$$\Pr(B | A) = \Pr(B)$$

$$\Pr(A \wedge B) / \Pr(A) = \Pr(B)$$

$$\Pr(A \wedge B) = \Pr(B) * \Pr(A)$$

independence
definition

Computational Impact of Independence

- ▶ This property allows us to “break” up the computation of a conjunction “ $\text{Pr}(A \wedge B)$ ” into two separate computations “ $\text{Pr}(A)$ ” and “ $\text{Pr}(B)$ ”.
- ▶ Dependent on how we express our probabilistic knowledge this yield great computational savings.

Computational Impact of Independence

- ▶ Similarly for conditional independence.

$$\Pr(B | C \wedge A) = \Pr(B | A) \rightarrow$$

$$\Pr(B \wedge C | A) = \Pr(B | A) * \Pr(C | A)$$

Proof:

$$\Pr(B | C \wedge A) = \Pr(B | A)$$

independence

$$\Pr(B \wedge C \wedge A) / \Pr(C \wedge A) = \Pr(B \wedge A) / \Pr(A) \quad \text{defn.}$$

$$\Pr(B \wedge C \wedge A) / \Pr(A) = \Pr(C \wedge A) / \Pr(A) * \Pr(B \wedge A) / \Pr(A)$$

$$\Pr(B \wedge C | A) = \Pr(B | A) * \Pr(C | A) \quad \text{defn.}$$

Computational Impact of Independence

- ▶ Conditional independence allows us to break up our computation onto distinct parts

$$\Pr(B \wedge C | A) = \Pr(B | A) * \Pr(C | A)$$

- ▶ And it also allows us to ignore certain pieces of information

$$\Pr(B | A \wedge C) = \Pr(B | A)$$

Bayes Rule (Review)

- ▶ Bayes rule is a simple mathematical fact. But it has great implications wrt how probabilities can be reasoned with.
- ▶ $\Pr(Y | X) = \Pr(X | Y)\Pr(Y)/\Pr(X)$

$$\begin{aligned}\Pr(Y | X) &= \Pr(Y \wedge X)/\Pr(X) \\ &= \Pr(Y \wedge X)/\Pr(X) * P(Y)/P(Y) \\ &= \Pr(Y \wedge X)/\Pr(Y) * \Pr(Y)/\Pr(X) \\ &= \Pr(X | Y)\Pr(Y)/\Pr(X)\end{aligned}$$

Bayes Rule

- ▶ Bayes rule allows us to use a supplied conditional probability in both directions.
- ▶ E.g., from treating patients with heart disease we might be able to estimate the value of
$$\Pr(\text{high_Cholesterol} \mid \text{heart_disease})$$
- ▶ With Bayes rule we can turn this around into a predictor for heart disease
$$\Pr(\text{heart_disease} \mid \text{high_Cholesterol})$$
- ▶ Now with a simple blood test we can determine "high_Cholesterol" use this to help estimate the likelihood of heart disease.

Bayes Rule

- ▶ For this to work we have to deal with the other factors as well

$$\begin{aligned} \Pr(\text{heart_disease} \mid \text{high_Cholesterol}) \\ = \Pr(\text{high_Cholesterol} \mid \text{heart_disease}) \\ * \Pr(\text{heart_disease}) / \Pr(\text{high_Cholesterol}) \end{aligned}$$

- ▶ We will return to this later.

Bayes Rule Example

- ▶ Disease $\in \{\text{malaria, cold, flu}\}$; Symptom = fever
 - ▶ Must compute $\Pr(D \mid \text{fever})$ to prescribe treatment
- ▶ Why not assess this quantity directly?
 - ▶ $\Pr(\text{mal} \mid \text{fever})$ is not natural to assess;
 $\Pr(\text{mal} \mid \text{fever})$ does not reflects the underlying
“causal” mechanism fever → malaria
 - ▶ $\Pr(\text{mal} \mid \text{fever})$ is not “stable”: a malaria epidemic
changes this quantity (for example)
- ▶ So we use Bayes rule:
 - ▶ $\Pr(\text{mal} \mid \text{fever}) = \Pr(\text{fever} \mid \text{mal}) \Pr(\text{mal}) / \Pr(\text{fever})$

Bayes Rule

- ▶ $\Pr(\text{mal} \mid \text{fever}) = \Pr(\text{fever} \mid \text{mal})\Pr(\text{mal})/\Pr(\text{fever})$
- ▶ $\Pr(\text{mal})$?
 - ▶ This is the prior probability of Malaria, i.e., before you exhibited a fever, and with it we can account for other factors, e.g., a malaria epidemic, or recent travel to a malaria risk zone.
 - ▶ E.g., The center for disease control keeps track of the rates of various diseases.
- ▶ $\Pr(\text{fever} \mid \text{mal})$?
 - ▶ This is the probability a patient with malaria exhibits a fever.
 - ▶ Again this kind of information is available from people who study malaria and its effects.

Bayes Rule

- ▶ $\Pr(\text{fever})?$
 - ▶ This is typically not known, but it can be computed!
 - ▶ We eventually have to divide by this probability to get the final answer:
 $\Pr(\text{mal} \mid \text{fever}) = \Pr(\text{fever} \mid \text{mal})\Pr(\text{mal})/\Pr(\text{fever})$
- ▶ First, we find a set of mutually exclusive and exhaustive causes for fever:
 - ▶ Say that in our example, mal, cold and flu are only possible causes of fever and they are mutually exclusive.
 - ▶ $\Pr(\text{fev} \mid \neg \text{mal} \wedge \neg \text{cold} \wedge \neg \text{flu}) = 0 \rightarrow \text{Fever can't happen with one of these causes.}$
 - ▶ $\Pr(\text{mal} \wedge \text{cold}) = \Pr(\text{mal} \wedge \text{flu}) = \Pr(\text{cold} \wedge \text{flu}) = 0 \rightarrow \text{these causes can't happen together.}$ (Note that our example is not very realistic!)
- ▶ Second, we compute

$$\begin{aligned}\Pr(\text{fever} \mid \text{mal})\Pr(\text{mal}), \\ \Pr(\text{fever} \mid \text{cold})\Pr(\text{cold}). \\ \Pr(\text{fever} \mid \text{flu})\Pr(\text{flu}).\end{aligned}$$

We know $\Pr(\text{fever} \mid \text{cold})$ and $\Pr(\text{fever} \mid \text{flu})$, along with $\Pr(\text{cold})$ and $\Pr(\text{flu})$ from the same sources as $\Pr(\text{fever} \mid \text{mal})$ and $\Pr(\text{mal})$

Bayes Rule

- ▶ Since flu, cold and malaria are exclusive, $\{\text{flu}, \text{cold}, \text{malaria}, \neg\text{mal} \wedge \neg\text{cold} \wedge \neg\text{flu}\}$ forms a partition of the universe. So

$$\begin{aligned}\Pr(\text{fever}) = & \Pr(\text{fever} | \text{mal}) * \Pr(\text{mal}) + \Pr(\text{fever} | \text{cold}) * \Pr(\text{cold}) \\ & + \Pr(\text{fever} | \text{flu}) * \Pr(\text{flu}) \\ & + \Pr(\text{fever} | \neg\text{mal} \wedge \neg\text{cold} \wedge \neg\text{flu}) * \Pr(\neg\text{mal} \wedge \neg\text{cold} \wedge \neg\text{flu})\end{aligned}$$

- ▶ The last term is zero as fever is not possible unless one of malaria, cold, or flu is true.
- ▶ So to compute the trio of numbers, $\Pr(\text{mal} | \text{fever})$, $\Pr(\text{cold} | \text{fever})$, $\Pr(\text{flu} | \text{fever})$, we compute the trio of numbers $\Pr(\text{fever} | \text{mal}) * \Pr(\text{mal})$, $\Pr(\text{fever} | \text{cold}) * \Pr(\text{cold})$, $\Pr(\text{fever} | \text{flu}) * \Pr(\text{flu})$
- ▶ And then we divide these three numbers by $\Pr(\text{fever})$.
 - ▶ That is we divide these three numbers by their sum:
This is called **normalizing** the numbers.
- ▶ Thus we never need actually compute $\Pr(\text{fever})$ (unless we want to).

Normalizing

- ▶ If we have a vector of k numbers, e.g., $\langle 3, 4, 2.5, 1, 10, 21.5 \rangle$ we can **normalize** these numbers by dividing each number by the sum of the numbers:
 - ▶ $3 + 4 + 2.5 + 1 + 10 + 21.5 = 42$
 - ▶ Normalized vector
 $= \langle 3/42, 4/42, 2.5/42, 1/42, 10/42, 21.5/42 \rangle$
 $= \langle 0.071, 0.095, 0.060, 0.024, 0.238, 0.512 \rangle$
- ▶ After normalizing the vector of numbers sums to 1
 - ▶ Exactly what is needed for these numbers to specify a probability distribution.

Chain Rule (Review)

- ▶ $\Pr(A_1 \wedge A_2 \wedge \dots \wedge A_n) =$
 $\Pr(A_1 | A_2 \wedge \dots \wedge A_n) * \Pr(A_2 | A_3 \wedge \dots \wedge A_n)$
 $* \dots * \Pr(A_{n-1} | A_n) * \Pr(A_n)$

Proof:

$$\begin{aligned}& \Pr(A_1 | A_2 \wedge \dots \wedge A_n) * \Pr(A_2 | A_3 \wedge \dots \wedge A_n) \\& * \dots * \Pr(A_{n-1} | A_n) \\&= \Pr(A_1 \wedge A_2 \wedge \dots \wedge A_n) / \Pr(A_2 \wedge \dots \wedge A_n) * \\& \quad \Pr(A_2 \wedge \dots \wedge A_n) / \Pr(A_3 \wedge \dots \wedge A_n) * \dots * \\& \quad \Pr(A_{n-1} \wedge A_n) / \Pr(A_n) * \Pr(A_n)\end{aligned}$$

Variable Independence

- ▶ Recall that we will be mainly dealing with probabilities over feature vectors.
- ▶ We have a set of variables, each with a domain of values.
- ▶ It could be that $\{V_1=a\}$ and $\{V_2=b\}$ are independent:

$$Pr(V_1=a \wedge V_2=b) = Pr(V_1=a) * Pr(V_2=b)$$

- ▶ It could also be that $\{V_1=b\}$ and $\{V_2=b\}$ are not independent:

$$Pr(V_1=b \wedge V_2=b) \neq Pr(V_1=b) * Pr(V_2=b)$$

Variable Independence

- ▶ However we will generally want to deal with the situation where we have **variable independence**.
 - ▶ Two **variables** X and Y are **conditionally independent given variable Z** iff
$$\forall x,y,z. x \in \text{Dom}(X) \wedge y \in \text{Dom}(Y) \wedge z \in \text{Dom}(Z)$$
$$\rightarrow X=x \text{ is conditionally independent of } Y=y \text{ given } Z=z$$
$$\equiv \Pr(X=x \wedge Y=y | Z=z)$$
$$= \Pr(X=x | Z=z) * \Pr(Y=y | Z=z)$$
- ▶ Also applies to sets of more than two variables
- ▶ Also to unconditional case (X,Y independent)

Variable Independence

- ▶ If you know the value of Z (*whatever* it is), learning Y's value (*whatever* it is) does not influence your beliefs about any of X's values.
- ▶ these definitions differ from earlier ones, which talk about particular sets of events being independent. Variable independence is a concise way of stating a number of individual independencies.

What does independence buys us?

- ▶ Suppose (say, boolean) variables X_1, X_2, \dots, X_n are mutually independent (i.e., every subset is variable independent of every other subset)
 - ▶ we can specify full joint distribution (probability function over all vectors of values) using only n parameters (linear) instead of $2^n - 1$ (exponential)
- ▶ How? Simply specify $\Pr(X_1), \dots \Pr(X_n)$ (i.e., $\Pr(X_i = \text{true})$ for all i)
 - ▶ from this I can recover probability of any primitive event easily (or any conjunctive query).
e.g. $\Pr(X_1 \rightarrow X_2 X_3 X_4) = \Pr(X_1) (1 - \Pr(X_2)) \Pr(X_3) \Pr(X_4)$
 - ▶ we can condition on observed value X_k (or $\neg X_k$) trivially
 $\Pr(X_1 \rightarrow X_2 | X_3) = \Pr(X_1) (1 - \Pr(X_2))$

The Value of Independence

- ▶ Complete independence reduces both *representation of joint* and *inference* from $O(2^n)$ to $O(n)!$
- ▶ Unfortunately, such complete mutual independence is very rare. Most realistic domains do not exhibit this property.
- ▶ Fortunately, most domains do exhibit a fair amount of conditional independence. And we can exploit conditional independence for representation and inference as well.
- ▶ **Bayesian networks** do just this

An Aside on Notation

- ▶ $\Pr(X)$ for variable X (or set of variables) refers to the *(marginal) distribution* over X .
 - ▶ It specifies $\Pr(X=d)$ for all $d \in \text{Dom}[X]$
- ▶ Note

$$\sum_{d \in \text{Dom}[X]} \Pr(X=d) = 1$$

(every vector of values must be in one of the sets $\{X=d\} d \in \text{Dom}[X]$)

- ▶ Also

$$\Pr(X=d_1 \wedge X=d_2) = 0 \text{ for all } d_1, d_2 \in \text{Dom}[X] \text{ } d_1 \neq d_2$$

(no vector of values contains two different values for X).

An Aside on Notation

- ▶ $\Pr(X | Y)$ refers to family of conditional distributions over X , one for each $y \in \text{Dom}(Y)$.
 - ▶ For each $d \in \text{Dom}[Y]$, $\Pr(X | Y)$ specifies a distribution over the values of X :
 $\Pr(X=d_1 | Y=d), \Pr(X=d_2 | Y=d), \dots, \Pr(X=d_n | Y=d)$
for $\text{Dom}[X] = \{d_1, d_2, \dots, d_n\}$.
- ▶ Distinguish between $\Pr(X)$ —which is a distribution—and $\Pr(x_i)$ ($x_i \in \text{Dom}[X]$)— which is a number. Think of $\Pr(X)$ as a function that accepts any $x_i \in \text{Dom}(X)$ as an argument and returns $\Pr(x_i)$.
- ▶ Similarly, think of $\Pr(X | Y)$ as a function that accepts any $x_i \in \text{Dom}[X]$ and $y_k \in \text{Dom}[Y]$ and returns $\Pr(x_i | y_k)$. Note that $\Pr(X | Y)$ is not a single distribution; rather it denotes the family of distributions (over X) induced by the different $y_k \in \text{Dom}(Y)$

Exploiting Conditional Independence

- ▶ Let's see what conditional independence buys us
- ▶ Consider a story:
 - ▶ If Craig woke up too early E, Craig probably needs coffee C; if C, Craig needs coffee, he's likely angry A. If A, there is an increased chance of an aneurysm (burst blood vessel) B. If B, Craig is quite likely to be hospitalized H.



E - Craig woke too early A - Craig is angry H - Craig hospitalized
C - Craig needs coffee B - Craig burst a blood vessel

Cond'l Independence in our Story



- ▶ If you learned any of E, C, A, or B, your assessment of $\text{Pr}(H)$ would change.
 - ▶ E.g., if any of these are seen to be true, you would increase $\text{Pr}(h)$ and decrease $\text{Pr}(\sim h)$.
 - ▶ So H is *not independent* of E, or C, or A, or B.
- ▶ But if you knew value of B (true or false), learning value of E, C, or A, would not influence $\text{Pr}(H)$. Influence these factors have on H is mediated by their influence on B.
 - ▶ Craig doesn't get sent to the hospital because he's angry, he gets sent because he's had an aneurysm.
 - ▶ So H is *independent* of E, and C, and A, given B

Cond'l Independence in our Story



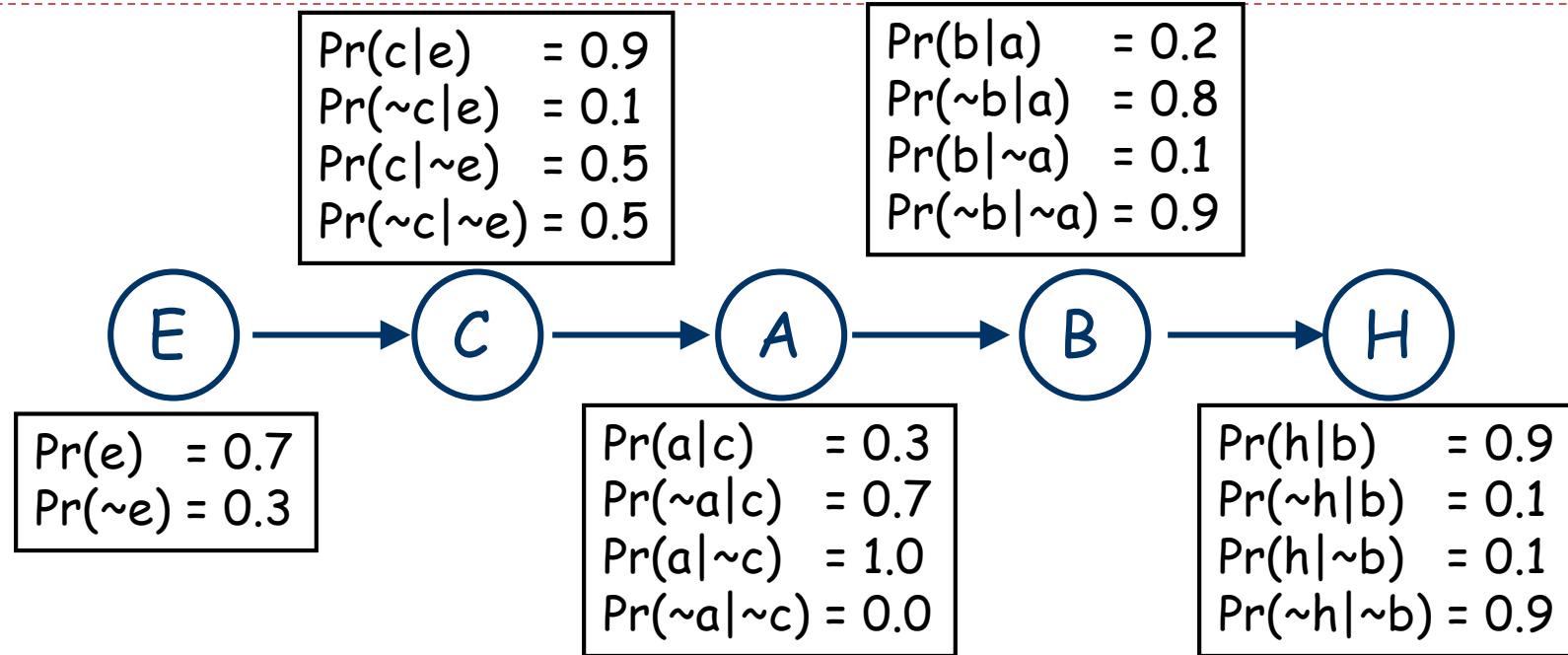
- ▶ Similarly:
 - ▶ B is *independent* of E, and C, given A
 - ▶ A is *independent* of E, given C
- ▶ This means that:
 - ▶ $\Pr(H | B, \{A,C,E\}) = \Pr(H | B)$
 - ▶ i.e., for any subset of $\{A,C,E\}$, this relation holds
 - ▶ $\Pr(B | A, \{C,E\}) = \Pr(B | A)$
 - ▶ $\Pr(A | C, \{E\}) = \Pr(A | C)$
 - ▶ $\Pr(C | E)$ and $\Pr(E)$ don't "simplify"

Cond'l Independence in our Story



- ▶ By the chain rule (for any instantiation of $H \dots E$):
 - ▶ $\Pr(H, B, A, C, E) =$
 $\Pr(H | B, A, C, E) \Pr(B | A, C, E) \Pr(A | C, E) \Pr(C | E) \Pr(E)$
- ▶ By our independence assumptions:
 - ▶ $\Pr(H, B, A, C, E) =$
 $\Pr(H | B) \Pr(B | A) \Pr(A | C) \Pr(C | E) \Pr(E)$
- ▶ We can specify the full joint by specifying five *local conditional distributions*: $\Pr(H | B)$; $\Pr(B | A)$; $\Pr(A | C)$; $\Pr(C | E)$; and $\Pr(E)$

Example Quantification



- ▶ Specifying the joint requires only 9 parameters (if we note that half of these are “1 minus” the others), instead of 31 for explicit representation
 - ▶ linear in number of vars instead of exponential!
 - ▶ linear generally if dependence has a chain structure

Inference is Easy



- Want to know $P(a)$? Use summing out rule:
 - Note the set of events $C=c_i$ for $c_i \in \text{Dom}(C)$ is a **partition**.

$$\begin{aligned} P(a) &= \sum_{c_i \in \text{Dom}(C)} \Pr(a | c_i) \Pr(c_i) \\ &= \sum_{c_i \in \text{Dom}(C)} \Pr(a | c_i) \sum_{e_i \in \text{Dom}(E)} \Pr(c_i | e_i) \Pr(e_i) \end{aligned}$$

These are all terms specified in our local distributions!

Inference is Easy



▶ Computing $P(a)$ in more concrete terms:

- ▶
$$\begin{aligned} P(c) &= P(c | e)P(e) + P(c | \sim e)P(\sim e) \\ &= 0.8 * 0.7 + 0.5 * 0.3 = 0.78 \end{aligned}$$
- ▶
$$P(\sim c) = P(\sim c | e)P(e) + P(\sim c | \sim e)P(\sim e) = 0.22$$
 - ▶ $P(\sim c) = 1 - P(c)$, as well
- ▶
$$\begin{aligned} P(a) &= P(a | c)P(c) + P(a | \sim c)P(\sim c) \\ &= 0.7 * 0.78 + 0.0 * 0.22 = 0.546 \end{aligned}$$
- ▶ $P(\sim a) = 1 - P(a) = 0.454$

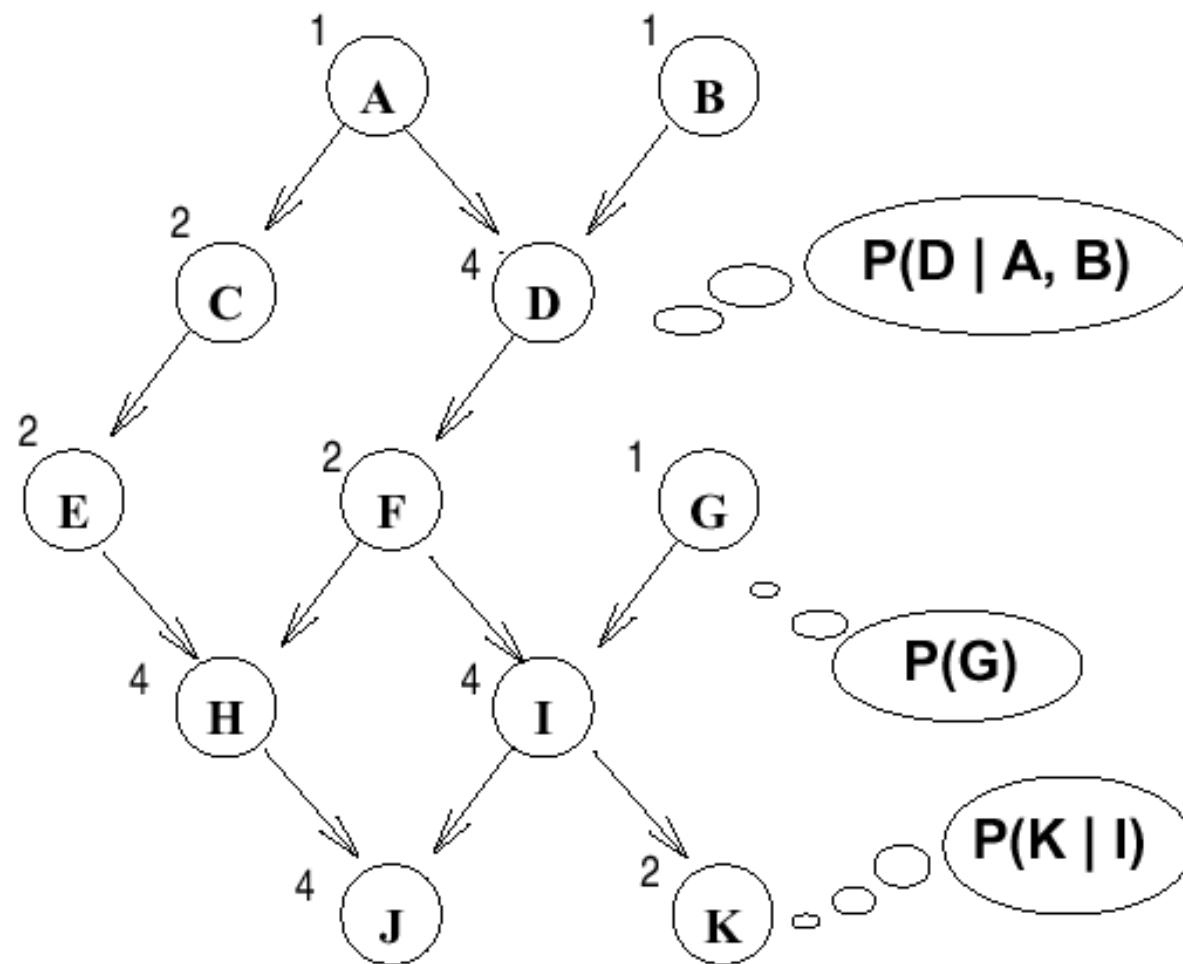
Bayesian Networks

- ▶ The structure above is a *Bayesian network*. A BN is a *graphical representation* of the direct dependencies over a set of variables, together with a set of *conditional probability tables* quantifying the strength of those influences.
- ▶ Bayes nets generalize the above ideas in very interesting ways, leading to effective means of representation and inference under uncertainty.

Bayesian Networks

- ▶ A BN over variables $\{X_1, X_2, \dots, X_n\}$ consists of:
 - ▶ a DAG (directed acyclic graph) whose nodes are the variables
 - ▶ a set of **CPTs** (conditional probability tables) $\Pr(X_i \mid \text{Par}(X_i))$ for each X_i
- ▶ Key notions (see text for defn's, all are intuitive):
 - ▶ parents of a node: $\text{Par}(X_i)$
 - ▶ children of node
 - ▶ descendants of a node
 - ▶ ancestors of a node
 - ▶ family: set of nodes consisting of X_i and its parents
 - ▶ CPTs are defined over families in the BN

Example (Binary valued Variables)



- ▶ A couple of the CPTs are “shown”

Semantics of Bayes Nets.

- ▶ A Bayes net specifies that the joint distribution over the variable in the net can be written as the following product decomposition.
- ▶ $\Pr(X_1, X_2, \dots, X_n)$
 $= \Pr(X_n \mid \text{Par}(X_n)) * \Pr(X_{n-1} \mid \text{Par}(X_{n-1}))$
 $* \dots * \Pr(X_1 \mid \text{Par}(X_1))$
- ▶ This equation hold for any set of values d_1, d_2, \dots, d_n for the variables X_1, X_2, \dots, X_n .

Semantics of Bayes Nets.

- ▶ E.g., say we have X_1, X_2, X_3 each with domain $\text{Dom}[X_i] = \{a, b, c\}$ and we have

$$\begin{aligned} & \Pr(X_1, X_2, X_3) \\ &= P(X_3 | X_2) P(X_2) P(X_1) \end{aligned}$$

Then

$$\begin{aligned} & \Pr(X_1=a, X_2=a, X_3=a) \\ &= P(X_3=a | X_2=a) P(X_2=a) P(X_1=a) \\ & \Pr(X_1=a, X_2=a, X_3=b) \\ &= P(X_3=b | X_2=a) P(X_2=a) P(X_1=a) \\ & \Pr(X_1=a, X_2=a, X_3=c) \\ &= P(X_3=c | X_2=a) P(X_2=a) P(X_1=a) \\ & \Pr(X_1=a, X_2=b, X_3=a) \\ &= P(X_3=a | X_2=b) P(X_2=b) P(X_1=a) \end{aligned}$$

...

Example (Binary valued Variables)

$$\Pr(a,b,c,d,e,f,g,h,i,j,k) =$$

$$\Pr(a)$$

$$\times \Pr(b)$$

$$\times \Pr(c | a)$$

$$\times \Pr(d | a,b)$$

$$\times \Pr(e | c)$$

$$\times \Pr(f | d)$$

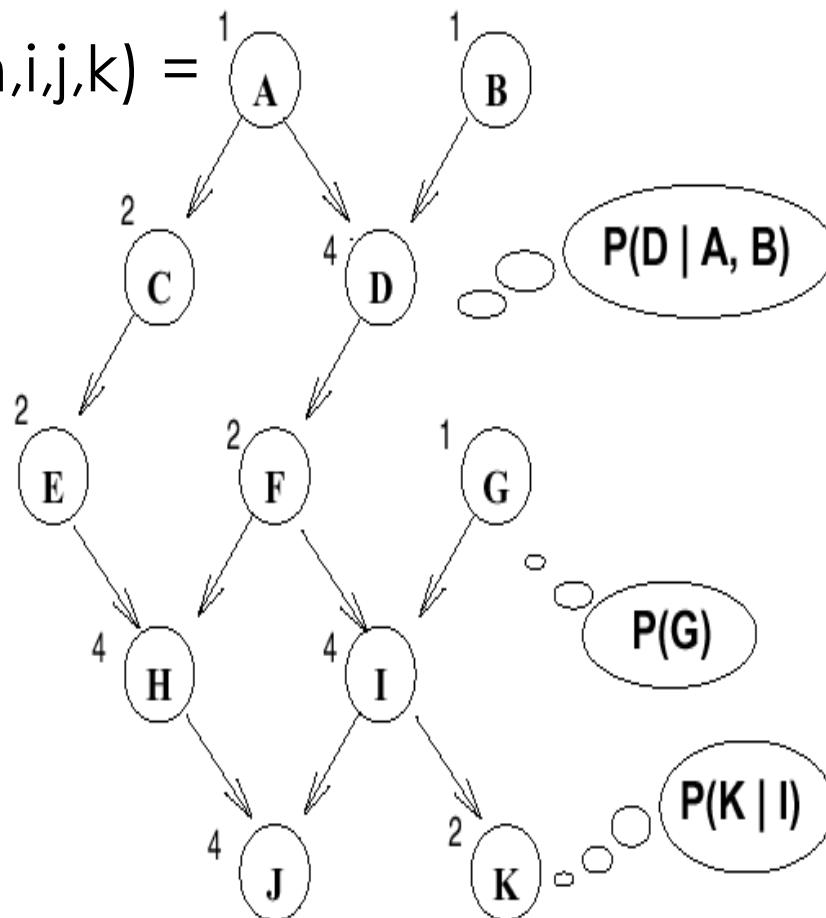
$$\times \Pr(g)$$

$$\times \Pr(h | e,f)$$

$$\times \Pr(i | f,g)$$

$$\times \Pr(j | h,i)$$

$$\times \Pr(k | i)$$



- ▶ Explicit joint requires $2^{11} - 1 = 2047$ parameters
- ▶ BN requires only 27 parameters (the number of entries for each CPT is listed)

Semantics of Bayes Nets.

- ▶ Note that this means we can compute the probability of any setting of the variables using only the information contained in the CPTs of the network.

Constructing a Bayes Net

- ▶ It is always possible to construct a Bayes net to represent any distribution over the variables X_1, X_2, \dots, X_n , using **any** ordering of the variables.
 - Take any ordering of the variables (say, the order given). From the chain rule we obtain.
$$\Pr(X_1, \dots, X_n) = \Pr(X_n | X_1, \dots, X_{n-1}) \Pr(X_{n-1} | X_1, \dots, X_{n-2}) \dots \Pr(X_1)$$
 - Now for each X_i go through its conditioning set X_1, \dots, X_{i-1} , and iteratively remove all variables X_j such that X_i is conditionally independent of X_j given the remaining variables. Do this until no more variables can be removed.
 - The final product will specify a Bayes net.

Constructing a Bayes Net

- ▶ The end result will be a product decomposition/Bayes net

$$\Pr(X_n \mid \text{Par}(X_n)) \Pr(X_{n-1} \mid \text{Par}(X_{n-1})) \dots \Pr(X_1)$$

- ▶ Now we specify the numeric values associated with each term $\Pr(X_i \mid \text{Par}(X_i))$ in a CPT.
- ▶ Typically we represent the CPT as a table mapping each setting of $\{X_i, \text{Par}(X_i)\}$ to the probability of X_i taking that particular value given that the variables in $\text{Par}(X_i)$ have their specified values.
- ▶ If each variable has d different values.
 - ▶ We will need a table of size $d^{|\{X_i, \text{Par}(X_i)\}|}$.
 - ▶ That is, exponential in the size of the parent set.
- ▶ Note that the original chain rule
 $\Pr(X_1, \dots, X_n) = \Pr(X_n \mid X_1, \dots, X_{n-1}) \Pr(X_{n-1} \mid X_1, \dots, X_{n-2}) \dots \Pr(X_1)$ requires as much space to represent as specifying the probability of each individual event.

Causal Intuitions

- ▶ The BN can be constructed using an arbitrary ordering of the variables.
- ▶ However, some orderings will yield BN's with very large parent sets. This requires exponential space, and (as we will see later) exponential time to perform inference.
- ▶ Empirically, and conceptually, a good way to construct a BN is to use an ordering based on causality. This often yields a more natural and compact BN.

Causal Intuitions

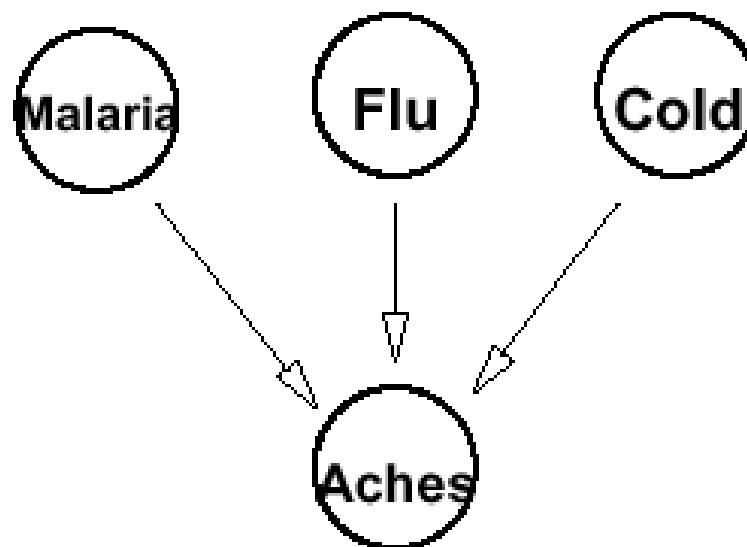
- ▶ Malaria, the flu and a cold all “cause” aches. So use the ordering that causes come before effects
Malaria, Flu, Cold, Aches

$$\Pr(M,F,C,A) = \Pr(A | M,F,C) \Pr(C | M,F) \Pr(F | M) \Pr(M)$$

- ▶ Each of these diseases affects the probability of aches, so the first conditional probability does not change.
- ▶ It is reasonable to assume that these diseases are independent of each other: having or not having one does not change the probability of having the others. So $\Pr(C | M,F) = \Pr(C)$
 $\Pr(F | M) = \Pr(F)$

Causal Intuitions

- ▶ This yields a fairly simple Bayes net.
- ▶ Only need one big CPT, involving the family of “Aches”.



Causal Intuitions

- ▶ Suppose we build the BN for distribution P using the opposite ordering
 - ▶ i.e., we use ordering Aches, Cold, Flu, Malaria

$$\Pr(A,C,F,M) = \Pr(M | A,C,F) \Pr(F | A,C) \Pr(C | A) \Pr(A)$$

- ▶ We can't reduce $\Pr(M | A,C,F)$. Probability of Malaria is clearly affected by knowing aches. What about knowing aches and Cold, or aches and Cold and Flu?
 - ▶ Probability of Malaria is affected by both of these additional pieces of knowledge

Knowing Cold and of Flu lowers the probability of Aches indicating Malaria since they “explain away” Aches!

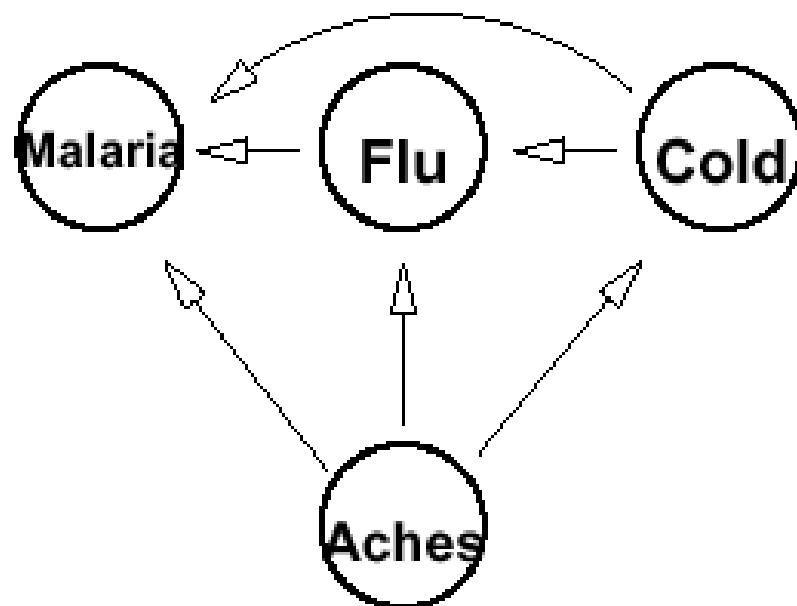
Causal Intuitions

$$\Pr(A,C,F,M) = \Pr(M | A,C,F) \Pr(F | A,C) \Pr(C | A) \Pr(A)$$

- ▶ Similarly, we can't reduce $\Pr(F | A,C)$.
- ▶ $\Pr(C | A) \neq \Pr(C)$

Causal Intuitions

- ▶ Obtain a much more complex Bayes net. In fact, we obtain no savings over explicitly representing the full joint distribution (i.e., representing the probability of every atomic event).

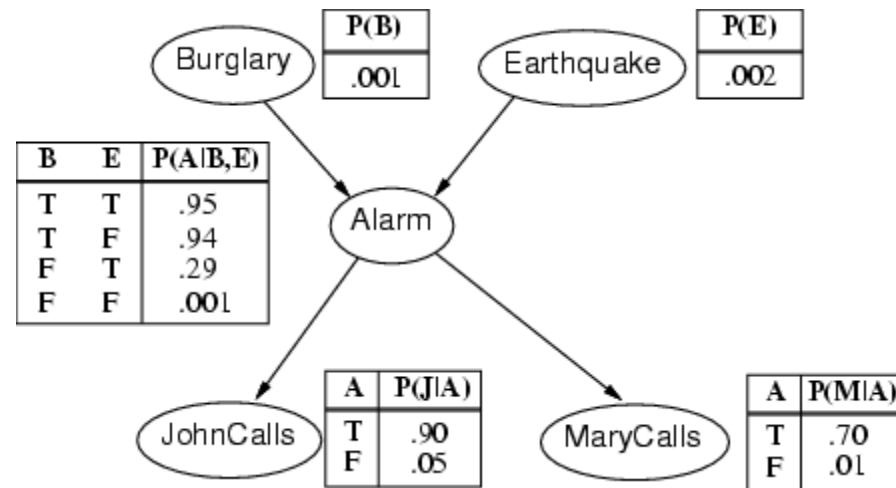


Bayes Net Examples

- ▶ I'm at work, neighbor John calls to say my alarm is ringing, but neighbor Mary doesn't call. Sometimes it's set off by minor earthquakes. Is there a burglar?
- ▶ Variables: *Burglary*, *Earthquake*, *Alarm*, *JohnCalls*, *MaryCalls*
- ▶ Network topology reflects "causal" knowledge:
 - ▶ A burglar can set the alarm off
 - ▶ An earthquake can set the alarm off
 - ▶ The alarm can cause Mary to call
 - ▶ The alarm can cause John to call

Burglary Example

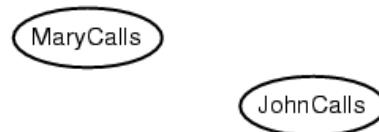
- A burglary can set the alarm off
- An earthquake can set the alarm off
- The alarm can cause Mary to call
- The alarm can cause John to call



- # of Params: $1 + 1 + 4 + 2 + 2 = 10$ (vs. $2^5 - 1 = 31$)

Example of Constructing Bayes Network

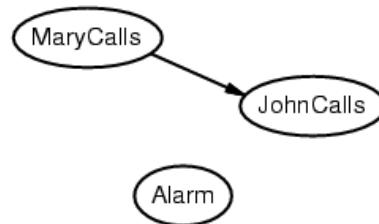
- ▶ Suppose we choose the ordering M, J, A, B, E
- ▶



$$P(J \mid M) = P(J) ?$$

Example continue...

- ▶ Suppose we choose the ordering M, J, A, B, E
- ▶



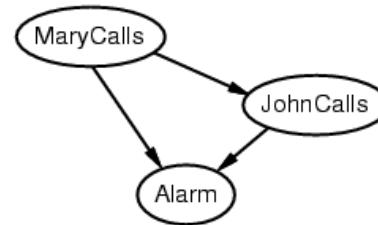
$$P(J \mid M) = P(J) ?$$

No

$$P(A \mid J, M) = P(A \mid J) ? \quad P(A \mid J, M) = P(A) ?$$

Example continue...

- ▶ Suppose we choose the ordering M, J, A, B, E
- ▶



$$P(J \mid M) = P(J) ?$$

Burglary

No

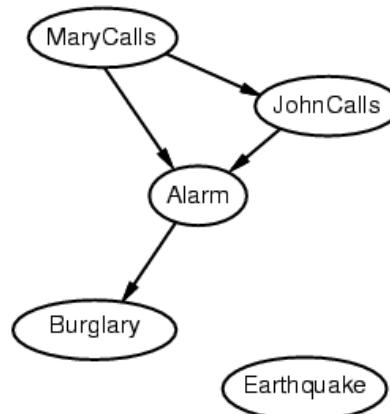
$$P(A \mid J, M) = P(A \mid J) ? \quad P(A \mid J, M) = P(A) ? \quad \text{No}$$

$$P(B \mid A, J, M) = P(B \mid A) ?$$

$$P(B \mid A, J, M) = P(B) ?$$

Example continue...

- ▶ Suppose we choose the ordering M, J, A, B, E
- ▶



$$P(J \mid M) = P(J)?$$

No

$$P(A \mid J, M) = P(A \mid J)? \quad P(A \mid J, M) = P(A)? \quad \textbf{No}$$

$$P(B \mid A, J, M) = P(B \mid A)? \quad \textbf{Yes}$$

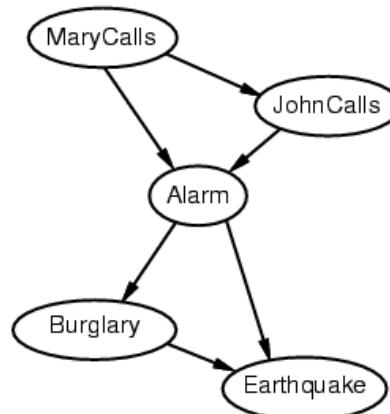
$$P(B \mid A, J, M) = P(B)? \quad \textbf{No}$$

$$P(E \mid B, A, J, M) = P(E \mid A)?$$

$$P(E \mid B, A, J, M) = P(E \mid A, B)?$$

Example continue...

- ▶ Suppose we choose the ordering M, J, A, B, E
- ▶



$$P(J \mid M) = P(J)?$$

No

$$P(A \mid J, M) = P(A \mid J)? \quad P(A \mid J, M) = P(A)? \quad \textbf{No}$$

$$P(B \mid A, J, M) = P(B \mid A)? \quad \textbf{Yes}$$

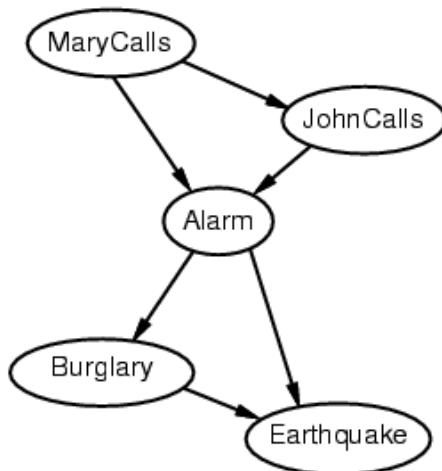
$$P(B \mid A, J, M) = P(B)? \quad \textbf{No}$$

$$P(E \mid B, A, J, M) = P(E \mid A)? \quad \textbf{No}$$

$$P(E \mid B, A, J, M) = P(E \mid A, B)? \quad \textbf{Yes}$$

Example continue...

- ▶ Deciding conditional independence **is hard** in non-causal directions!
- ▶ (Causal models and conditional independence seem hardwired for humans!)
- ▶ Network is **less compact**: $1 + 2 + 4 + 2 + 4 = 13$ numbers needed



Inference in Bayes Nets

- ▶ Given a Bayes net

$$\Pr(X_1, X_2, \dots, X_n)$$

$$= \Pr(X_n \mid \text{Par}(X_n)) * \Pr(X_{n-1} \mid \text{Par}(X_{n-1})) \\ * \dots * \Pr(X_1 \mid \text{Par}(X_1))$$

- ▶ And some evidence $E = \{\text{a set of values for some of the variables}\}$ we want to compute the new probability distribution

$$\Pr(X_k \mid E)$$

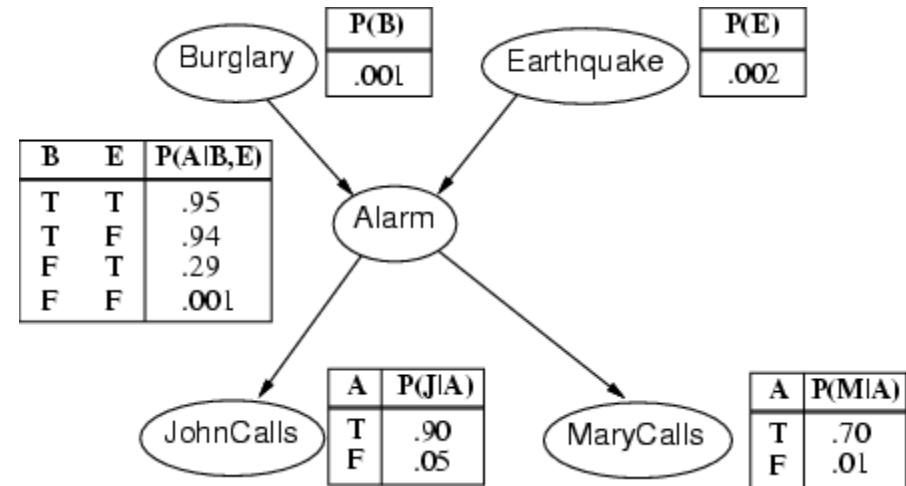
- ▶ That is, we want to figure our $\Pr(X_k = d \mid E)$ for all $d \in \text{Dom}[X_k]$

Inference in Bayes Nets

- ▶ Other types of examples are, computing probability of different diseases given symptoms, computing probability of hail storms given different metrological evidence, etc.
- ▶ In such cases getting a good estimate of the probability of the unknown event allows us to respond more effectively (gamble rationally)

Inference in Bayes Nets

- ▶ In the Alarm example:



- ▶ $\Pr(\text{Burglary}, \text{Earthquake}, \text{Alarm}, \text{JohnCalls}, \text{MaryCalls}) = \Pr(\text{Earthquake}) * \Pr(\text{Burglary}) * \Pr(\text{Alarm} | \text{Earthquake}, \text{Burglary}) * \Pr(\text{JohnCalls} | \text{Alarm}) * \Pr(\text{MaryCalls} | \text{Alarm})$
- ▶ And, e.g., we want to compute things like $\Pr(\text{Burglary}=\text{True} | \text{MaryCalls}=\text{false}, \text{JohnCalls}=\text{true})$

Variable Elimination

- ▶ Variable elimination uses the product decomposition that defines the Bayes Net and the summing out rule to compute posterior probabilities from the information (CPTs) already in the network.

Example (Binary valued Variables)

$$\Pr(A, B, C, D, E, F, G, H, I, J, K) =$$

$$\Pr(A)$$

$$\times \Pr(B)$$

$$\times \Pr(C | A)$$

$$\times \Pr(D | A, B)$$

$$\times \Pr(E | C)$$

$$\times \Pr(F | D)$$

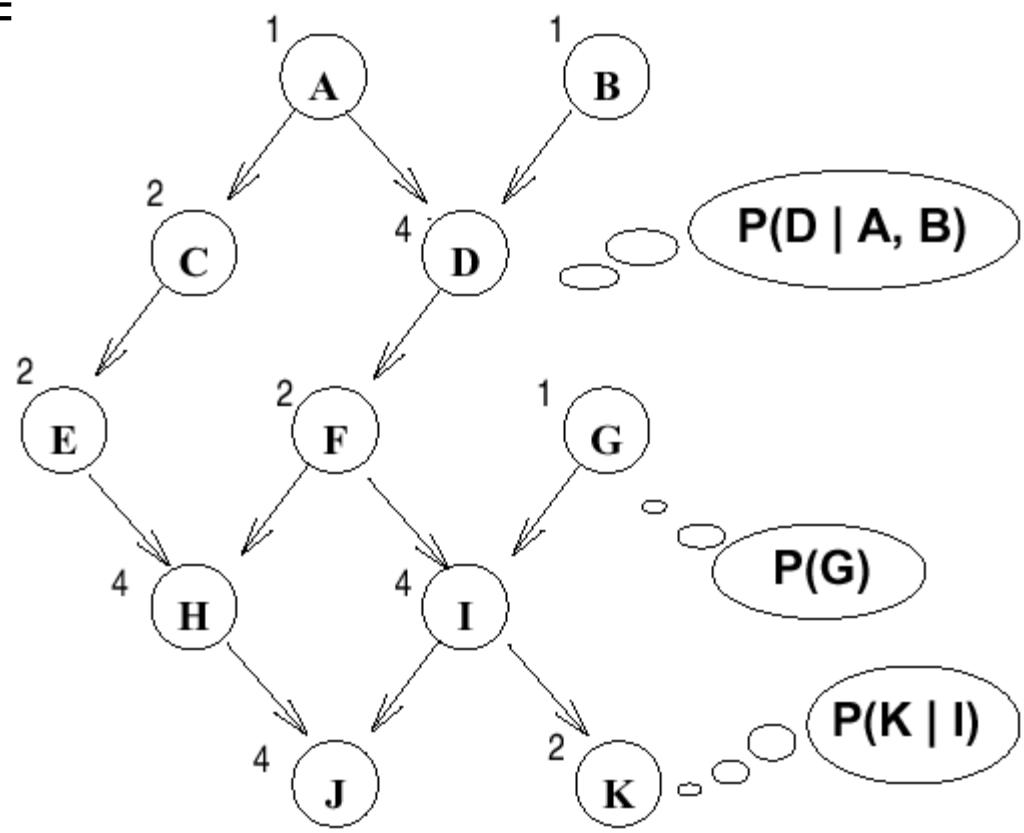
$$\times \Pr(G)$$

$$\times \Pr(H | E, F)$$

$$\times \Pr(I | F, G)$$

$$\times \Pr(J | H, I)$$

$$\times \Pr(K | I)$$



Example

$$\begin{aligned}\Pr(A, B, C, D, E, F, G, H, I, J, K) = \\ \Pr(A)\Pr(B)\Pr(C | A)\Pr(D | A, B)\Pr(E | C)\Pr(F | D)\Pr(G) \\ \Pr(H | E, F)\Pr(I | F, G)\Pr(J | H, I)\Pr(K | I)\end{aligned}$$

Say that $E = \{H=\text{true}, I=\text{false}\}$, and we want to know
 $\Pr(D | h, i)$ (h : H is true, $-h$: H is false)

1. Write as a sum for each value of D

$$\begin{aligned}\sum_{A,B,C,E,F,G,J,K} \Pr(A, B, C, d, E, F, h, -i, J, K) \\ = \Pr(d, h, -i)\end{aligned}$$

$$\begin{aligned}\sum_{A,B,C,E,F,G,J,K} \Pr(A, B, C, -d, E, F, h, -i, J, K) \\ = \Pr(-d, h, -i)\end{aligned}$$

Example

2. $\Pr(d,h,-i) + \Pr(-d,h,-i) = \Pr(h,-i)$

3. $\Pr(d|h,-i) = \Pr(d,h,-i)/\Pr(h,-i)$
 $\Pr(-d|h,-i) = \Pr(-d,h,-i)/\Pr(h,-i)$

So we only need to compute $\Pr(d,h,-i)$ and $\Pr(-d,h,-i)$ and then normalize to obtain the conditional probabilities we want.

Example

$$\Pr(d, h, \neg i) = \sum_{A,B,C,E,F,G,J,K} \Pr(A, B, C, d, E, F, h, \neg i, J, K)$$

Use Bayes Net product decomposition to rewrite summation:

$$\begin{aligned} & \sum_{A,B,C,E,F,G,J,K} \Pr(A, B, C, d, E, F, h, \neg i, J, K) \\ &= \sum_{A,B,C,E,F,G,J,K} \Pr(A)\Pr(B)\Pr(C | A)\Pr(d | A, B)\Pr(E | C) \\ & \quad \Pr(F | d)\Pr(G)\Pr(h | E, F)\Pr(\neg i | F, G)\Pr(J | h, \neg i) \\ & \quad \Pr(K | \neg i) \end{aligned}$$

Now rearrange summations so that we are not summing over that do not depend on the summed variable.

Example

$$= \sum_A \sum_B \sum_C \sum_E \sum_F \sum_G \sum_J \sum_K \\ \Pr(A) \Pr(B) \Pr(C | A) \Pr(d | A, B) \Pr(E | C) \\ \Pr(F | d) \Pr(G) \Pr(h | E, F) \Pr(-i | F, G) \Pr(J | h, -i) \\ \Pr(K | -i)$$

$$= \sum_A \Pr(A) \sum_B \Pr(B) \sum_C \Pr(C | A) \Pr(d | A, B) \sum_E \Pr(E | C) \\ \sum_F \Pr(F | d) \sum_G \Pr(G) \Pr(h | E, F) \Pr(-i | F, G) \sum_J \Pr(J | h, -i) \\ \sum_K \Pr(K | -i)$$

$$= \sum_A \Pr(A) \sum_B \Pr(B) \Pr(d | A, B) \sum_C \Pr(C | A) \sum_E \Pr(E | C) \\ \sum_F \Pr(F | d) \Pr(h | E, F) \sum_G \Pr(G) \Pr(-i | F, G) \sum_J \Pr(J | h, -i) \\ \sum_K \Pr(K | -i)$$

Example

- ▶ Now start computing.

$$\begin{aligned} & \sum_A \Pr(A) \sum_B \Pr(B) \Pr(d | A, B) \sum_C \Pr(C | A) \sum_E \Pr(E | C) \\ & \quad \sum_F \Pr(F | d) \Pr(h | E, F) \sum_G \Pr(G) \Pr(-i | F, G) \\ & \quad \sum_J \Pr(J | h, -i) \\ & \quad \sum_K \Pr(K | -i) \end{aligned}$$

$$\sum_K \Pr(K | -i) = \Pr(k | -i) + \Pr(-k | -i) = c_1$$

$$\begin{aligned} \sum_J \Pr(J | h, -i) c_1 &= c_1 \sum_J \Pr(J | h, -i) \\ &= c_1 (\Pr(j | h, -i) + \Pr(-j | h, -i)) \\ &= c_1 c_2 \end{aligned}$$

Example



$$\begin{aligned} & \sum_A \Pr(A) \sum_B \Pr(B) \Pr(d | A,B) \sum_C \Pr(C | A) \sum_E \Pr(E | C) \\ & \quad \sum_F \Pr(F | d) \Pr(h | E,F) \sum_G \Pr(G) \Pr(-i | F,G) \\ & \quad \sum_J \Pr(J | h,-i) \\ & \quad \sum_K \Pr(K | -i) \end{aligned}$$

$$\begin{aligned} & c_1 c_2 \sum_G \Pr(G) \Pr(-i | F,G) \\ & = c_1 c_2 (\Pr(g) \Pr(-i | F,g) + \Pr(-g) \Pr(-i | F,-g)) \end{aligned}$$

!!But $\Pr(-i | F,g)$ depends on the value of F, so this is not a single number.

Example

- Try the other order of summing.

$$\begin{aligned} & \sum_A \Pr(A) \sum_B \Pr(B) \Pr(d | A, B) \sum_C \Pr(C | A) \sum_E \Pr(E | C) \\ & \quad \sum_F \Pr(F | d) \Pr(h | E, F) \sum_G \Pr(G) \Pr(-i | F, G) \\ & \quad \sum_J \Pr(J | h, -i) \\ & \quad \sum_K \Pr(K | -i) \end{aligned}$$

=

$$\begin{aligned} & \Pr(a) \sum_B \Pr(B) \Pr(d | a, B) \sum_C \Pr(C | a) \sum_E \Pr(E | C) \\ & \quad \sum_F \Pr(F | d) \Pr(h | E, F) \sum_G \Pr(G) \Pr(-i | F, G) \\ & \quad \sum_J \Pr(J | h, -i) \\ & \quad \sum_K \Pr(K | -i) \end{aligned}$$

+

$$\begin{aligned} & \Pr(-a) \sum_B \Pr(B) \Pr(d | -a, B) \sum_C \Pr(C | -a) \sum_E \Pr(E | C) \\ & \quad \sum_F \Pr(F | d) \Pr(h | E, F) \sum_G \Pr(G) \Pr(-i | F, G) \\ & \quad \sum_J \Pr(J | h, -i) \\ & \quad \sum_K \Pr(K | -i) \end{aligned}$$

Example

=

$$\Pr(a)\Pr(b) \Pr(d|a,b) \sum_C \Pr(C|a) \sum_E \Pr(E|C) \\ \sum_F \Pr(F|d) \Pr(h|E,F) \sum_G \Pr(G) \Pr(-i|F,G) \\ \sum_J \Pr(J|h,-i) \\ \sum_K \Pr(K|-i)$$

+

$$\Pr(a)\Pr(-b) \Pr(d|a,-b) \sum_C \Pr(C|a) \sum_E \Pr(E|C) \\ \sum_F \Pr(F|d) \Pr(h|E,F) \sum_G \Pr(G) \Pr(-i|F,G) \\ \sum_J \Pr(J|h,-i) \\ \sum_K \Pr(K|-i)$$

+

$$\Pr(-a)\Pr(b) \Pr(d|-a,b) \sum_C \Pr(C|-a) \sum_E \Pr(E|C) \\ \sum_F \Pr(F|d) \Pr(h|E,F) \sum_G \Pr(G) \Pr(-i|F,G) \\ \sum_J \Pr(J|h,-i) \\ \sum_K \Pr(K|-i)$$

+

$$\Pr(-a)\Pr(-b) \Pr(d|-a,-b) \sum_C \Pr(C|-a) \sum_E \Pr(E|C) \\ \sum_F \Pr(F|d) \Pr(h|E,F) \sum_G \Pr(G) \Pr(-i|F,G) \\ \sum_J \Pr(J|h,-i) \\ \sum_K \Pr(K|-i)$$

Example

=

Yikes! The size of the sum is doubling as we expand each variable (into $-v$ and v). This approach has exponential complexity.

But let's look a bit closer.

Example

=

$$\Pr(a)\Pr(b) \Pr(d | a,b) \sum_C \Pr(C | a) \sum_E \Pr(E | C) \\ \sum_F \Pr(F | d) \Pr(h | E,F) \sum_G \Pr(G) \Pr(-i | F,G) \\ \sum_J \Pr(J | h,-i) \\ \sum_K \Pr(K | -i)$$

■ Repeated subterm

+

$$\Pr(a)\Pr(-b) \Pr(d | a,-b) \sum_C \Pr(C | a) \sum_E \Pr(E | C) \\ \sum_F \Pr(F | d) \Pr(h | E,F) \sum_G \Pr(G) \Pr(-i | F,G) \\ \sum_J \Pr(J | h,-i) \\ \sum_K \Pr(K | -i)$$

+

$$\Pr(-a)\Pr(b) \Pr(d | -a,b) \sum_C \Pr(C | -a) \sum_E \Pr(E | C) \\ \sum_F \Pr(F | d) \Pr(h | E,F) \sum_G \Pr(G) \Pr(-i | F,G) \\ \sum_J \Pr(J | h,-i) \\ \sum_K \Pr(K | -i)$$

+

$$\Pr(-a)\Pr(-b) \Pr(d | -a,-b) \sum_C \Pr(C | -a) \sum_E \Pr(E | C) \\ \sum_F \Pr(F | d) \Pr(h | E,F) \sum_G \Pr(G) \Pr(-i | F,G) \\ \sum_J \Pr(J | h,-i) \\ \sum_K \Pr(K | -i)$$

■ Repeated subterm

Dynamic Programming

- ▶ If we store the value of the subterms, we need only compute them once.

Dynamic Programming

$$= \Pr(a)\Pr(b) \Pr(d|a,b) \sum_C \Pr(C|a) \sum_E \Pr(E|C)$$

$$\sum_F \Pr(F|d) \Pr(h|E,F) \sum_G \Pr(G) \Pr(-i|F,G)$$

$$\sum_J \Pr(J|h,-i)$$

$$\sum_K \Pr(K|-i)$$


$$+ \Pr(a)\Pr(-b) \Pr(d|a,-b) \sum_C \Pr(C|a) \sum_E \Pr(E|C)$$

$$\sum_F \Pr(F|d) \Pr(h|E,F) \sum_G \Pr(G) \Pr(-i|F,G)$$

$$\sum_J \Pr(J|h,-i)$$

$$\sum_K \Pr(K|-i)$$

$$+ \Pr(-a)\Pr(b) \Pr(d|-a,b) \sum_C \Pr(C|-a) \sum_E \Pr(E|C)$$

$$\sum_F \Pr(F|d) \Pr(h|E,F) \sum_G \Pr(G) \Pr(-i|F,G)$$

$$\sum_J \Pr(J|h,-i)$$

$$\sum_K \Pr(K|-i)$$


$$+ \Pr(-a)\Pr(-b) \Pr(d|-a,-b) \sum_C \Pr(C|-a) \sum_E \Pr(E|C)$$

$$\sum_F \Pr(F|d) \Pr(h|E,F) \sum_G \Pr(G) \Pr(-i|F,G)$$

$$\sum_J \Pr(J|h,-i)$$

$$\sum_K \Pr(K|-i)$$

- $= c_1 f_1 + c_2 f_1 + c_3 f_2 + c_3 f_2$
- $c_1 = \Pr(a)\Pr(b) \Pr(d|a,b)$
- $c_2 = \Pr(a)\Pr(-b) \Pr(d|a,-b)$
- $c_3 = \Pr(a)\Pr(-b) \Pr(d|a,-b)$
- $c_4 = \Pr(a)\Pr(-b) \Pr(d|a,-b)$

Dynamic Programming

$$f_1 = \sum_C \Pr(C | a) \sum_E \Pr(E | C) \\ \sum_F \Pr(F | d) \Pr(h | E, F) \sum_G \Pr(G) \Pr(-i | F, G) \\ \sum_J \Pr(J | h, -i) \\ \sum_K \Pr(K | -i)$$

$$= \Pr(c | a) \sum_E \Pr(E | c) \\ \sum_F \Pr(F | d) \Pr(h | E, F) \sum_G \Pr(G) \Pr(-i | F, G) \\ \sum_J \Pr(J | h, -i) \\ \sum_K \Pr(K | -i)$$

+

$$\Pr(-c | a) \sum_E \Pr(E | -c) \\ \sum_F \Pr(F | d) \Pr(h | E, F) \sum_G \Pr(G) \Pr(-i | F, G) \\ \sum_J \Pr(J | h, -i) \\ \sum_K \Pr(K | -i)$$

■ Repeated
subterm

Dynamic Programming

- ▶ So within the computation of the subterms we obtain more repeated smaller subterms.
- ▶ The core idea of dynamic programming is to remember all “smaller” computations, so that they can be reused.
- ▶ This can convert an exponential computation into one that takes only polynomial time.
- ▶ Variable elimination is a dynamic programming technique that computes the sum from the bottom up (starting with the smaller subterms and working its way up to the bigger terms).

Relevant (return to this later)

- ▶ A brief aside is to also note that in the sum

$$\begin{aligned} & \sum_A \Pr(A) \sum_B \Pr(B) \Pr(d | A, B) \sum_C \Pr(C | A) \sum_E \Pr(E | C) \\ & \sum_F \Pr(F | d) \Pr(h | E, F) \sum_G \Pr(G) \Pr(-i | F, G) \\ & \sum_J \Pr(J | h, -i) \\ & \sum_K \Pr(K | -i) \end{aligned}$$

we have that $\sum_K \Pr(K | -i) = 1$ ([Why?](#)), thus
 $\sum_J \Pr(J | h, -i) \sum_K \Pr(K | -i) = \sum_J \Pr(J | h, -i)$

Furthermore $\sum_J \Pr(J | h, -i) = 1$.

So we could drop these last two terms from the computation---J and K are not relevant given our query D and our evidence $-i$ and $-h$. For now we keep these terms.

Variable Elimination (VE)

- ▶ VE works from the inside out, summing out K, then J, then G, ..., as we tried to before.
- ▶ When we tried to sum out G

$$\begin{aligned} & \sum_A \Pr(A) \sum_B \Pr(B) \Pr(d | A, B) \sum_C \Pr(C | A) \sum_E \Pr(E | C) \\ & \quad \sum_F \Pr(F | d) \Pr(h | E, F) \sum_G \Pr(G) \Pr(-i | F, G) \\ & \quad \sum_J \Pr(J | h, -i) \\ & \quad \sum_K \Pr(K | -i) \end{aligned}$$

$$\begin{aligned} & c_1 c_2 \sum_G \Pr(G) \Pr(-i | F, G) \\ & = c_1 c_2 (\Pr(g) \Pr(-i | F, g) + \Pr(-g) \Pr(-i | F, -g)) \end{aligned}$$

we found that $\Pr(-i | F, -g)$ depends on the value of F, it wasn't a single number.

- ▶ However, we can still continue with the computation by computing **two** different numbers, one for each value of F (-f, f)!

Variable Elimination (VE)

▶ $t(\neg f) = c_1 c_2 \sum_G \Pr(G) \Pr(\neg i \mid \neg f, G)$

$$t(f) = c_1 c_2 (\sum_G \Pr(G) \Pr(\neg i \mid f, G))$$

- ▶ $t(\neg f) = c_1 c_2 (\Pr(g) \Pr(\neg i \mid \neg f, g) + \Pr(\neg g) \Pr(\neg i \mid \neg f, \neg g))$
- ▶ $t(\neg f) = c_1 c_2 (\Pr(g) \Pr(\neg i \mid f, g) + \Pr(\neg g) \Pr(\neg i \mid f, \neg g))$

▶ Now we sum out F

Variable Elimination (VE)

- ▶ $\sum_A \Pr(A) \sum_B \Pr(B) \Pr(d | A,B) \sum_C \Pr(C | A) \sum_E \Pr(E | C)$
 $\sum_F \Pr(F | d) \Pr(h | E,F) \sum_G \Pr(G) \Pr(-i | F,G)$
 $\sum_J \Pr(J | h,-i)$
 $\sum_K \Pr(K | -i)$

$$c_1 c_2 \sum_F \Pr(F | d) \Pr(h | E,F) \sum_G \Pr(G) \Pr(-i | F,G)$$

$$= c_1 c_2 (\Pr(f | d) \Pr(h | E,f) (\sum_G \Pr(G) \Pr(-i | f,G)) \\ + \Pr(-f | d) \Pr(h | E,-f) (\sum_G \Pr(G) \Pr(-i | -f,G)))$$

$$= c_1 c_2 \sum_F \Pr(F | d) \Pr(h | E,F) t(F)$$

$$t(f), t(-f)$$

Variable Elimination (VE)

- ▶ $c_1 c_2 (\Pr(f | d) \Pr(h | E, f) t(f) + \Pr(\neg f | d) \Pr(h | E, \neg f) t(\neg f))$
- ▶ This is a function of E , so we obtain two new numbers

$$s(e) = c_1 c_2 (\Pr(f | d) \Pr(h | e, f) t(f) + \Pr(\neg f | d) \Pr(h | e, \neg f) t(\neg f))$$

$$s(\neg e) = c_1 c_2 (\Pr(f | d) \Pr(h | \neg e, f) t(f) + \Pr(\neg f | d) \Pr(h | \neg e, \neg f) t(\neg f))$$

Variable Elimination (VE)

- ▶ On summing out E we obtain two numbers, or a function of C. Then a function of B, then a function of A. On finally summing out A we obtain the single number we wanted to compute which is $\text{Pr}(d,h,-i)$.
- ▶ Now we can repeat the process to compute $\text{Pr}(-d,h,-i)$.
- ▶ But instead of doing it twice, we can simply regard D as an variable in the computation.
- ▶ This will result in some computations depending on the value of D, and we obtain a different number for each value of D.
- ▶ Proceeding in this manner, summing out A will yield a function of D. (I.e., a number for each value of D).

Variable Elimination (VE)

- ▶ In general, at each stage VE will be compute a table of numbers: one number for each different instantiation of the variables that are in the sum.
- ▶ The size of these tables is exponential in the number of variables appearing in the sum, e.g.,

$$\sum_F \Pr(F \mid D) \Pr(h \mid E, F) t(F)$$

depends on the value of D and E, thus we will obtain $| \text{Dom}[D] | * | \text{Dom}[E] |$ different numbers in the resulting table.

Factors

- ▶ we call these tables of values computed by VE factors.
- ▶ Note that the original probabilities that appear in the summation, e.g., $P(C | A)$, are also tables of values (one value for each instantiation of C and A).
- ▶ Thus we also call the original CPTs factors.
- ▶ Each factor is a function of some variables, e.g., $P(C | A) = f(A, C)$: it maps each value of its arguments to a number.
 - ▶ A tabular representation is exponential in the number of variables in the factor.

Operations on Factors

- ▶ If we examine the inside-out summation process we see that various operations occur on factors.
- ▶ Notation: $f(\underline{X}, \underline{Y})$ denotes a factor over the variables $\underline{X} \cup \underline{Y}$ (where \underline{X} and \underline{Y} are sets of variables)

The Product of Two Factors

- Let $f(\mathbf{X}, \mathbf{Y})$ & $g(\mathbf{Y}, \mathbf{Z})$ be two factors with variables \mathbf{Y} in common
- The *product* of f and g , denoted $h = f \times g$ (or sometimes just $h = fg$), is defined:

$$h(\mathbf{X}, \mathbf{Y}, \mathbf{Z}) = f(\mathbf{X}, \mathbf{Y}) \times g(\mathbf{Y}, \mathbf{Z})$$

f(A,B)		g(B,C)		h(A,B,C)			
ab	0.9	bc	0.7	abc	0.63	ab~c	0.27
a~b	0.1	b~c	0.3	a~bc	0.08	a~b~c	0.02
~ab	0.4	~bc	0.8	~abc	0.28	~ab~c	0.12
~a~b	0.6	~b~c	0.2	~a~bc	0.48	~a~b~c	0.12

Summing a Variable Out of a Factor

- ▶ Let $f(X, Y)$ be a factor with variable X (Y is a set)
- ▶ We *sum out* variable X from f to produce a new factor $h = \sum_X f$, which is defined:

$$h(Y) = \sum_{x \in \text{Dom}(X)} f(x, Y)$$

$f(A, B)$		$h(B)$	
ab	0.9	b	1.3
a~b	0.1	~b	0.7
~ab	0.4		
~a~b	0.6		

Restricting a Factor

- ▶ Let $f(X, Y)$ be a factor with variable X (Y is a set)
- ▶ We *restrict* factor f to $X=a$ by setting X to the value x and “deleting” incompatible elements of f ’s domain . Define $h = f_{X=a}$ as: $h(Y) = f(a, Y)$

$f(A, B)$		$h(B) = f_{A=a}$	
ab	0.9	b	0.9
$a \sim b$	0.1	$\sim b$	0.1
$\sim ab$	0.4		
$\sim a \sim b$	0.6		

Variable Elimination the Algorithm

Given query var Q , evidence vars \mathbf{E} (set of variables observed to have values $\underline{\mathbf{e}}$), remaining vars \mathbf{Z} . Let F be original CPTs.

1. Replace each factor $f \in F$ that mentions a variable(s) in E with its restriction $f_{E=\underline{\mathbf{e}}}$ (this might yield a “constant” factor)
2. For each Z_j —in the order given—eliminate $Z_j \in Z$ as follows:
 - (a) Compute new factor $g_j = \sum_{Z_j} f_1 \times f_2 \times \dots \times f_k$, where the f_i are the factors in F that include Z_j
 - (b) Remove the factors f_i (that mention Z_j) from F and add new factor g_j to F
3. The remaining factors refer only to the query variable Q . Take their product and normalize to produce $Pr(Q|E)$

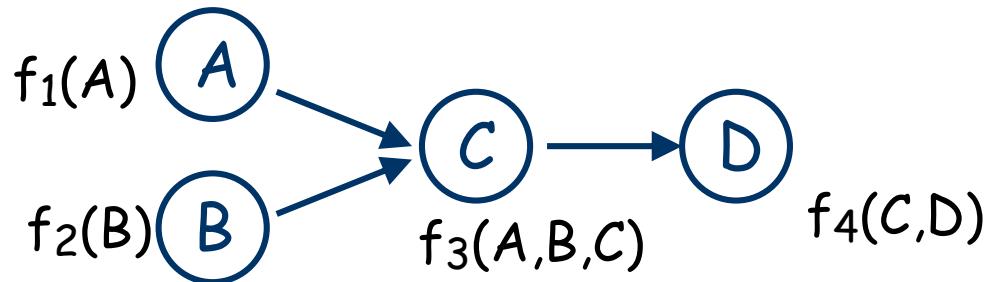
VE: Example

Factors: $f_1(A)$ $f_2(B)$ $f_3(A,B,C)$
 $f_4(C,D)$

Query: $P(A) ?$

Evidence: $D = d$

Elim. Order: C, B



Restriction: replace $f_4(C,D)$ with $f_5(C) = f_4(C,d)$

Step 1: Compute & Add $f_6(A,B) = \sum_C f_5(C) f_3(A,B,C)$

Remove: $f_3(A,B,C)$, $f_5(C)$

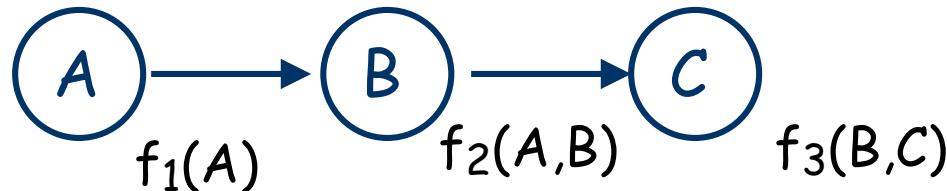
Step 2: Compute & Add $f_7(A) = \sum_B f_6(A,B) f_2(B)$

Remove: $f_6(A,B)$, $f_2(B)$

Last factors: $f_7(A)$, $f_1(A)$. The product $f_1(A) \times f_7(A)$ is (unnormalized) posterior. So... $P(A | d) = \alpha f_1(A) \times f_7(A)$
where $\alpha = 1 / \sum_A f_1(A) f_7(A)$

Numeric Example

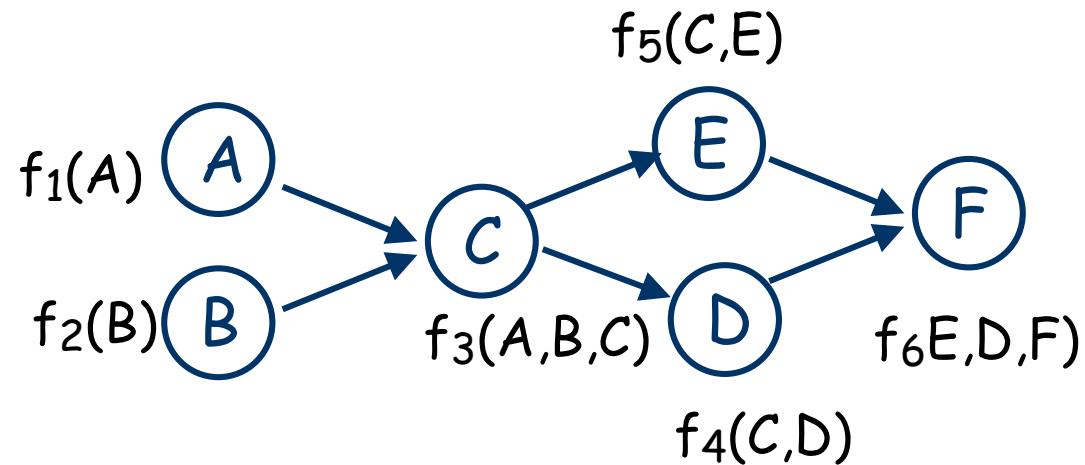
- Here's the example with some numbers



$f_1(A)$		$f_2(A,B)$		$f_3(B,C)$		$f_4(B)$ $\Sigma_A f_2(A,B)f_1(A)$		$f_5(C)$ $\Sigma_B f_3(B,C) f_4(B)$	
a	0.9	ab	0.9	bc	0.7	b	0.85	c	0.625
$\sim a$	0.1	$a \sim b$	0.1	$b \sim c$	0.3	$\sim b$	0.15	$\sim c$	0.375
		$\sim ab$	0.4	$\sim bc$	0.2				
		$\sim a \sim b$	0.6	$\sim b \sim c$	0.8				

VE: Buckets as a Notational Device

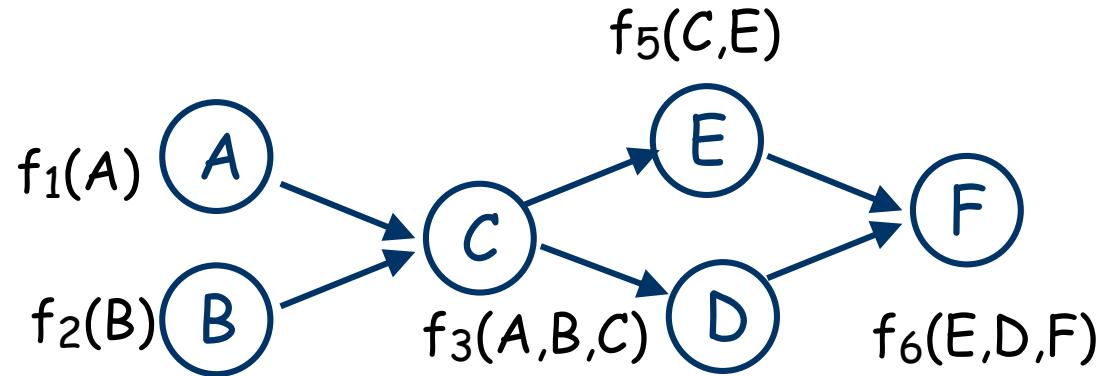
Ordering:
 C, F, A, B, E, D



1. C:
2. F:
3. A:
4. B:
5. E:
6. D:

VE: Buckets—Place Original Factors in first applicable bucket.

Ordering:
 C, F, A, B, E, D



1. $C: f_3(A, B, C), f_4(C, D), f_5(C, E)$

2. $F: f_6(E, D, F)$

3. $A: f_1(A)$

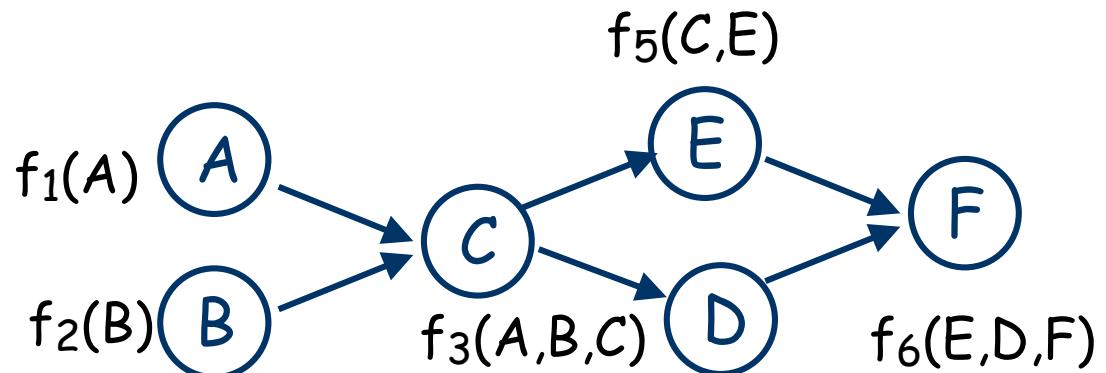
4. $B: f_2(B)$

5. $E:$

6. $D:$

VE: Eliminate the variables in order, placing new factor in first applicable bucket.

Ordering:
 C, F, A, B, E, D



1. ~~$C: f_3(A, B, C), f_4(C, D), f_5(C, E)$~~

2. $F: f_6(E, D, F)$

3. $A: f_1(A), f_7(A, B, D, E)$

4. $B: f_2(B)$

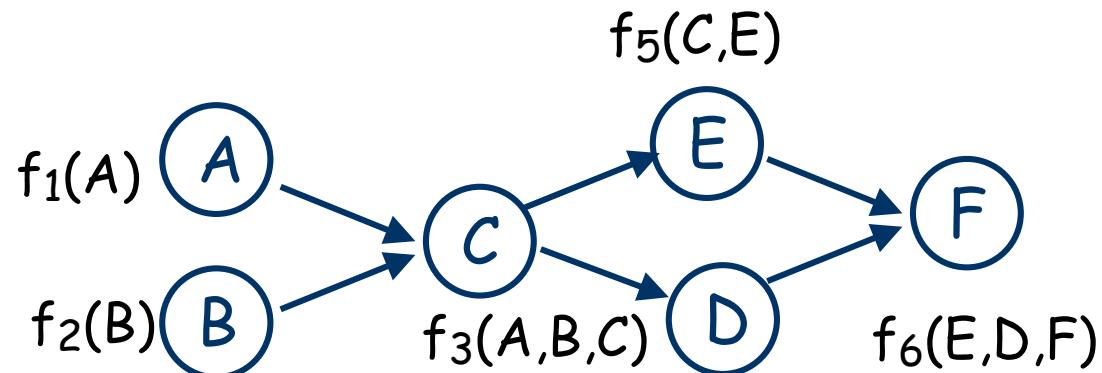
5. $E:$

6. $D:$

1. $\sum_C f_3(A, B, C), f_4(C, D), f_5(C, E)$
= $f_7(A, B, D, E)$

VE: Eliminate the variables in order, placing new factor in first applicable bucket.

Ordering:
 C, F, A, B, E, D



1. C : $f_3(A, B, C), f_4(C, D), f_5(C, E)$

2. F : $f_6(E, D, F)$

3. A : $f_1(A), f_7(A, B, D, E)$

4. B : $f_2(B)$

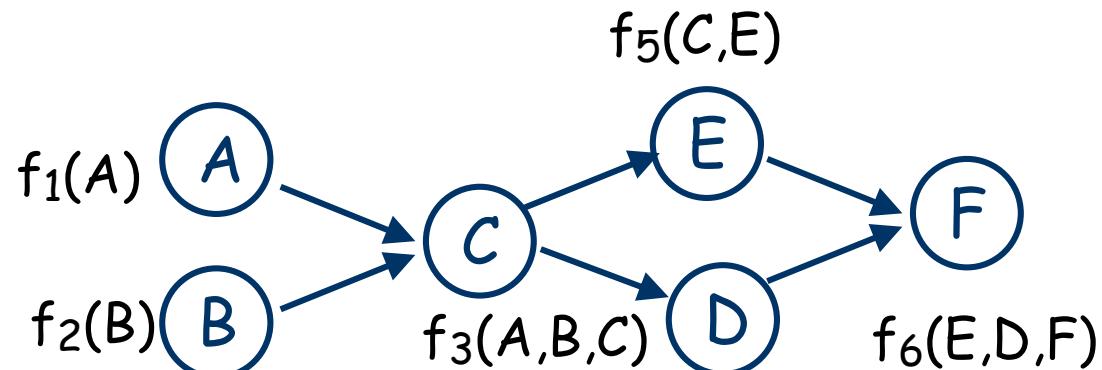
5. E : $f_8(E, D)$

6. D :

2. $\sum_F f_6(E, D, F) = f_8(E, D)$

VE: Eliminate the variables in order, placing new factor in first applicable bucket.

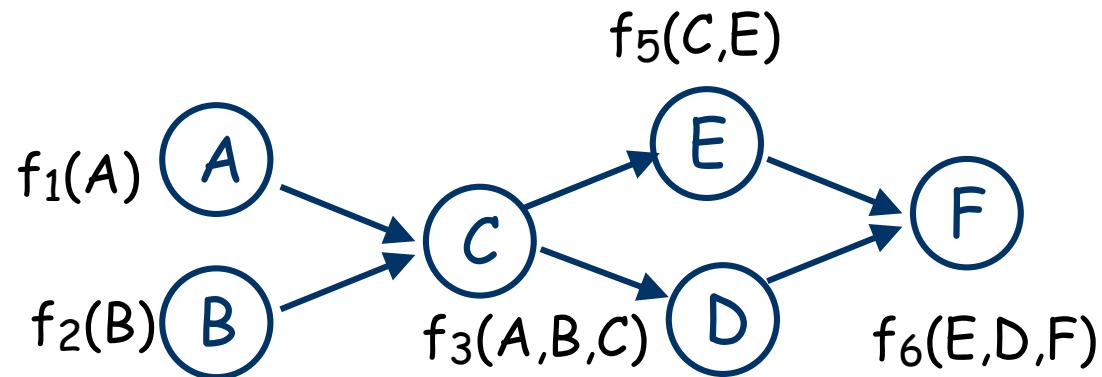
Ordering:
 C, F, A, B, E, D



1. ~~C: $f_3(A, B, C), f_4(C, D), f_5(C, E)$~~
2. ~~F: $f_6(E, D, F)$~~
3. ~~A: $f_1(A), f_7(A, B, D, E)$~~
4. B: $f_2(B), f_9(B, D, E)$
5. E: $f_8(E, D)$
6. D:

VE: Eliminate the variables in order, placing new factor in first applicable bucket.

Ordering:
 C, F, A, B, E, D



1. $C: f_3(A, B, C), f_4(C, D), f_5(C, E)$

2. $F: f_6(E, D, F)$

3. $A: f_1(A), f_7(A, B, D, E)$

4. $B: f_2(B), f_9(B, D, E)$

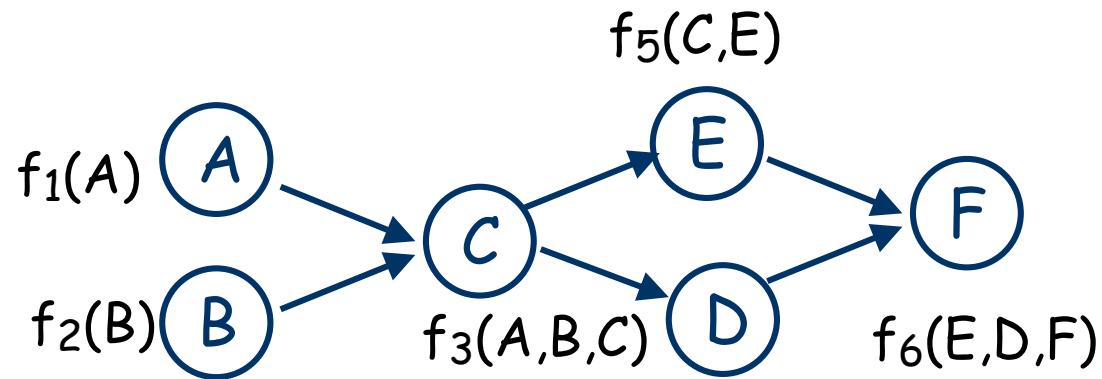
5. $E: f_8(E, D), \textcolor{blue}{f_{10}(D, E)}$

6. $D:$

4. $\sum_B f_2(B), f_9(B, D, E)$
= $f_{10}(D, E)$

VE: Eliminate the variables in order, placing new factor in first applicable bucket.

Ordering:
 C, F, A, B, E, D



1. $C: f_3(A, B, C), f_4(C, D), f_5(C, E)$

2. $F: f_6(E, D, F)$

3. $A: f_1(A), f_7(A, B, D, E)$

4. $B: f_2(B), f_9(B, D, E)$

5. $E: f_8(E, D), f_{10}(D, E)$

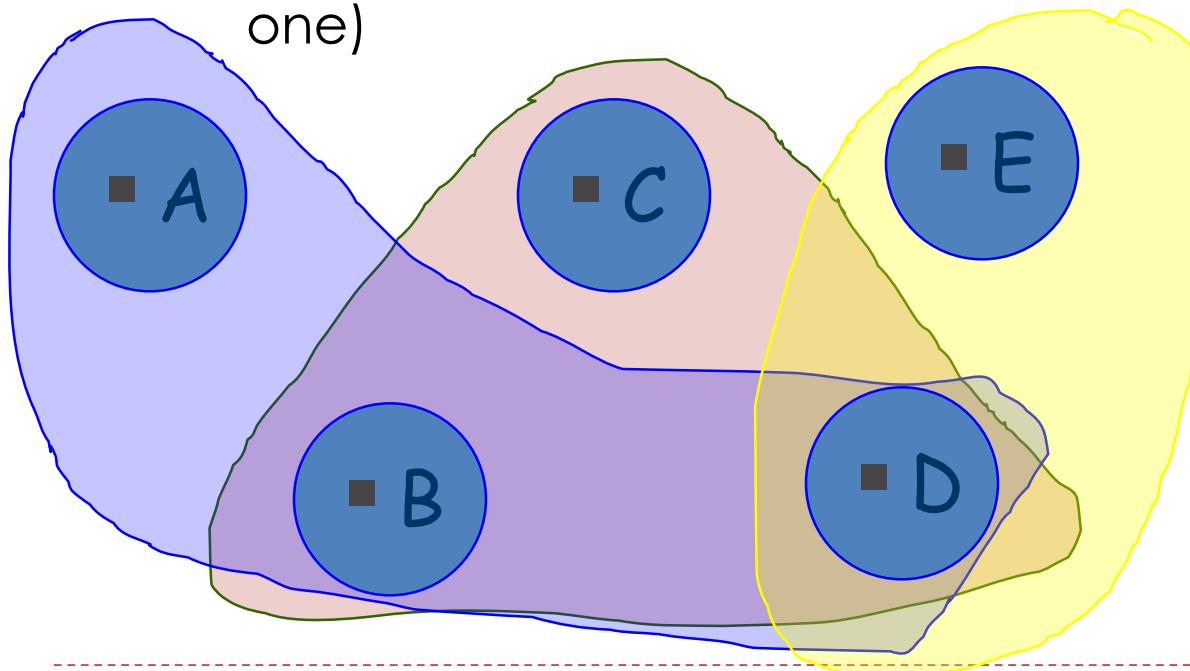
6. $D: f_{11}(D)$

5. $\sum_E f_8(E, D), f_{10}(D, E)$
= $f_{11}(D)$

f_{11} is the final answer, once we normalize it.

Complexity of Variable Elimination

- ▶ Hypergraph of Bayes Net.
 - ▶ Hypergraph has vertices just like an ordinary graph, but instead of edges between two vertices $X \leftrightarrow Y$ it contains **hyperedges**.
 - ▶ A hyperedge is a set of vertices (i.e., potentially more than one)



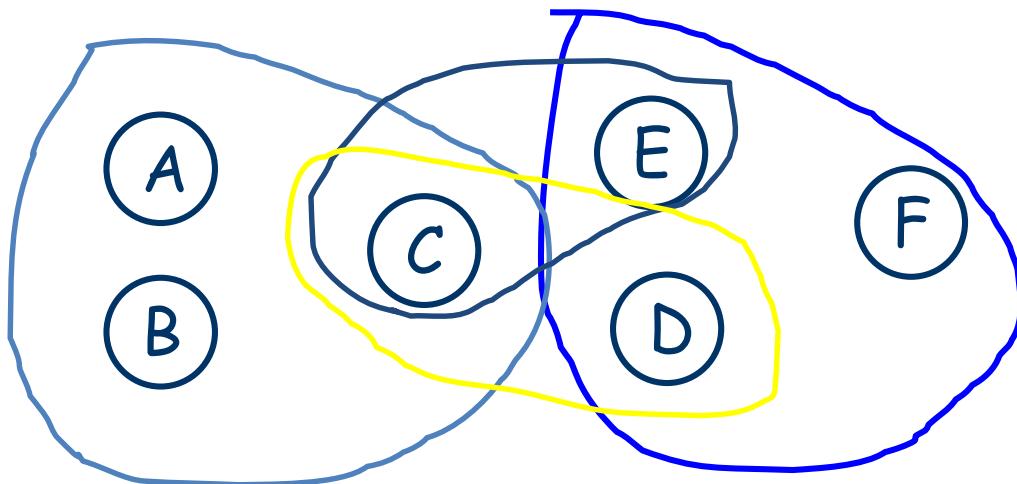
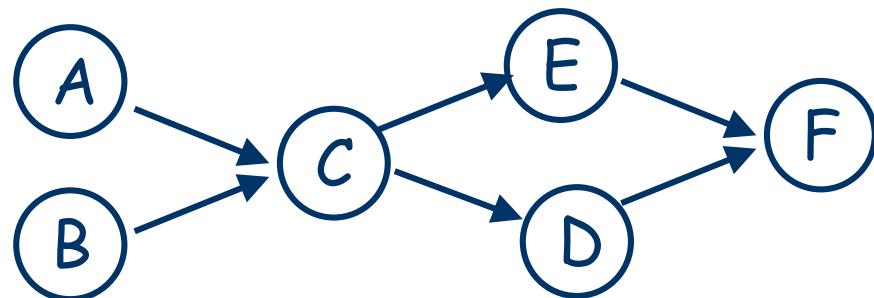
■ $\{A, B, D\}$
 $\{B, C, D\}$
 $\{E, D\}$

Complexity of Variable Elimination

- ▶ Hypergraph of Bayes Net.
 - ▶ The set of vertices are precisely the nodes of the Bayes net.
 - ▶ The hyperedges are the variables appearing in each CPT.
 - ▶ $\{X_i\} \cup \text{Par}(X_i)$

Complexity of Variable Elimination

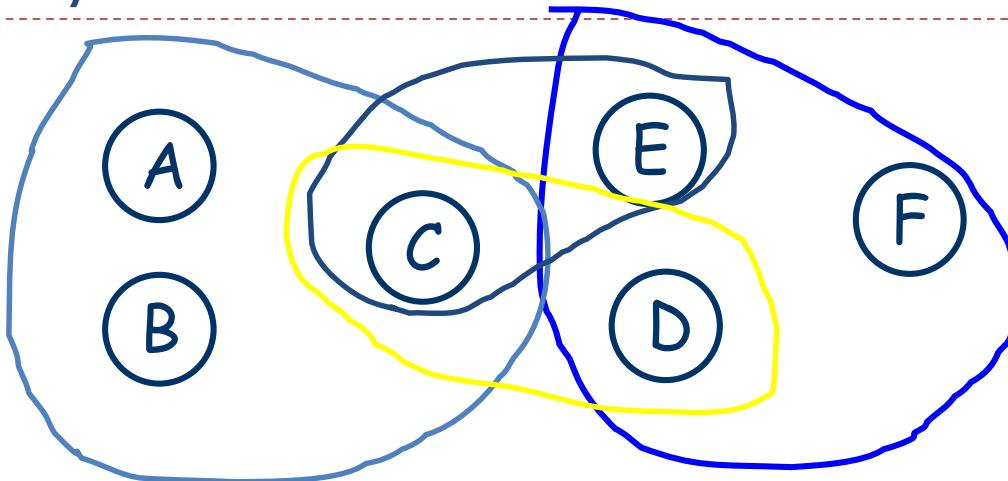
- ▶ $\Pr(A, B, C, D, E, F) =$
 $\Pr(A)\Pr(B)$
 $\times \Pr(C | A, B)$
 $\times \Pr(E | C)$
 $\times \Pr(D | C)$
 $\times \Pr(F | E, D).$



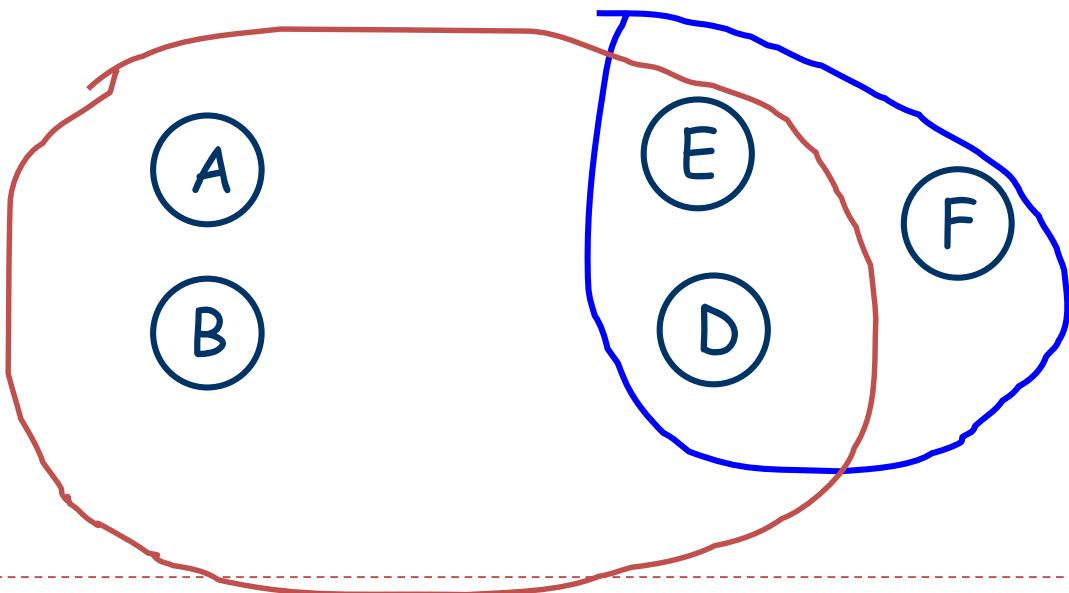
Variable Elimination in the HyperGraph

- ▶ To eliminate variable X_i in the hypergraph we
 - ▶ we remove the vertex X_i
 - ▶ Create a new hyperedge H_i equal to the union of all of the hyperedges that contain X_i minus X_i
 - ▶ Remove all of the hyperedges containing X_i from the hypergraph.
 - ▶ Add the new hyperedge H_i to the hypergraph.

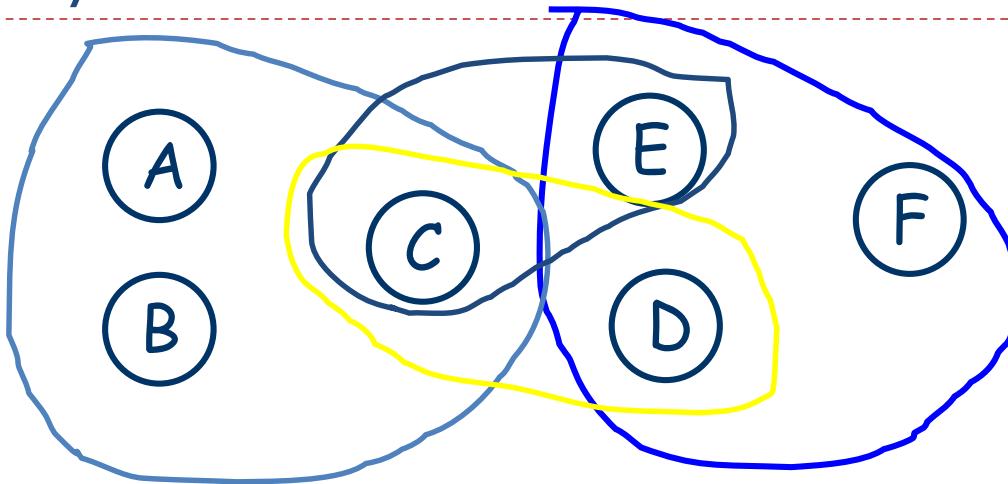
Complexity of Variable Elimination



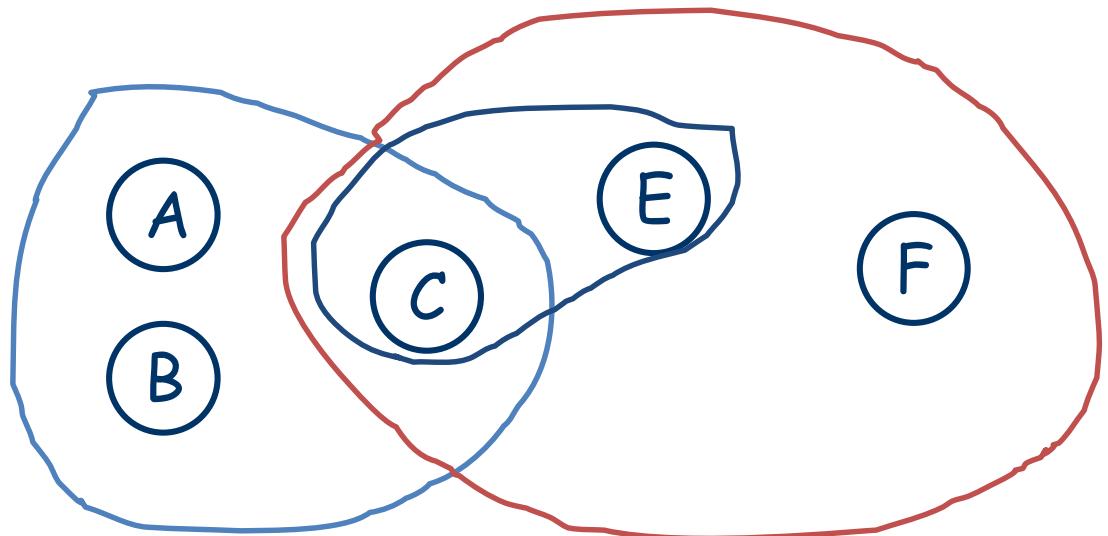
▶ Eliminate C



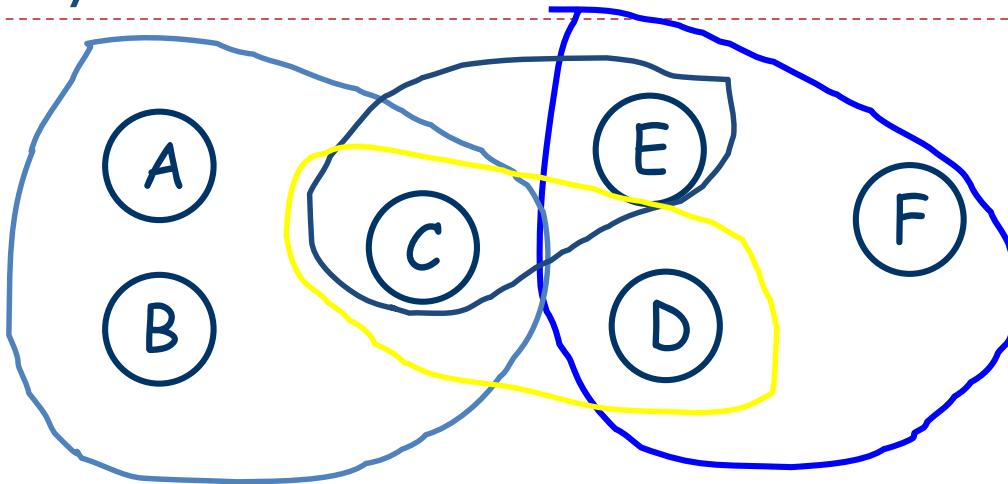
Complexity of Variable Elimination



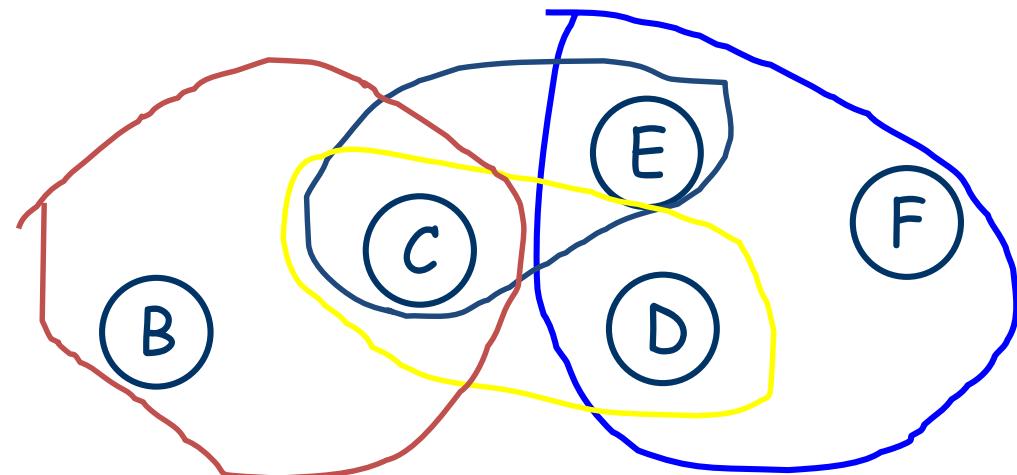
▶ Eliminate D



Complexity of Variable Elimination



▶ Eliminate A

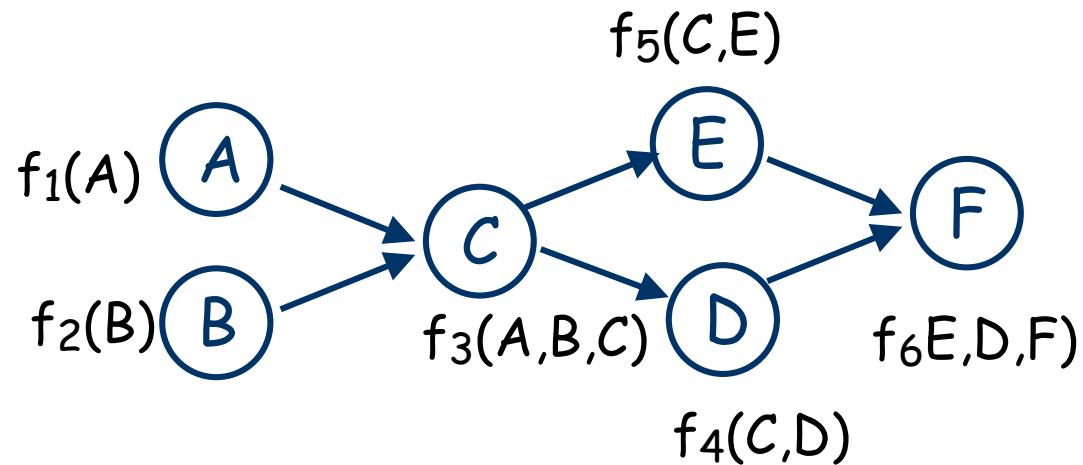


Variable Elimination

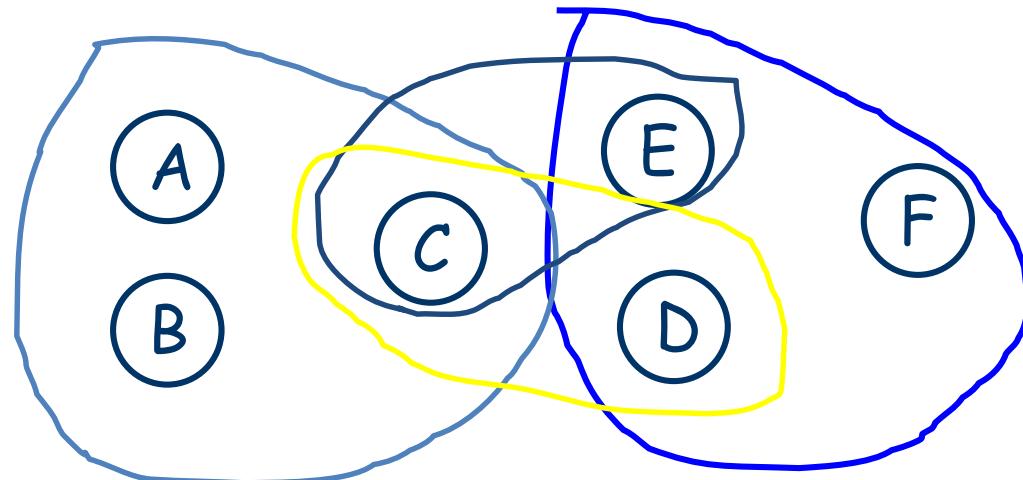
- ▶ Notice that when we start VE we have a set of factors consisting of the **reduced CPTs**. The unassigned variables for the vertices and the set of variables each factor depends on forms the hyperedges of a hypergraph H_1 .
- ▶ If the first variable we eliminate is X , then we remove all factors containing X (all hyperedges) and add a new factor that has as variables the union of the variables in the factors containing X (we add a hyperedge that is the union of the removed hyperedges minus X).

VE Factors

Ordering:
 C, F, A, B, E, D

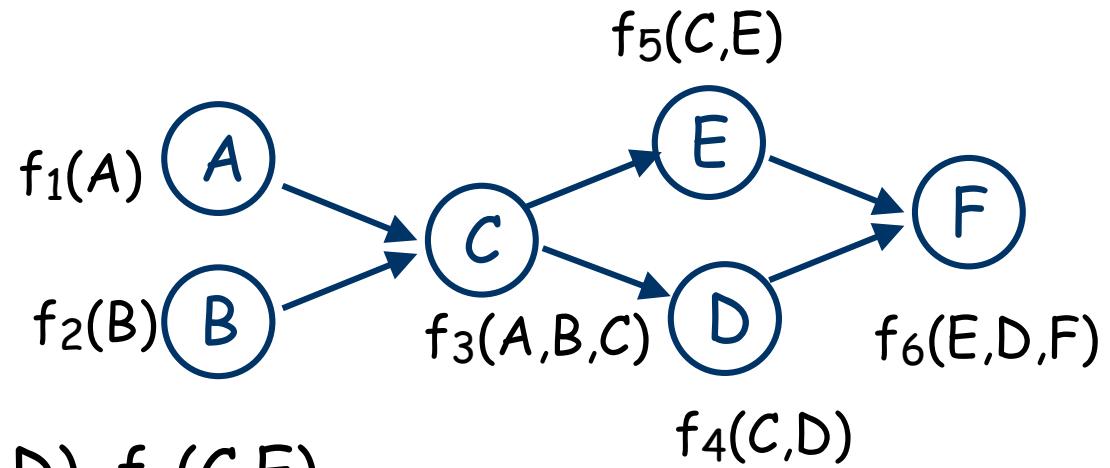


1. C:
2. F:
3. A:
4. B:
5. E:
6. D:



VE: Place Original Factors in first applicable bucket.

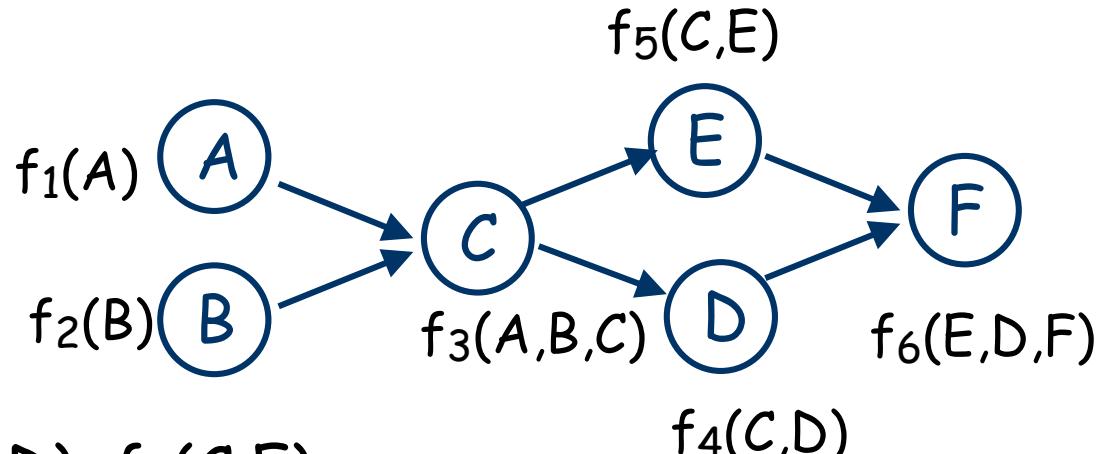
Ordering:
 C, F, A, B, E, D



1. $C: f_3(A, B, C), f_4(C, D), f_5(C, E)$
2. $F: f_6(E, D, F)$
3. $A: f_1(A)$
4. $B: f_2(B)$
5. $E:$
6. $D:$

VE: Eliminate C, placing new factor f_7 in first applicable bucket.

Ordering:
 C, F, A, B, E, D



1. $C: f_3(A,B,C), f_4(C,D), f_5(C,E)$

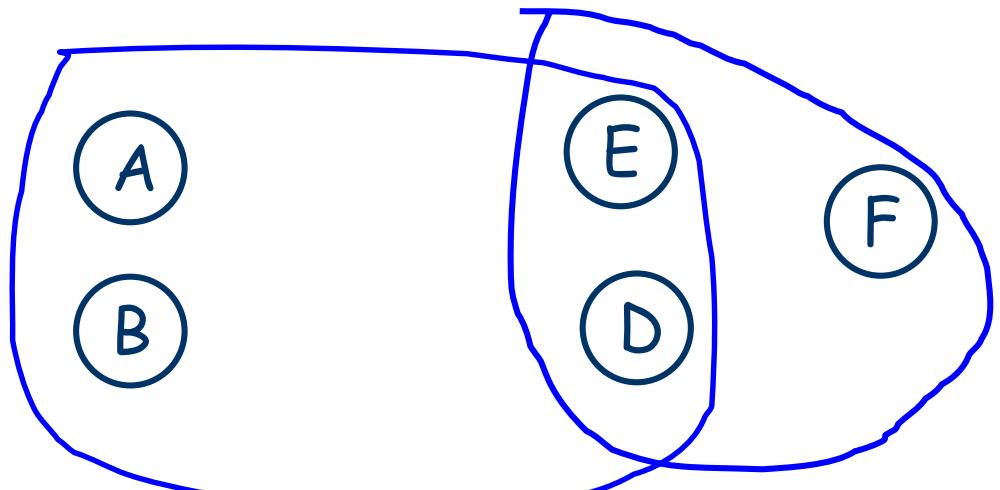
2. $F: f_6(E,D,F)$

3. $A: f_1(A), f_7(A,B,D,E)$

4. $B: f_2(B)$

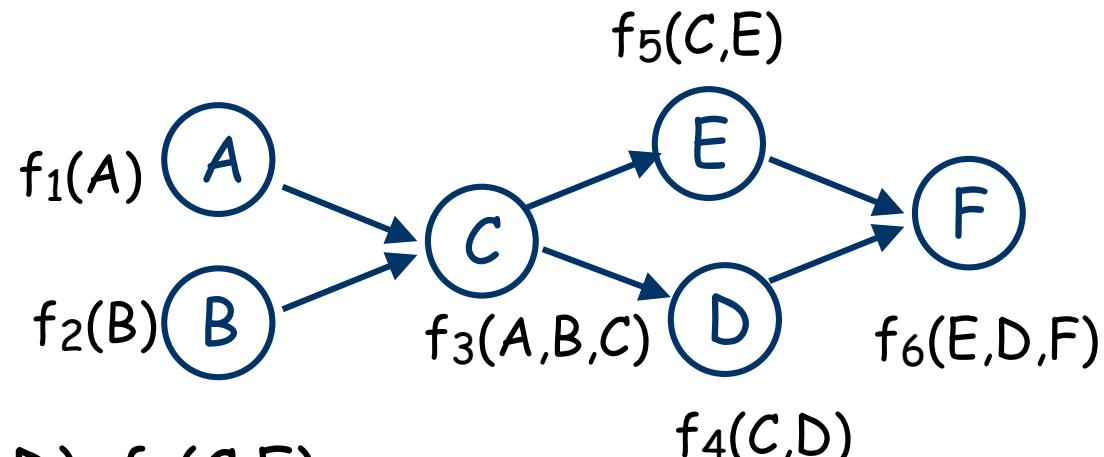
5. $E:$

6. $D:$

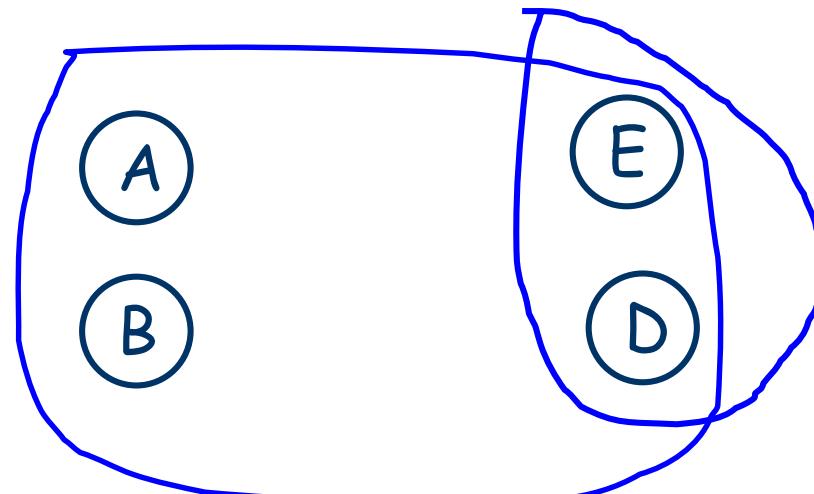


VE: Eliminate F, placing new factor f_8 in first applicable bucket.

Ordering:
 C, F, A, B, E, D

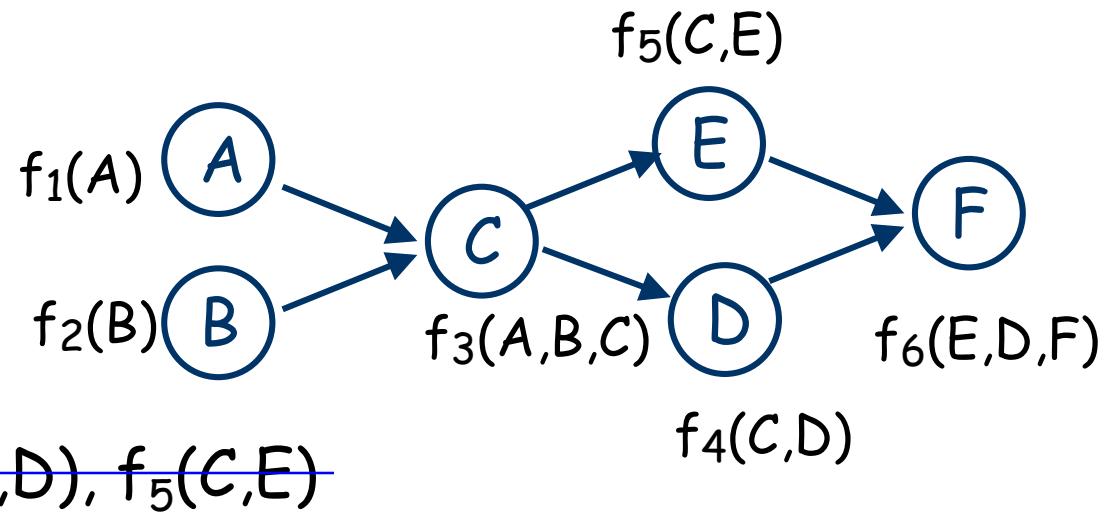


1. $C: f_3(A, B, C), f_4(C, D), f_5(C, E)$
2. $F: f_6(E, D, F)$
3. $A: f_1(A), f_7(A, B, D, E)$
4. $B: f_2(B)$
5. $E: f_8(E, D)$
6. $D:$

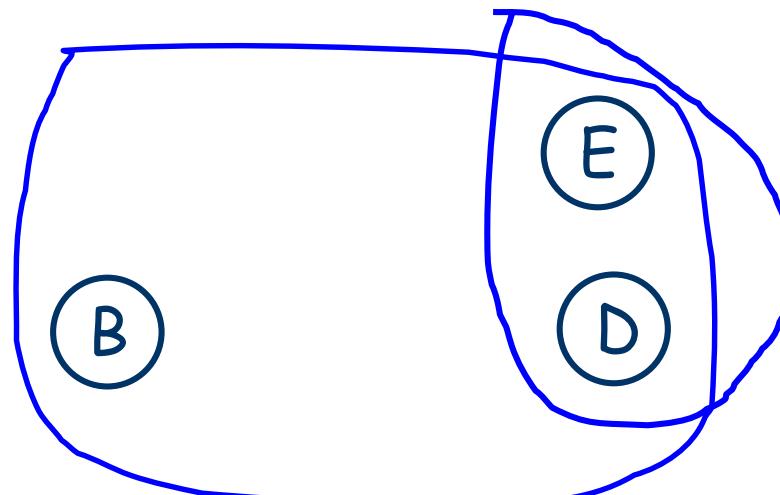


VE: Eliminate A, placing new factor f_9 in first applicable bucket.

Ordering:
 C, F, A, B, E, D

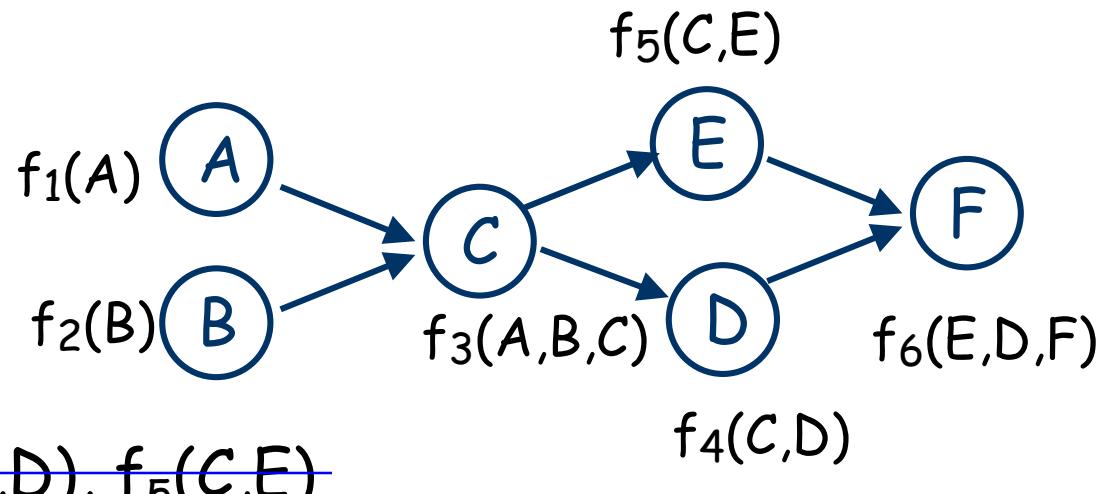


1. $C: f_3(A,B,C), f_4(C,D), f_5(C,E)$
2. $F: f_6(E,D,F)$
3. $A: f_1(A), f_7(A,B,D,E)$
4. $B: f_2(B), f_9(B,D,E)$
5. $E: f_8(E,D)$
6. $D:$



VE: Eliminate B, placing new factor f_{10} in first applicable bucket.

Ordering:
 C, F, A, B, E, D



1. $C: f_3(A, B, C), f_4(C, D), f_5(C, E)$

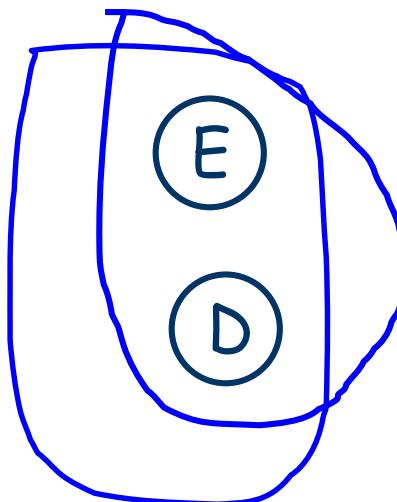
2. $F: f_6(E, D, F)$

3. $A: f_1(A), f_7(A, B, D, E)$

4. $B: f_2(B), f_9(B, D, E)$

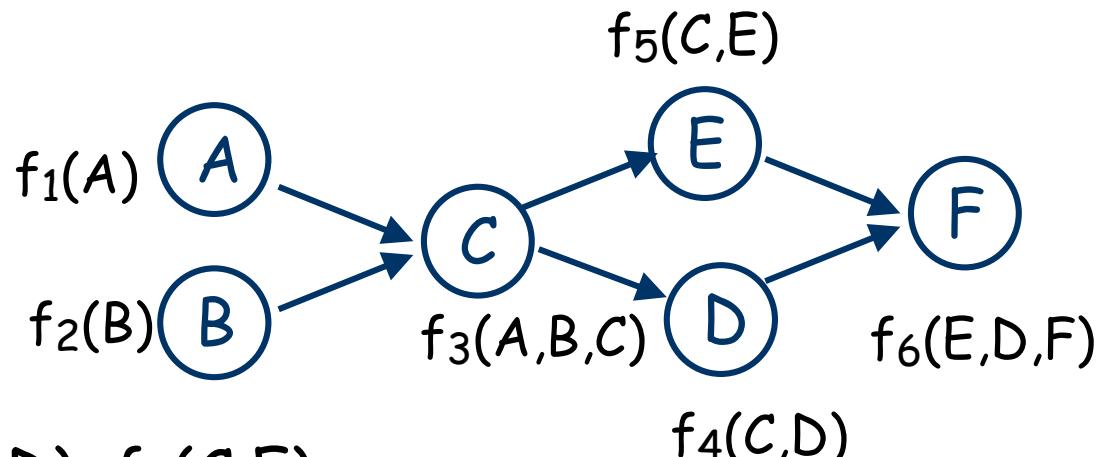
5. $E: f_8(E, D), \textcolor{blue}{f_{10}(D, E)}$

6. $D:$

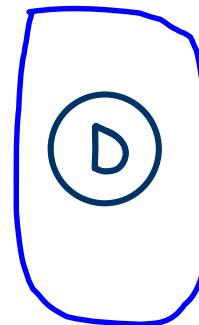


VE: Eliminate E, placing new factor f_{11} in first applicable bucket.

Ordering:
 C, F, A, B, E, D



1. ~~$C: f_3(A, B, C), f_4(C, D), f_5(C, E)$~~
2. ~~$F: f_6(E, D, F)$~~
3. ~~$A: f_1(A), f_7(A, B, D, E)$~~
4. ~~$B: f_2(B), f_9(B, D, E)$~~
5. ~~$E: f_8(E, D), f_{10}(D, E)$~~
6. $D: f_{11}(D)$



Elimination Width

- ▶ Given an ordering π of the variables and an initial hypergraph \mathcal{H} eliminating these variables yields a sequence of hypergraphs
$$\mathcal{H} = H_0, H_1, H_2, \dots, H_n$$
- ▶ Where H_n contains only one vertex (the query variable).
- ▶ The elimination width π is the maximum size (number of variables) of any hyperedge in any of the hypergraphs H_0, H_1, \dots, H_n .
- ▶ The elimination width of the previous example was 4 ($\{A, B, E, D\}$ in H_1 and H_2).

Elimination Width

- ▶ If the elimination width of an ordering π is k , then the complexity of VE using that ordering is $2^{O(k)}$
- ▶ Elimination width k means that at some stage in the elimination process a factor involving k variables was generated.
- ▶ That factor will require $2^{O(k)}$ space to store
 - ▶ space complexity of VE is $2^{O(k)}$
- ▶ And it will require $2^{O(k)}$ operations to process (either to compute in the first place, or when it is being processed to eliminate one of its variables).
 - ▶ Time complexity of VE is $2^{O(k)}$
- ▶ NOTE, that k is the elimination width of this particular ordering.

Tree Width

- ▶ Given a hypergraph \mathcal{H} with vertices $\{X_1, X_2, \dots, X_n\}$ the **tree width** of \mathcal{H} is the MINIMUM elimination width of any of the $n!$ different orderings of the X_i minus 1.
- ▶ Thus VE has best case complexity of $2^{O(\omega)}$ where ω is the TREE WIDTH of the initial Bayes Net.
- ▶ In the worst case the tree width can be equal to the number of variables.

Tree Width

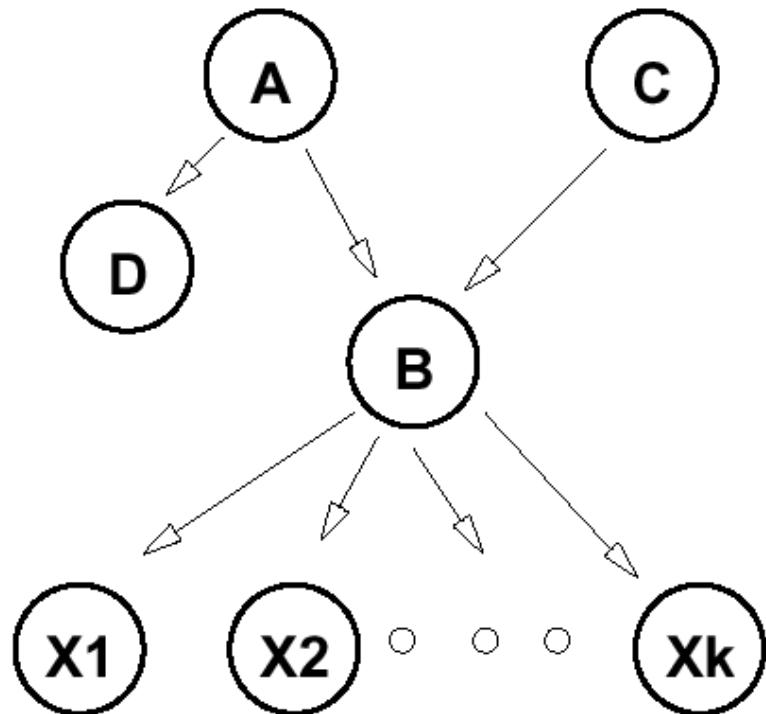
- ▶ Exponential in the tree width is the best that VE can do.
 - ▶ Finding an ordering that has elimination width equal to tree width is NP-Hard.
 - ▶ so in practice there is no point in trying to speed up VE by finding the best possible elimination ordering.
 - ▶ Heuristics are used to find orderings with good (low) elimination widths.
 - ▶ In practice, this can be very successful. Elimination widths can often be relatively small, 8-10 even when the network has 1000s of variables.
 - ▶ Thus VE can be much!! more efficient than simply summing the probability of all possible events (which is exponential in the number of variables).
 - ▶ Sometimes, however, the treewidth is equal to the number of variables.

Finding Good Orderings

- ▶ A *polytree* is a singly connected Bayes Net: in particular there is only one path between any two nodes.
- ▶ A node can have multiple parents, but we have no cycles.
- ▶ Good orderings are easy to find for polytrees
 - ▶ At each stage eliminate **a singly connected node**.
 - ▶ Because we have a polytree we are assured that a singly connected node will exist at each elimination stage.
 - ▶ The size of the factors in the tree never increase.

Elimination Ordering: Polytrees

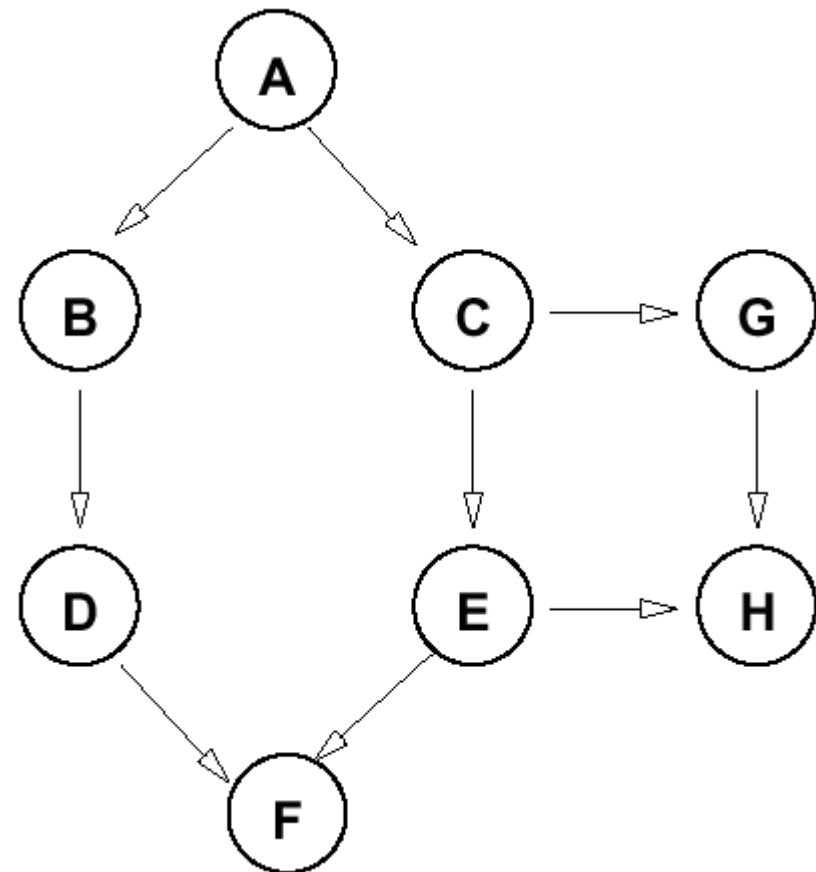
- ▶ Treewidth of a polytree is 1!
- ▶ Eliminating singly connected nodes allows VE to run in time linear in size of network
 - ▶ e.g., in this network, eliminate D, A, C, X₁,...,; or eliminate X₁,..., X_k, D, A C; or mix up...
 - ▶ result: no factor ever larger than original CPTs
 - ▶ eliminating B before these gives factors that include all of A,C, X₁,... X_k !!!



Effect of Different Orderings

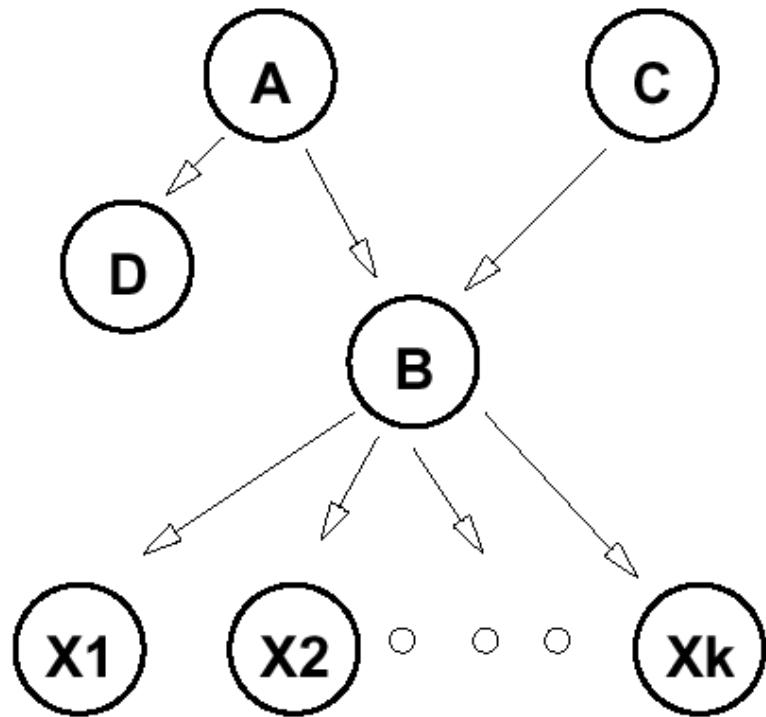
- ▶ Suppose query variable is D. Consider different orderings for this network (not a polytree!)

- ▶ A,F,H,G,B,C,E:
 - ▶ good
- ▶ E,C,A,B,G,H,F:
 - ▶ bad



Min Fill Heuristic

- ▶ A fairly effective heuristic is always eliminate next the variable that creates the smallest size factor.
- ▶ This is called the min-fill heuristic.
- ▶ B creates a factor of size $k+2$
- ▶ A creates a factor of size 2
- ▶ D creates a factor of size 1
- ▶ The heuristic always solves polytrees in linear time.



Relevance



- ▶ Certain variables have no impact on the query.
In network ABC, computing $\Pr(A)$ with no evidence requires elimination of B and C.
 - ▶ But when you sum out these vars, you compute a trivial factor (whose value are all ones); for example:
 - ▶ eliminating C: $f_4(B) = \sum_C f_3(B,C) = \sum_C \Pr(C | B)$
 - ▶ 1 for any value of B (e.g., $\Pr(c | b) + \Pr(\sim c | b) = 1$)
- ▶ No need to think about B or C for this query

Relevance

- ▶ Can restrict attention to *relevant* variables.
Given query q , evidence \mathbf{E} :
 - ▶ q itself is relevant
 - ▶ if any node \mathbf{z} is relevant, its parents are relevant
 - ▶ if $e \in \mathbf{E}$ is a descendent of a relevant node, then E is relevant
- ▶ We can restrict our attention to the *subnetwork comprising only relevant variables* when evaluating a query Q

Relevance: Examples

- ▶ Query: $P(F)$
 - ▶ relevant: F, C, B, A
- ▶ Query: $P(F | E)$
 - ▶ relevant: F, C, B, A
 - ▶ **also: E, hence D, G**
 - ▶ intuitively, we need to compute $P(C|E)$ to compute $P(F | E)$

- ▶ Query: $P(F | H)$
 - ▶ relevant F,C,A,B.

$$Pr(A)Pr(B)Pr(C | A,B)Pr(F | C) Pr(G)Pr(h | G)Pr(D | G,C)Pr(E | D)$$

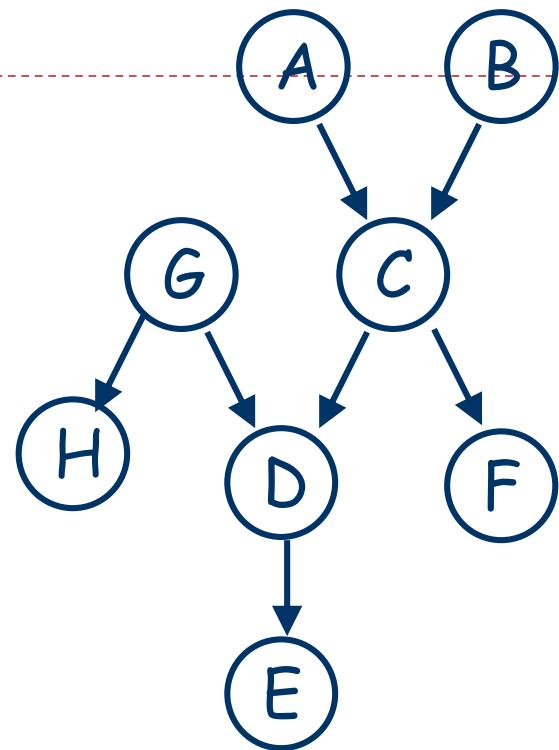
$$= \dots Pr(G)Pr(h | G)Pr(D | G,C) \sum_E Pr(E | D) = \text{a table of 1's}$$

$$= \dots Pr(G)Pr(h | G) \sum_D Pr(D | G,C) = \text{a table of 1's}$$

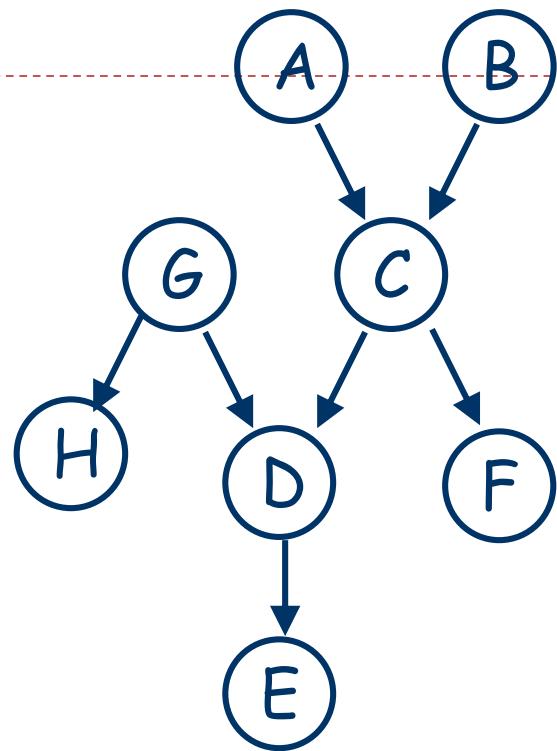
$$= [Pr(A)Pr(B)Pr(C | A,B)Pr(F | C)] [Pr(G)Pr(h | G)]$$

$[Pr(G)Pr(h | G)] \neq 1$ but irrelevant

once we normalize, multiplies each value of F equally



Relevance: Examples



▶ Query: $P(F | E, C)$

- ▶ algorithm says all vars except H are relevant; but really none except C, F (since C cuts off all influence of others)
- ▶ algorithm is overestimating relevant set

Independence in a Bayes Net

- ▶ Another piece of information we can obtain from a Bayes net is the “structure” of relationships in the domain.
- ▶ The structure of the BN means: every X_i is *conditionally independent of all of its nondescendants given its parents*:

$$\Pr(X_i | S \cup \text{Par}(X_i)) = \Pr(X_i | \text{Par}(X_i))$$

for any subset $S \subseteq \text{NonDescendents}(X_i)$

More generally...

- ▶ Many conditional independencies hold in a given BN.
- ▶ These independencies are useful in computation, explanation, etc.
- ▶ Some of these independencies can be detected using a graphical condition called **D-Separation**.

Approximate Inference in Bayes Nets

- ▶ Often the Bayes net is not solvable by Variable Elimination: under any ordering of the variables we end up with a factor that is too large to compute (or store).
- ▶ Since we are trying to compute a probability (which only predicts the likelihood of an event occurring) it is natural to consider approximating answer.

Sampling Techniques

- ▶ **Direct Sampling** from the **prior** distribution.
- ▶ Every Bayes net specifies the probability of every atomic event:
 - ▶ Each atomic event is a particular assignment of values to all of the variables in the Bayes nets.
 - ▶ Let V_1, \dots, V_n be the variables in the Bayes net.
 - ▶ Let d_1, \dots, d_n be values for these variables (d_i is the value variable V_i takes).
 - ▶ The Bayes net specifies that

$$\Pr(V_1 = d_1, V_2 = d_2, \dots, V_n = d_n) = \prod_{i=1}^n \Pr(V_i = d_i \mid \text{ParVals}(V_i))$$

where $\text{ParVals}(V_i)$ is the set of assignments $V_k = d_k$ for each $V_k \in \text{Par}(V_i)$

Sampling Techniques

- ▶ So we want to sample atomic events in such a ways that the probability we select event \mathbf{e} is equal to $\Pr(\mathbf{e})$
1. Select an unselected variable v_i such that all parents of v_i in the Bayes Net have already been selected.
 2. Let $[P_1, P_2, \dots, P_k]$ be the parents of v_i in the Bayes net. Let $[b_1, \dots, b_k]$ be the values that have already been selected for these parents ($P_i = b_i$).
 3. Set v_i to the value $d \in \text{Dom}[v_i]$ with probability

$$\Pr(v_i = d \mid P_1=b_1, P_2=b_2, \dots, P_k=b_k)$$

Sampling Techniques

- ▶ Note that the probabilities

$\Pr(V_i = d \mid P_1 = b_1, P_2 = b_2, \dots, P_k = b_k)$
are specified in V_i 's CPT in the Bayes net.

- ▶ Each variable is given a value by a separate random selection so the probability one obtains a particular atomic event **e** (a setting of all of the variables) via this algorithm is exactly **Pr(e)** as specified by the Bayes Net.

$$\Pr(e = [V_1 = d_1, V_2 = d_2, \dots, V_n = d_n]) = \prod_{i=1}^n \Pr(V_i = d_i \mid \text{ParVals}(V_i))$$

Sampling Techniques

- ▶ Say we want to evaluate $\Pr(V_1 = d_3)$
- ▶ We select **N** random samples of atomic events via this method
- ▶ Then we compute the proportion of these N events in which $V_1 = d_3$
- ▶ This proportion
$$(\text{Number of Events where } V_1 = d_3)/\mathbf{N}$$
is an estimate of $\Pr(V_1 = d_3)$.
- ▶ The estimate gets better as **N** gets larger, and by the law of large numbers as **N** approaches infinity the estimate converges (becomes closer and closer) to the exact $\Pr(V_1 = d_3)$

Sampling Techniques

- ▶ If we want to compute a conditional probability like $\Pr(V_1 = d_3 | V_4 = d_1)$, then we can
 - ▶ Discard all atomic events in which $V_4 \neq d_1$
 - ▶ This gives a new smaller set of N' sampled atomic events.
 - ▶ From those N' we compute the proportion in which $V_1 = d_3$
 - ▶ This proportion
$$(\text{Number of Events where } V_1 = d_3 \text{ from the remaining samples})/N'$$
is an estimate of $\Pr(V_1 = d_3 | V_4 = d_1)$
 - ▶ This is called **Rejection Sampling**

Sampling Techniques

- ▶ **Problem**, almost all samples might be rejected if $V_4 = d_1$ has very low probability.
- ▶ The accuracy of the estimate depends on the size of N' (the samples that remain after rejection).
- ▶ So if very few are left our estimate is not good.
- ▶ E.g., if $\Pr(V_4 = d_1) = 0.0000001$, then if we generate $1 / 0.0000001 = 10,000,000$ samples we expect to reject 9,999,999 of them. In that case our estimate of $\Pr(V_1 = d_3 | V_4 = d_1)$ will be 1 or 0! (Either our sole remaining sample has $V_1 = d_3$ or it doesn't).
- ▶ In most cases we want to compute **posterior** probabilities, i.e., probabilities conditioned on the **evidence**. So this is a major problem.

Sampling Techniques

- ▶ **Likelihood Weighting** tries to address this issue.
- ▶ Force all samples to be compatible with the conditioning event.
- ▶ Don't select a value for a variable whose value is specified in the evidence that we are conditioning on.
- ▶ Weigh each sample by its probability—some samples count more than others in computing the estimate.

Sampling Techniques

1. Set $w = 1$, let the evidence be a set of variables whose values are already given.
2. **while there are unselected variables**
 1. Select an unselected variable v_i such that all parents of v_i in the Bayes Net have already been selected.
 2. Let $[P_1, P_2, \dots, P_k]$ be the parents of v_i in the Bayes net. Let $[b_1, \dots, b_k]$ be the values that have already been selected for these parents ($P_i=b_i$).
 3. **If** v_i 's value is specified in the evidence and d is the value specified then
$$w = w * \Pr(v_i = d \mid P_1=b_1, P_2=b_2, \dots, P_k=b_k)$$
 4. **Else** set v_i to the value $d \in \text{Dom}[v_i]$ with probability
$$\Pr(v_i = d \mid P_1=b_1, P_2=b_2, \dots, P_k=b_k)$$

Sampling Techniques

- ▶ If we want to compute a conditional probability like $\Pr(V_1 = d_3 | V_4 = d_1)$, then we can
 - ▶ Generate a collection **N** of likelihood weighted samples using the evidence $V_4 = d_1$
 - ▶ Each sample (atomic event) **e** has a weight **w**.
 - ▶ We compute the sum of the weights of the samples in **N** in $V_1 = d_3$ and divide this by the sum of the weights of all samples in **N**.
 - ▶ This number
$$(\text{Sum of weights of samples in } \mathbf{N} \text{ where } V_1 = d_3) / (\text{sum of weights of samples in } \mathbf{N})$$
is an estimate of $\Pr(V_1 = d_3 | V_4 = d_1)$

Sampling Techniques

- ▶ **Problem**, many samples might have very low weight.
Some might even have zero weight.
 - ▶ Zero weight occurs when we have selected the parents of an evidence variable in such a way that
$$\Pr(V_i = d \mid P_1=b_1, P_2=b_2, \dots, P_k=b_k)$$
is zero (this is multiplied into the sample weight).
 - ▶ The accuracy of the estimate increases as the total weight of the samples increases, so if each sample has very low weight, we may need a very large number of weights.

Sampling Techniques

- ▶ Markov Chain Monte Carlo (MCMC) methods solve some of these problems.
- ▶ The book gives a description of Gibbs Sampling (a form of MCMC).

Make Change

- Starting with 0 cents, we want to reach some number of cents between [0, 5, 10, 15, 20, ..., 500] using the least number of coins.
- We can add 5, 10, 25, 100, or 200.
- Solve this problem using search?

Make Change

- States:
- Actions:

Make Change

- States: Integers between 0 and 500 that are divisible by 5.
- Actions: Add 5, 10, 25, 100, or 200.

Example

- Initial state 0
 - Goal state (365)
-
- BFS
 - DFS
 - UCS?
 - A*?

Logistics World

- Set of Cities
- For each City a set of locations in that city.
- Some locations are Airports.
- Set of Trucks, each truck is in some city.
- Set of Airplanes
- Trucks can move between any location in the same city.
- Airplanes can move between any two airport.
- Set of packages each in some city at some location.
- Packages can be loaded into a truck or airplane if that vehicle is at the same location at the package.
- If a package is in a vehicle it is moved when the vehicle is moved.

Logistics World

- Aim is to pickup a bunch of packages and deliver them to some goal locations.

Logistics World

- State Space
- Actions

Logistics World

- State Space: Location for every vehicle. And for every package either a location where it is, or a vehicle that it is in.
- Actions:
 1. If a truck is at location locA, we can move it to location locB if locB is in the same city as locA
 2. If a truck is at location locA, we can move it to location locB if locB is an airport.
 3. If package P1 is at location locA, and a vehicle V1 is also at location locA, we can load P1 into V1. P1 is now in V1.
 4. If a package P1 is in a vehicle V1 and V1 is at location locA, we can unload P1. P1 now is at location locA
 5. Costs vary—cost of moving a vehicle depends on distance traveled. Loading and unloading package has fixed (low cost).

Logistics World

- Initial state: a set of packages and vehicles and their locations.
- Goal state: a set of destination locations for some of the packages.
- BFS?
- UCS?
- A*?

Vacuum World

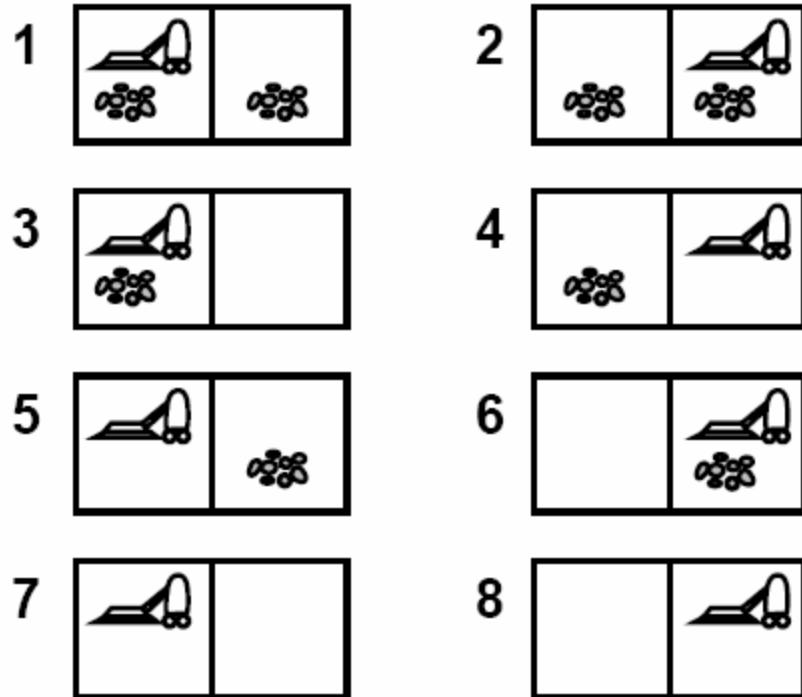
- In the previous examples, a state in the search space corresponded to a unique state of the world (modulo details we have abstracted away).
 - However, states need not map directly to world configurations. Instead, a state could map to **knowledge states**.
-
- If you know the exact state of the world your knowledge state is a single unique state.
 - If you don't know some things, then your knowledge state is a **set** of world states.

Vacuum World

- A knowledge state will include every world state that might be possible.

Example 3. Vacuum World

- We have a vacuum cleaner and two rooms.
- Each room may or may not be dirty.
- The vacuum cleaner can move **left** or **right** (*the action has no effect if there is no room to the right/left*).
- The vacuum cleaner can **suck**; this cleans the room (*even if the room was already clean*).

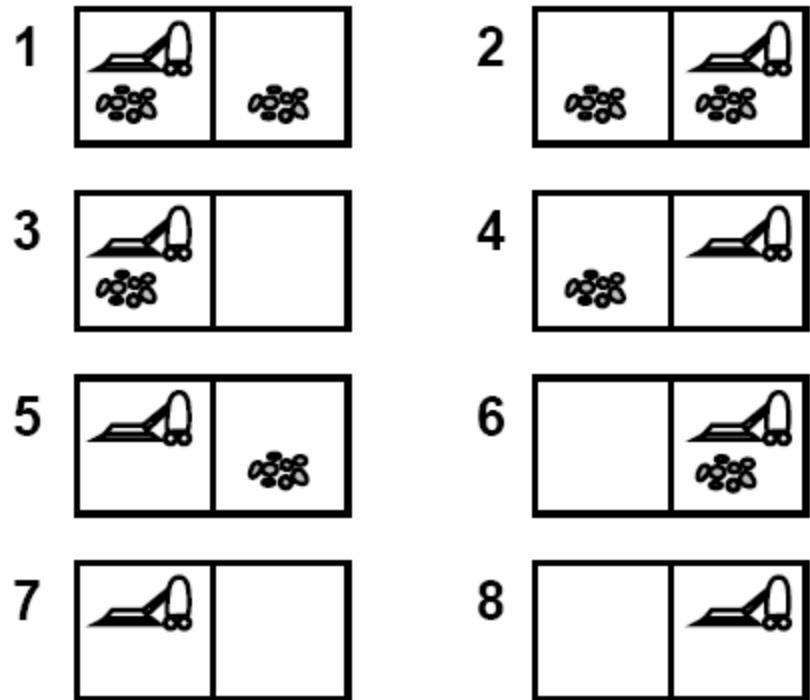


Physical states

Example 3. Vacuum World

Knowledge-level State Space

- Each state can consist of a set of possible world states. The agent knows that it is in one of these states, but doesn't know which.

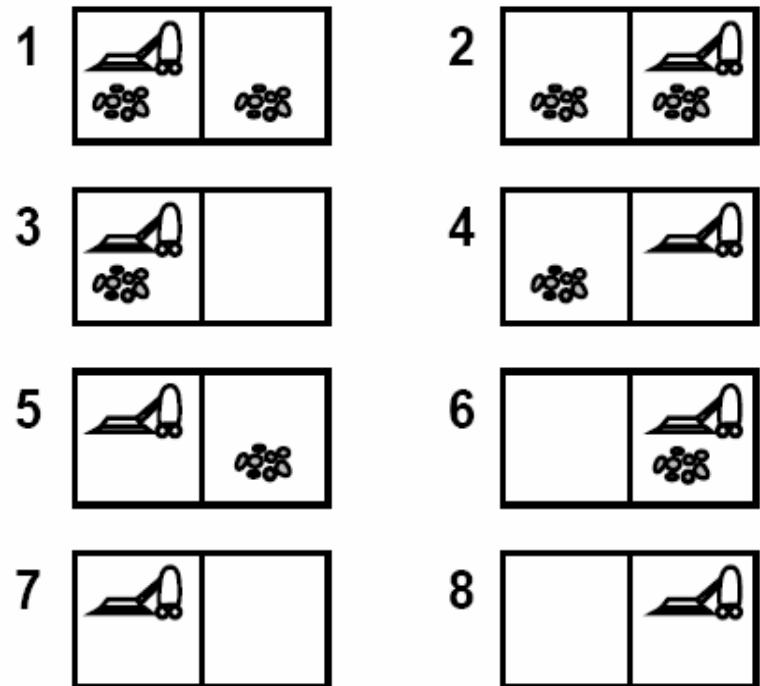


Goal is to have all rooms clean.

Example 3. Vacuum World

Knowledge-level State Space

- Complete knowledge of the world: agent knows exactly which physical state it is in. Then the states in the agent's state space consist of single physical states.
- Start in {5}:
<right, suck>

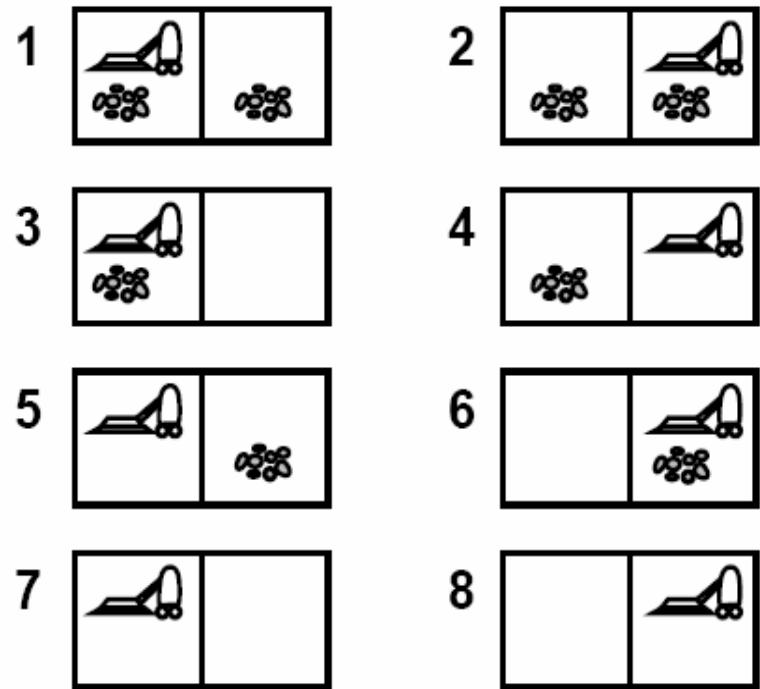


Goal is to have all rooms clean.

Example 3. Vacuum World

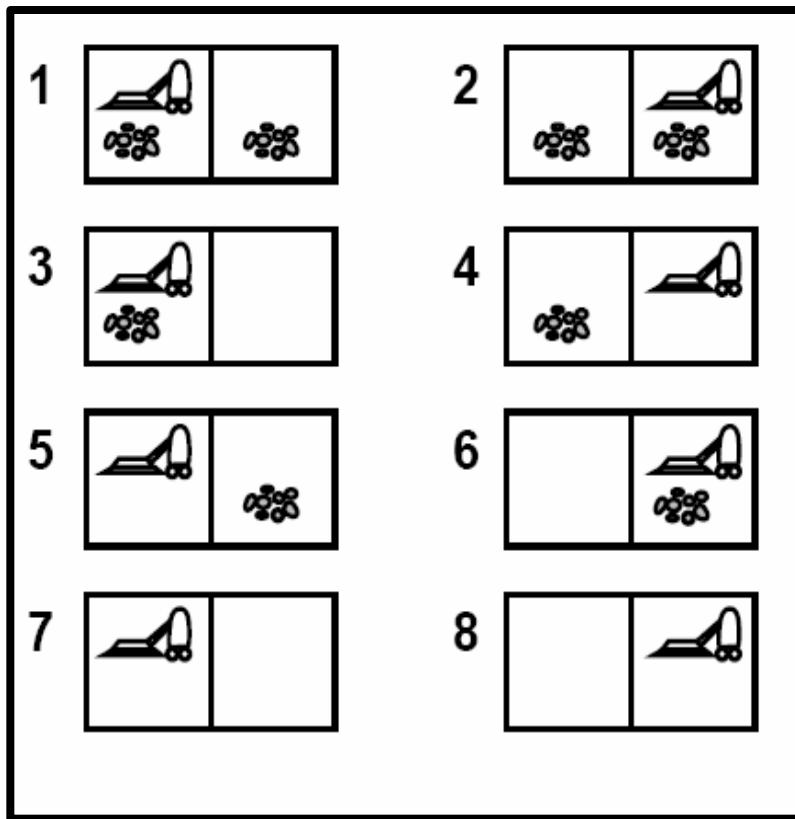
Knowledge-level State Space

- No knowledge of the world:
Agent's states consist of *sets of world states*.
- E.g. starting in $\{1,2,3,4,5,6,7,8\}$, the agent doesn't have any knowledge of where it is.
- Nevertheless, the action sequence
<right, suck, left, suck>
achieves the goal.

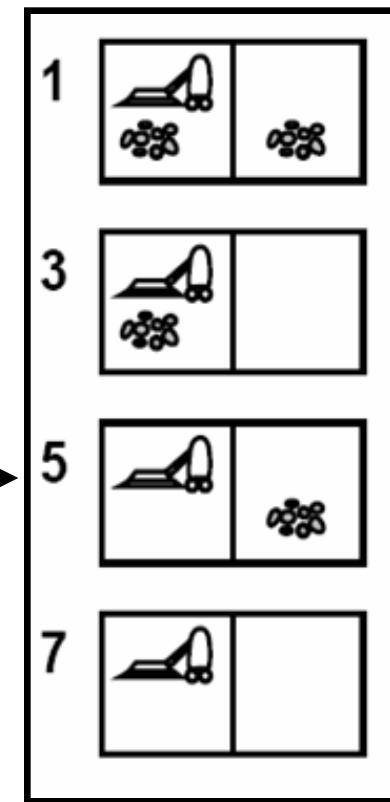


Goal is to have all rooms clean.

Example 3. Vacuum World



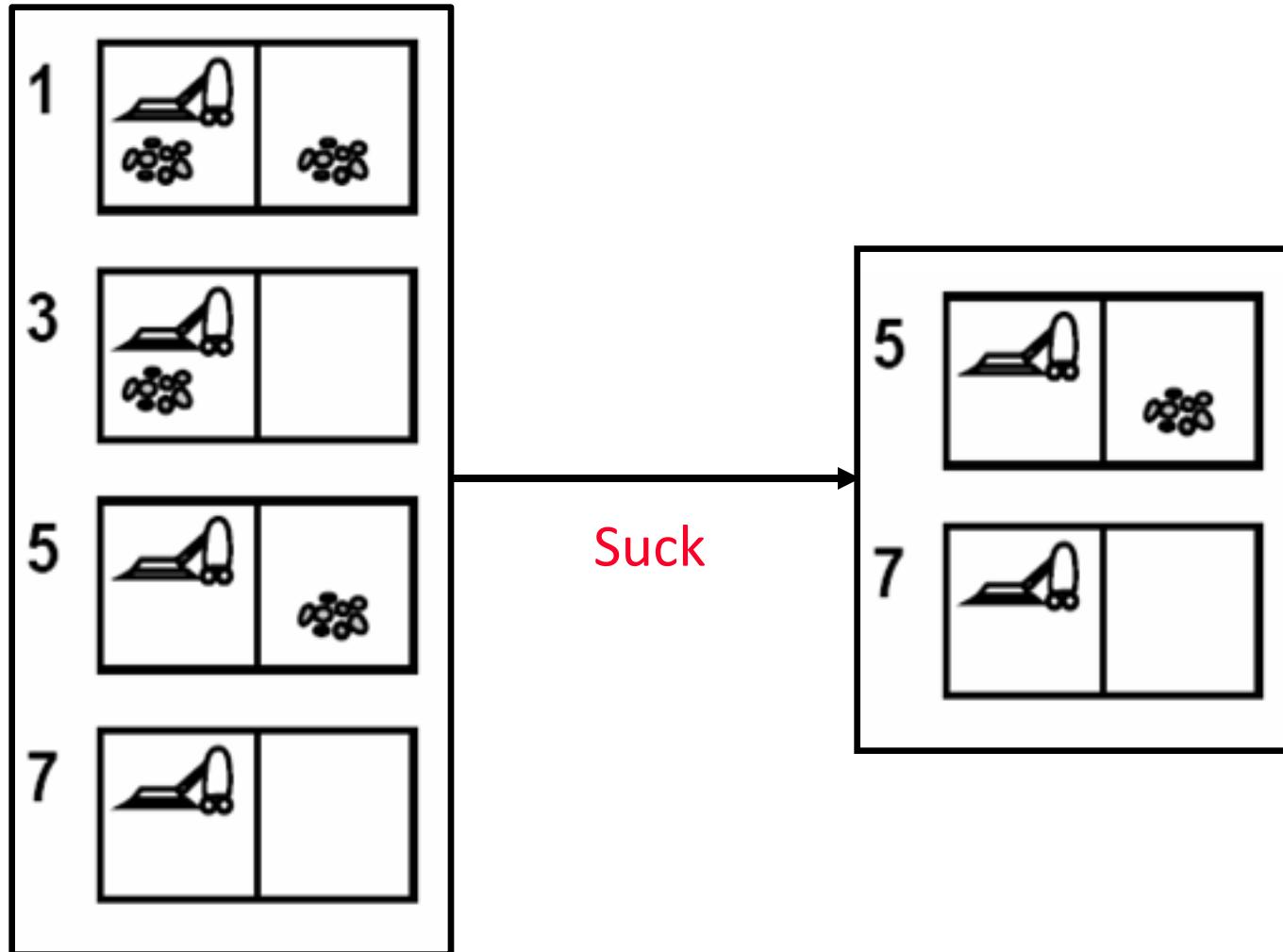
Left



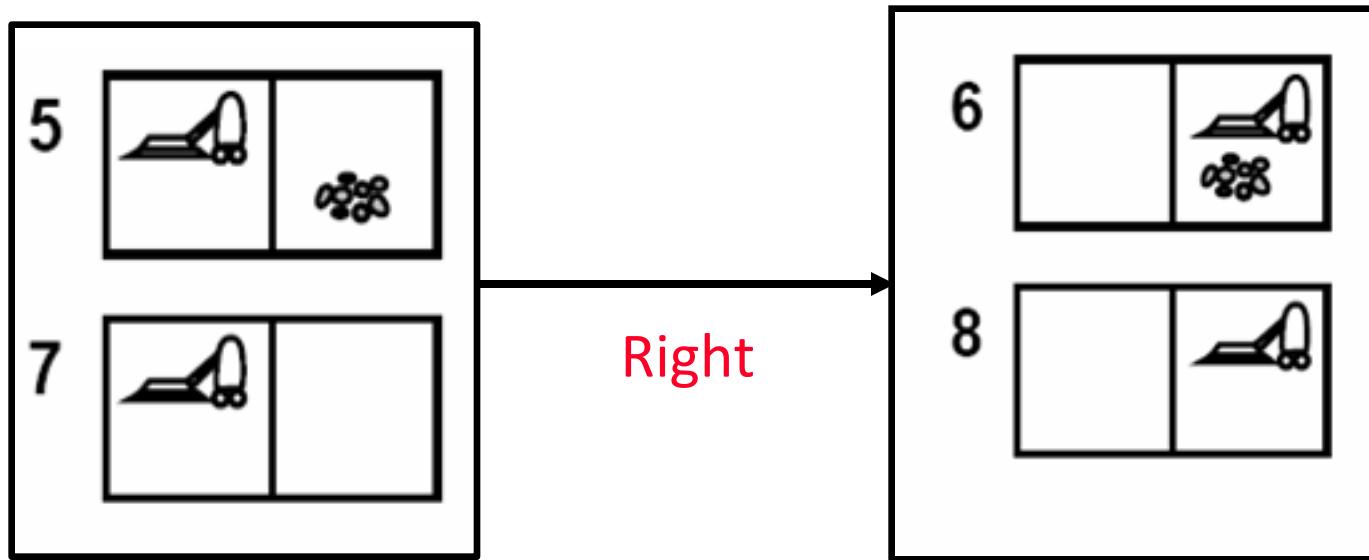
Initial state.

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

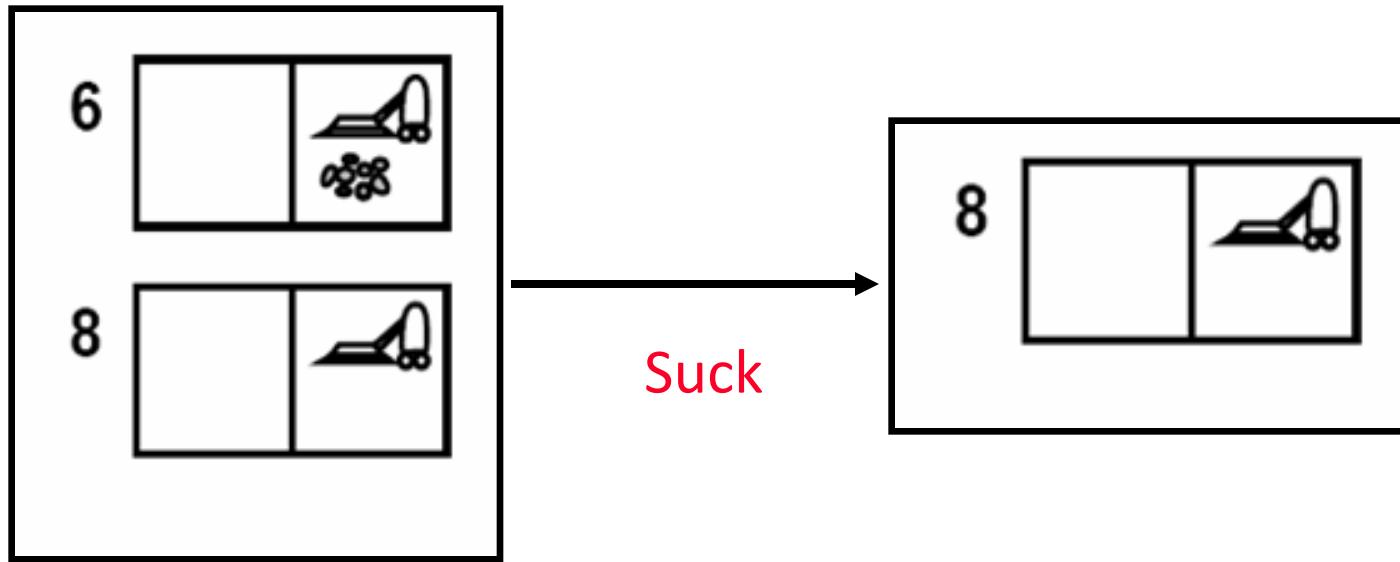
Example 3. Vacuum World



Example 3. Vacuum World



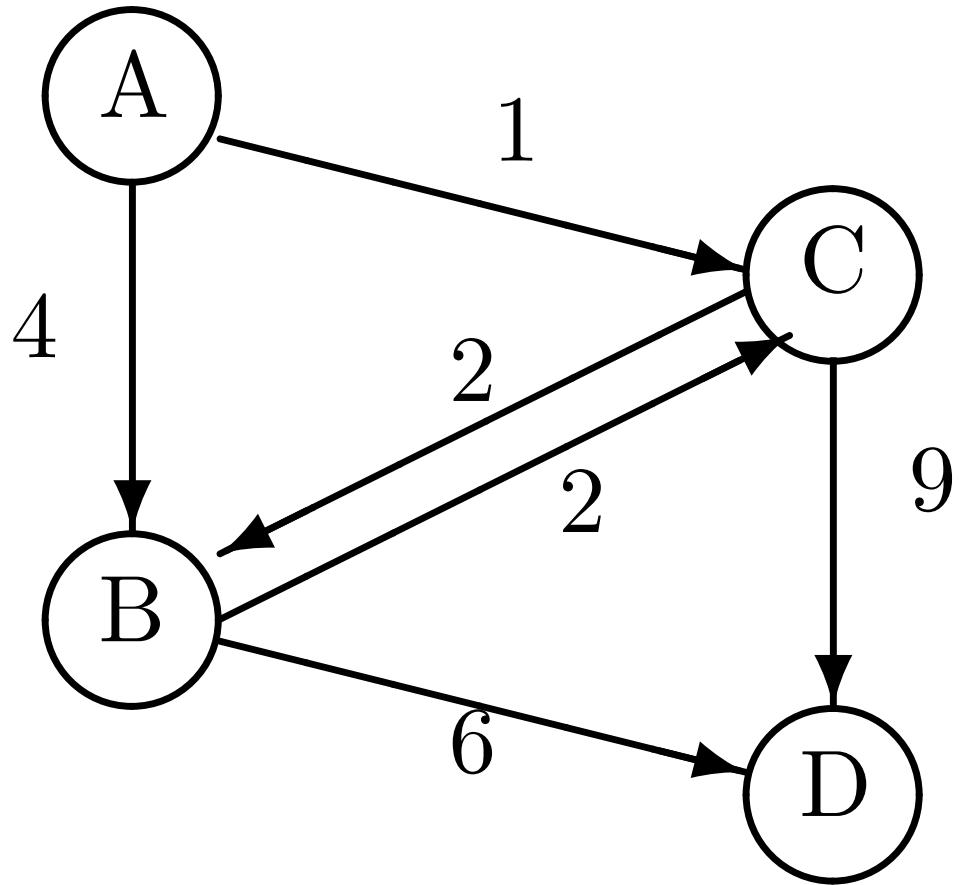
Example 3. Vacuum World



More complex situations

- The agent might be able to perform some sensing actions. These actions change the agent's mental state, not the world configuration.
- With sensing can search for a **contingent** solution: a solution that is contingent on the outcome of the sensing actions
 - <**right, if dirt then suck**>
- Now the issue of interleaving execution and search comes into play.

A* Example



$$h(A) = 8$$

$$h(B) = 3$$

$$h(C) = 7$$

$$h(D) = 0$$

START = A

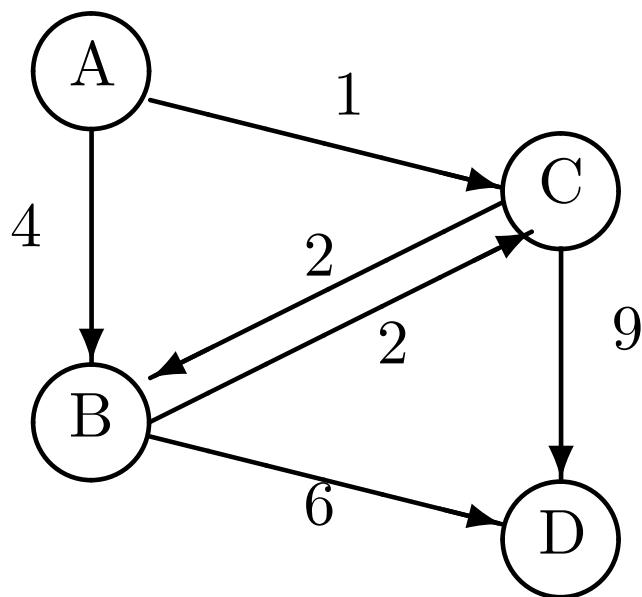
GOAL = D

A*

- Successive states of OPEN:

Items on OPEN are $\langle \text{Node}, g\text{-val} + h\text{-val} = f\text{-val} \rangle$

Where Node = $[s_0, s_2 \dots]$ a sequence of states representing the path.



$$h(A) = 8$$

$$h(B) = 3$$

$$h(C) = 7$$

$$h(D) = 0$$

A*

{<[A],0+8=8>}

{<[A,C], 1+7=8>, <[A,B] = 4+3 = 7>}

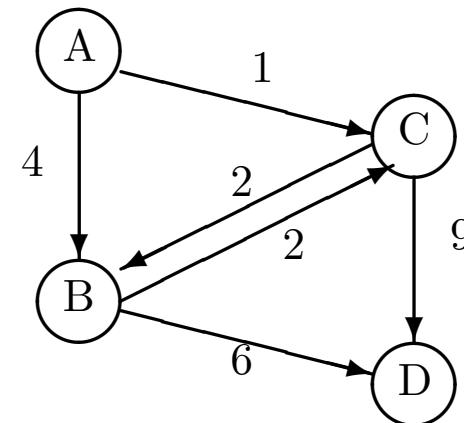
{<[A,C], 1+7=8>, <[A,B,C] = 6+7=13>, <[A,B,D] = 10+0=10>}

{<[A,C,B], 3+3=6>, <[A,C,D], 10+0=10>, <[A,B,C] = 6+7=13>, <[A,B,D] = 10+0=10>}

{<[A,C,B,C] = 5+7=12>, <[A,C,B,D] = 9+0= 9>, <[A,C,D], 10+0=10>, <[A,B,C] = 6+7=13>, <[A,B,D] = 10+0=10>}

Node selected for expansion reaches a goal.

Green = next node expanded



$$\begin{aligned} h(A) &= 8 \\ h(B) &= 3 \\ h(C) &= 7 \\ h(D) &= 0 \end{aligned}$$

A* Path Checking

{<[A],0+8=8>}

{<[A,C], 1+7=8, <[A,B] = 4+3 = 7>}

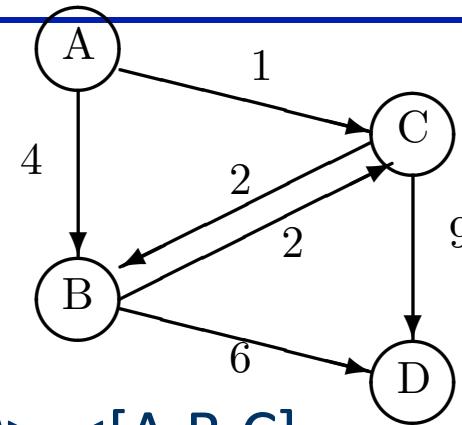
{<[A,C], 1+7=8, <[A,B,C] = 6+7=13>,
<[A,B,D] = 10+0=10>}

{<[A,C,B], 3+3=6>, <[A,C,D], 10+0=10>, <[A,B,C] =
6+7=13>,

<[A,B,D] = 10+0=10>}

{[A,C,B,C] = 5+7=12, [A,C,B,D] = 9+0= 9, <[A,C,D],
10+0=10>, <[A,B,C] = 6+7=13>,

<[A,B,D] = 10+0=10>}



$$h(A) = 8$$

$$h(B) = 3$$

$$h(C) = 7$$

$$h(D) = 0$$

Red pruned by cycle checking.

A* Full Cycle Checking

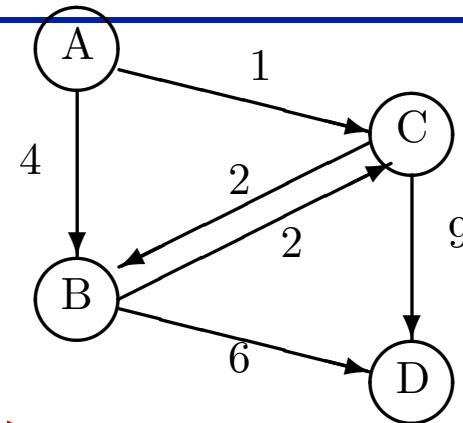
{<[A], 0+8=8>}

{<[A,C], 1+7=8, <[A,B] = 4+3 = 7>}

{<[A,C], 1+7=8, <[A,B,C] = 6+7=13>, <[A,B,D] = 10+0=10>}

{<[A,C,B], 3+3=6>, <[A,C,D], 10+0=10>, <[A,B,C] = 6+7=13>, <[A,B,D] = 10+0=10>}

{<[A,C,B,C] = 5+7=12, [A,C,B,D] = 9+0= 9, <[A,C,D], 10+0=10>, <[A,B,C] = 6+7=13>, <[A,B,D] = 10+0=10>}



$$h(A) = 8$$

$$h(B) = 3$$

$$h(C) = 7$$

$$h(D) = 0$$

A* Short Questions

- If $h(n)$ is admissible and s is the start node how is $h(s)$ related to the cost of the solution eventually found by A*?
- If there is a solution, then during its operation A* always has at least one prefix of an optimal path to a goal on OPEN.
- What happens when $h(n) = h^*(n)$
 - a. A* only expands nodes that lie on an optimal path to the goal
 - b. Does this mean that A* will find a solution in time linear in the length of an optimal solution?

GAC Example

- (a) $\text{Dom}[X] = \{1, 2, 3, 4\}$
- (b) $\text{Dom}[Y] = \{1, 2, 3, 4\}$
- (c) $\text{Dom}[Z] = \{1, 2, 3, 4\}$
- (d) $\text{Dom}[W] = \{1, 2, 3, 4, 5\}$

And 3 constraints:

- (a) $C_1(X, Y, Z)$ which is satisfied only when $X = Y + Z$
- (b) $C_2(X, W)$ which is satisfied only when $W > X$
- (c) $C_3(X, Y, Z, W)$ which is satisfied only when $W = X + Z + Y$

Enforce GAC on these constraints, and give the resultant GAC consistent variable domains.

GAC Example

- (a) $\text{Dom}[X] = \{1, 2, 3, 4\}$
- (b) $\text{Dom}[Y] = \{1, 2, 3, 4\}$
- (c) $\text{Dom}[Z] = \{1, 2, 3, 4\}$
- (d) $\text{Dom}[W] = \{1, 2, 3, 4, 5\}$

And 3 constraints:

- (a) $C_1(X, Y, Z)$ which is satisfied only when $X = Y + Z$
- (b) $C_2(X, W)$ which is satisfied only when $W > X$
- (c) $C_3(X, Y, Z, W)$ which is satisfied only when $W = X + Z + Y$

Enforce GAC on these constraints, and give the resultant GAC consistent variable domains.

■ All constraints put on GAC queue.

■ Process C_3 first.

$X = 1$ ($X=1, Y=1, Z=1, W=3$)

$X = 2$ ($X=2, Y=1, Z=1, W=4$)

$X = 3$ ($X=3, Y=1, Z=1, W=5$)

$X = 4$ – Inconsistent.

$\text{Dom}(X) = \{1, 2, 3\}$

similarly

$\text{Dom}(Y) = \{1, 2, 3\}$

$\text{Dom}(Z) = \{1, 2, 3\}$

$W = 1$ – inconsistent

$W = 2$ – inconsistent

$W = 3$ – same support as $X=1$

$W = 4$ – same support as $X = 2$

$W = 5$ – same support as $X = 3$

$\text{Dom}(W) = \{3, 4, 5\}$

All domains pruned, but all other constraints already on GAC queue

GAC Example

- (a) $\text{Dom}[X] = \{1, 2, 3, 4\}$
- (b) $\text{Dom}[Y] = \{1, 2, 3, 4\}$
- (c) $\text{Dom}[Z] = \{1, 2, 3, 4\}$
- (d) $\text{Dom}[W] = \{1, 2, 3, 4, 5\}$

And 3 constraints:

- (a) $C_1(X, Y, Z)$ which is satisfied only when $X = Y + Z$
- (b) $C_2(X, W)$ which is satisfied only when $W > X$
- (c) $C_3(X, Y, Z, W)$ which is satisfied only when $W = X + Z + Y$

Enforce GAC on these constraints, and give the resultant GAC consistent variable domains.

■ Process C_2 next
Currently

$$\begin{aligned}\text{Dom}(X) &= \{1, 2, 3\} \\ \text{Dom}(W) &= \{3, 4, 5\}\end{aligned}$$

$$\begin{aligned}X = 1 \quad &(X=1, W=3) \\ X = 2 \quad &(X=2, W=3) \\ X = 3 \quad &(X=3, W=4)\end{aligned}$$

$$W=5 \quad (X=1, W=5)$$

No domains pruned. Nothing added to GAC Queue

$W=3, W=4$ found supports already

GAC Example

- (a) $\text{Dom}[X] = \{1, 2, 3, 4\}$
- (b) $\text{Dom}[Y] = \{1, 2, 3, 4\}$
- (c) $\text{Dom}[Z] = \{1, 2, 3, 4\}$
- (d) $\text{Dom}[W] = \{1, 2, 3, 4, 5\}$

And 3 constraints:

- (a) $C_1(X, Y, Z)$ which is satisfied only when $X = Y + Z$
- (b) $C_2(X, W)$ which is satisfied only when $W > X$
- (c) $C_3(X, Y, Z, W)$ which is satisfied only when $W = X + Z + Y$

Enforce GAC on these constraints, and give the resultant GAC consistent variable domains.

■ Process C_1 next

At this stage

$$\begin{aligned}\text{Dom}(X) &= \text{Dom}(Y) = \text{Dom}(Z) \\ &= \{1, 2, 3\}\end{aligned}$$

$X = 1$ – inconsistent

$X = 2$ – ($X=2, Y=1, Z=1$)

$X = 3$ – ($X=3, Y=1, Z=2$)

$Z = 1$ – same support as $X=2$

$Z = 2$ – same support as $X=3$

$Z = 3$ – inconsistent

Updated domains

$X = \{2, 3\}$

$Y = \{1, 2\}$

$Z = \{1, 2\}$

Put C_2 and C_3 back onto GAC queue

$Y = 1$ – same support as $X=2$

$Y = 2$ – ($X=3, Y=2, Z=1$)

$Y = 3$ – inconsistent

GAC Example

- (a) $\text{Dom}[X] = \{1, 2, 3, 4\}$
- (b) $\text{Dom}[Y] = \{1, 2, 3, 4\}$
- (c) $\text{Dom}[Z] = \{1, 2, 3, 4\}$
- (d) $\text{Dom}[W] = \{1, 2, 3, 4, 5\}$

And 3 constraints:

- (a) $C_1(X, Y, Z)$ which is satisfied only when $X = Y + Z$
- (b) $C_2(X, W)$ which is satisfied only when $W > X$
- (c) $C_3(X, Y, Z, W)$ which is satisfied only when $W = X + Z + Y$

Enforce GAC on these constraints, and give the resultant GAC consistent variable domains.

■ Process C_3 next current domains:

$$\text{Dom}(X) = \{2, 3\}$$

$$\text{Dom}(Y) = \{1, 2\}$$

$$\text{Dom}(Z) = \{1, 2\}$$

$$\text{Dom}(W) = \{3, 4, 5\}$$

$$X = 2 - \{X=2, W=4, Y=1, Z=1\}$$

$$X = 3 - \{X=3, W=5, Y=1, Z=1\}$$

$$Y = 1 - \text{found support}$$

$$Y = 2 - \{X=2, W=5, Y=2, Z=1\}$$

$$Z = 1 - \text{found support}$$

$$Z = 2 - \{X=2, W=5, Y=1, Z=2\}$$

$W = 3$ inconsistent

$W = 4$ – found support

$W = 5$ – found support

Pruned domains

$$W = \{4, 5\}$$

C_2 already on GAC queue

GAC Example

- (a) $\text{Dom}[X] = \{1, 2, 3, 4\}$
- (b) $\text{Dom}[Y] = \{1, 2, 3, 4\}$
- (c) $\text{Dom}[Z] = \{1, 2, 3, 4\}$
- (d) $\text{Dom}[W] = \{1, 2, 3, 4, 5\}$

And 3 constraints:

- (a) $C_1(X, Y, Z)$ which is satisfied only when $X = Y + Z$
- (b) $C_2(X, W)$ which is satisfied only when $W > X$
- (c) $C_3(X, Y, Z, W)$ which is satisfied only when $W = X + Z + Y$

Enforce GAC on these constraints, and give the resultant GAC consistent variable domains.

- Process C_2 next current domains:
 - No Domains pruned.
 - Nothing added to queue
- $\text{Dom}(X) = \{2, 3\}$
- $\text{Dom}(W) = \{4, 5\}$
- $X = 2 - \{X=2, W=4\}$
- $X = 3 - \{X=3, W=4\}$
- $W = 4 - \text{found support}$
- $W = 5 - \{X=3, W=5\}$
- Queue Empty
- GAC finished.
- GAC domains:
 - $X = \{2, 3\}$
 - $Z = \{1, 2\}$
 - $Y = \{1, 2\}$
 - $W = \{4, 5\}$

GAC Example

- (a) $\text{Dom}[X] = \{1, 2, 3, 4\}$
- (b) $\text{Dom}[Y] = \{1, 2, 3, 4\}$
- (c) $\text{Dom}[Z] = \{1, 2, 3, 4\}$
- (d) $\text{Dom}[W] = \{1, 2, 3, 4, 5\}$

And 3 constraints:

- (a) $C_1(X, Y, Z)$ which is satisfied only when $X = Y + Z$
- (b) $C_2(X, W)$ which is satisfied only when $W > X$
- (c) $C_3(X, Y, Z, W)$ which is satisfied only when $W = X + Z + Y$

Enforce GAC on these constraints, and give the resultant GAC consistent variable domains.

- Note GAC enforce does not find a solution

To find a solution we must use do search while enforcing GAC.

- Branch on X.

$$X = 2$$

$$\text{GAC}(C_1) \rightarrow Y = 1, Z = 1$$

$$\text{GAC}(C_2) \rightarrow \text{no changes}$$

$$\text{GAC}(C_3) \rightarrow W = 4$$

This is a solution.

- Branch on $X = 3$

$$\text{GAC}(C_1) \rightarrow \text{no changes}$$

$$\text{GAC}(C_2) \rightarrow \text{no changes}$$

$$\text{GAC}(C_3) \rightarrow \text{Prune } W=4$$

Prune $Y = 2$

Prune $Z = 2$

Current Domains

$$X=\{3\}, Y=\{1\}, Z=\{1\}, W=\{5\}$$

$$\text{GAC}(C_1) \rightarrow \text{Prune } Y=\{1\} \text{ DWO}$$

NOTE No solution with $X=3$ but $X=3$ not pruned by GAC enforce.

Example

■ C1(V1,V2,V3)

V1	V2	V3
A	B	C
B	A	C
A	A	B

■ C2(V1,V3,V4,V5)

V1	V3	V4	V5
A	A	A	A
A	B	C	B
B	C	B	B
C	A	B	C
C	B	A	B

■ C3(V2,V3,V5)

V2	V3	V5
A	A	A
A	B	C
B	C	B
C	A	B
C	B	A

■ Dom[V1]...Dom[V5] = {a, b, c}

Example

- C1(V1,V2,V3)

V1	V2	V3
A	B	C
B	A	C
A	A	B

- C2(V1,V3,V4,V5)

V1	V3	V4	V5
A	A	A	A
A	B	C	B
B	C	B	B
C	A	B	C
C	B	A	B

- C3(V2,V3,V5)

V2	V3	V5
A	A	A
A	B	C
B	C	B
C	A	B
C	B	A

- V1=C: no support
- V2=C: no support
- V3=A: no support

- V1={a,b}
- V2={a,b}
- V3={b,c}

Example

- C1(V1,V2,V3)

V1	V2	V3
A	B	C
B	A	C
A	A	B

- C2(V1,V3,V4,V5)

V1	V3	V4	V5
A	A	A	A
A	B	C	B
B	C	B	B
C	A	B	C
C	B	A	B

- C3(V2,V3,V5)

V2	V3	V5
A	A	A
A	B	C
B	C	B
C	A	B
C	B	A

- V1=C: no support
- V2=C: no support
- V3=A: no support

- V1={a,b}
- V2={a,b}
- V3={b,c}

Example

■ C1(V1,V2,V3)

V1	V2	V3
A	B	C
B	A	C
A	A	B

■ C2(V1,V3,V4,V5)

V1	V3	V4	V5
A	A	A	A
A	B	C	B
B	C	B	B
C	A	B	C
C	B	A	B

■ C3(V2,V3,V5)

V2	V3	V5
A	A	A
A	B	C
B	C	B
C	A	B
C	B	A

- V1=C: no support
- V2=C: no support
- V3=A: no support

- V1={a,b}
- V2={a,b}
- V3={b,c}

- V4=A: no support
- V5=A: no support
- V5=C: no support

- V4={C,B}
- V5={B}

Example

- C1(V1,V2,V3)

V1	V2	V3
A	B	C
B	A	C
A	A	B

- V1=C: no support
- V2=C: no support
- V3=A: no support

- V1={a,b}
- V2={a,b}
- V3={b,c}

- C2(V1,V3,V4,V5)

V1	V3	V4	V5
A	A	A	A
A	B	C	B
B	C	B	B
C	A	B	C
C	B	A	B

- V4=A: no support
- V5=A: no support
- V5=C: no support

- V4={C,B}
- V5={B}

- C2(V2,V3,V5)

V2	V3	V5
A	A	A
A	B	C
B	C	B
C	A	B
C	B	A

Example

- C1(V1,V2,V3)

V1	V2	V3
A	B	C
B	A	C
A	A	B

- V1=C: no support
- V2=C: no support
- V3=A: no support

- V1={a,b}
- V2={a,b}
- V3={b,c}

- C2(V1,V3,V4,V5)

V1	V3	V4	V5
A	A	A	A
A	B	C	B
B	C	B	B
C	A	B	C
C	B	A	B

- V4=A: no support
- V5=A: no support
- V5=C: no support

- V4={C,B}
- V5={B}

- C2(V2,V3,V5)

V2	V3	V5
A	A	A
A	B	C
B	C	B
C	A	B
C	B	A

- V2=A: no support
- V3=B: no support

- V2={B}
- V3={C}

Example

- C1(V1,V2,V3)

V1	V2	V3
A	B	C
B	A	C
A	A	B

- C2(V1,V3,V4,V5)

V1	V3	V4	V5
A	A	A	A
A	B	C	B
B	C	B	B
C	A	B	C
C	B	A	B

- C2(V2,V3,V5)

V2	V3	V5
A	A	A
A	B	C
B	C	B
C	A	B
C	B	A

- V1=B has no support
- V1={A}

- V4={C,B}
- V5={B}

- V2={B}
- V3={C}

Example

- C1(V1,V2,V3)

V1	V2	V3
A	B	C
B	A	C
A	A	B

- C2(V1,V3,V4,V5)

V1	V3	V4	V5
A	A	A	A
A	B	C	B
B	C	B	B
C	A	B	C
C	B	A	B

- C2(V2,V3,V5)

V2	V3	V5
A	A	A
A	B	C
B	C	B
C	A	B
C	B	A

- V1=B has no support
- V1={A}

- V4=B has no support
- V4={B}
- V5={B}
- V3=C has no support

- V3={} DWO

- V2={B}
- V3={C}

Tutorial Examples Logic

March 13, 2015

English To Logic

1. Nobody likes taxes.
2. Some people like anchovies.
3. Emma is a Doberman pincher and a good dog.

English To Logic

1. Nobody likes taxes.
 - ▶ $\neg \exists X. likesTaxes(X)$
 - ▶ $\neg \exists X, Y. person(X) \wedge tax(Y) \wedge likes(X, Y)$
 - ▶ $\forall X. person(X) \rightarrow \neg \exists Y. tax(Y) \wedge likex(X, Y)$
2. Some people like anchovies.
3. Emma is a Doberman pincher and a good dog.

English To Logic

1. Nobody likes taxes.

- ▶ $\neg \exists X. likes(Taxes(X))$

Works if every object in the domain is a person.

- ▶ $\neg \exists X, Y. person(X) \wedge tax(Y) \wedge likes(X, Y)$

Works if we want talk about different kinds of taxes or different things that are liked.

- ▶ $\forall X. person(X) \rightarrow (\neg \exists Y. tax(Y) \wedge likes(X, Y))$

Equivalent. $\neg \exists X, Y. person(X) \wedge tax(Y) \wedge likes(X, Y) \equiv$

$\forall X, Y. \neg person(X) \vee \neg tax(Y) \vee \neg likes(X, Y) \equiv$

$\forall X. person(X) \rightarrow (\forall Y. \neg tax(Y) \vee \neg likes(X, Y)) \equiv$

$\forall X. person(X) \rightarrow (\neg \exists Y. tax(Y) \wedge likes(X, Y)) \equiv$

2. Some people like anchovies.

3. Emma is a Doberman pincher and a good dog.

English To Logic

1. Nobody likes taxes.
2. Some people like anchovies.
 - ▶ $\exists X.\text{person}(X) \wedge \text{likes}(X, \text{anchovies})$.
 - ▶ $\exists X, Y.\text{person}(X) \wedge \text{anchovy}(Y) \wedge \text{likes}(X, Y)$.
3. Emma is a Doberman pincher and a good dog.

English To Logic

1. Nobody likes taxes.
2. Some people like anchovies.
3. Emma is a Doberman pincher and a good dog.
 - ▶ $doberman(Emma) \wedge good(Emma)$.

Models

Consider a first-order language \mathcal{L} containing the following basic symbols:

- ▶ Constants, A, B, C, D .
- ▶ The binary predicate R .
- ▶ The unary predicates P and Q .

Models

Let \mathcal{M} be a model for \mathcal{L} , with domain $D = \{a, b, c, d\}$, and interpretation function σ :

1. $A^\sigma = a, B^\sigma = b, C^\sigma = c, D^\sigma = d.$
2. $R^\sigma = \{(b, a), (c, d)\}.$
3. $P^\sigma = \{b, c\}.$
4. $Q^\sigma = \{a, d\}.$

Which of the following formulas are satisfied by \mathcal{M} .

1. $R(C, B) \vee R(B, A)$

Models

Let \mathcal{M} be a model for \mathcal{L} , with domain $D = \{a, b, c, d\}$, and interpretation function σ :

1. $A^\sigma = a, B^\sigma = b, C^\sigma = c, D^\sigma = d.$
2. $R^\sigma = \{(b, a), (c, d)\}.$
3. $P^\sigma = \{b, c\}.$
4. $Q^\sigma = \{a, d\}.$

Which of the following formulas are satisfied by \mathcal{M} .

1. $\forall x.P(x) \wedge \neg Q(x).$

Models

Let \mathcal{M} be a model for \mathcal{L} , with domain $D = \{a, b, c, d\}$, and interpretation function σ :

1. $A^\sigma = a, B^\sigma = b, C^\sigma = c, D^\sigma = d.$
2. $R^\sigma = \{(b, a), (c, d)\}.$
3. $P^\sigma = \{b, c\}.$
4. $Q^\sigma = \{a, d\}.$

Which of the following formulas are satisfied by \mathcal{M} .

1. $\forall x.P(x) \rightarrow \neg \exists y.R(y, x).$

Models

Let \mathcal{M} be a model for \mathcal{L} , with domain $D = \{a, b, c, d\}$, and interpretation function σ :

1. $A^\sigma = a, B^\sigma = b, C^\sigma = c, D^\sigma = d.$
2. $R^\sigma = \{(b, a), (c, d)\}.$
3. $P^\sigma = \{b, c\}.$
4. $Q^\sigma = \{a, d\}.$

Which of the following formulas are satisfied by \mathcal{M} .

1. $\forall x.Q(x) \rightarrow \neg \exists y.R(y, x).$

Models

Let \mathcal{M} be a model for \mathcal{L} , with domain $D = \{a, b, c, d\}$, and interpretation function σ :

1. $A^\sigma = a, B^\sigma = b, C^\sigma = c, D^\sigma = d.$
2. $R^\sigma = \{(b, a), (c, d)\}.$
3. $P^\sigma = \{b, c\}.$
4. $Q^\sigma = \{a, d\}.$

Which of the following formulas are satisfied by \mathcal{M} .

1. $\forall x.Q(x) \rightarrow \exists y.R(y, x).$

Models

Let \mathcal{M} be a model for \mathcal{L} , with domain $D = \{a, b, c, d\}$, and interpretation function σ :

1. $A^\sigma = a, B^\sigma = b, C^\sigma = c, D^\sigma = d.$
2. $R^\sigma = \{(b, a), (c, d)\}.$
3. $P^\sigma = \{b, c\}.$
4. $Q^\sigma = \{a, d\}.$

Which of the following formulas are satisfied by \mathcal{M} .

1. $\exists x, y. (P(x) \wedge Q(y)) \rightarrow R(y, x).$

Resolution Question

Consider the following sentences:

1. Marcus was a man.
2. Marcus was a Roman.
3. All men are people.
4. Caesar was a ruler.
5. All Romans were either loyal to Caesar or hated him (or both).
6. Everyone is loyal to someone.
7. People only try to assassinate rulers they are not loyal to.
8. Marcus tried to assassinate Caesar.

Covert to First Order Logic

1. Marcus was a man.

man(marcus)

2. Marcus was a Roman.

roman(marcus)

3. All men are people.

$\forall X. man(X) \rightarrow person(X)$

4. Caesar was a ruler.

ruler(caesar)

5. All Romans were either loyal to Caesar or hated him (or both).

$\forall X. roman(X) \rightarrow loyal_to(X, caesar) \vee hates(X, caesar)$

6. Everyone is loyal to someone.

$\forall X. \exists Y. person(Y) \wedge loyal_to(X, Y)$

7. People only try to assassinate rulers they are not loyal to.

$\forall X, Y. tried_to_kill(X, Y) \rightarrow \neg loyal_to(X, Y)$

8. Marcus tried to assassinate Caesar.

tried_to_kill(marcus, caesar)

Covert to Clauses

1. $\text{man}(\text{marcus})$
2. $\text{roman}(\text{marcus})$
3. $\forall X. \text{man}(X) \rightarrow \text{person}(X)$
 $(\neg \text{man}(X) \vee \text{person}(X))$
4. $\text{ruler}(\text{caesar})$
5. $\forall X. \text{roman}(X) \rightarrow \text{loyal_to}(X, \text{caesar}) \vee \text{hates}(X, \text{caesar})$
 $(\neg \text{roman}(X) \vee \text{loyal_to}(X, \text{caesar}) \vee \text{hates}(X, \text{caesar}))$
6. $\forall X. \exists Y. \text{person}(Y) \wedge \text{loyal_to}(X, Y)$
 $\text{person}(f(X))$
 $\text{loyal_to}(X, f(X))$
7. $\forall X, Y. \text{tried_to_kill}(X, Y) \rightarrow \neg \text{loyal_to}(X, Y)$
 $(\neg \text{tried_to_kill}(X, Y) \vee \neg \text{loyal_to}(X, Y))$
8. Marcus tried to assassinate Caesar.
 $\text{tried_to_kill}(\text{marcus}, \text{caesar})$

Query

1. "Who hated Caesar?"

2. First order logic:

$$\exists Z. \text{hates}(Z, \text{caesar})$$

3. Negate:

$$\forall Z. \neg \text{hates}(Z, \text{caesar})$$

4. Clausal form.

$$\neg \text{hates}(Z, \text{caesar}) \vee \text{answer}(Z)$$

Resolution Proof

1. $man(marcus)$
2. $roman(marcus)$
3. $(\neg man(X) \vee person(X))$
4. $ruler(caesar)$
5. $(\neg roman(X) \vee loyal_to(X, caesar) \vee hates(X, caesar))$
6. $person(f(X))$
7. $loyal_to(X, f(X))$
8. $(\neg tried_to_kill(X, Y) \vee \neg loyal_to(X, Y))$
9. $\text{tried_to_kill}(marcus, caesar)$
10. $\neg hates(Z, caesar) \vee answer(Z)$
11. $R[9, 8a]\{X=marcus, Y=caesar\} \neg loyal_to(marcus, caesar)$

Resolution Proof

1. $man(marcus)$
2. $roman(marcus)$
3. $(\neg man(X) \vee person(X))$
4. $ruler(caesar)$
5. $(\neg roman(X) \vee \text{loyal_to}(X, caesar) \vee \text{hates}(X, caesar))$
6. $person(f(X))$
7. $\text{loyal_to}(X, f(X))$
8. $(\neg \text{tried_to_kill}(X, Y) \vee \neg \text{loyal_to}(X, Y))$
9. $\text{tried_to_kill}(marcus, caesar)$
10. $\neg \text{hates}(Z, caesar) \vee \text{answer}(Z)$
11. $R[9, 8a]\{X=marcus, Y=caesar\} \neg \text{loyal_to}(marcus, caesar)$
12. $R[11, 5c]\{X=marcus\} \neg \text{roman}(marcus) \vee \text{hates}(marcus, caesar)$

Resolution Proof

1. $\text{man}(\text{marcus})$
2. $\text{roman}(\text{marcus})$
3. $(\neg \text{man}(X) \vee \text{person}(X))$
4. $\text{ruler}(\text{caesar})$
5. $(\neg \text{roman}(X) \vee \text{loyal_to}(X, \text{caesar}) \vee \text{hates}(X, \text{caesar}))$
6. $\text{person}(f(X))$
7. $\text{loyal_to}(X, f(X))$
8. $(\neg \text{tried_to_kill}(X, Y) \vee \neg \text{loyal_to}(X, Y))$
9. $\text{tried_to_kill}(\text{marcus}, \text{caesar})$
10. $\neg \text{hates}(Z, \text{caesar}) \vee \text{answer}(Z)$
11. $R[9, 8a]\{X = \text{marcus}, Y = \text{caesar}\} \neg \text{loyal_to}(\text{marcus}, \text{caesar})$
12. $R[11, 5c]\{X = \text{marcus}\} \neg \text{roman}(\text{marcus}) \vee \text{hates}(\text{marcus}, \text{caesar})$
13. $R[12a, 2]\{} \text{hates}(\text{marcus}, \text{caesar})$

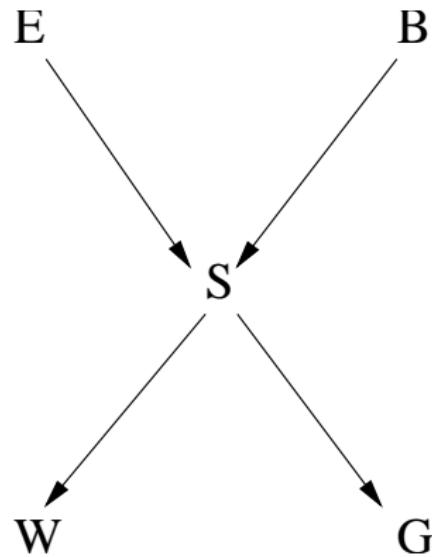
Resolution Proof

1. $\text{man}(\text{marcus})$
2. $\text{roman}(\text{marcus})$
3. $(\neg \text{man}(X) \vee \text{person}(X))$
4. $\text{ruler}(\text{caesar})$
5. $(\neg \text{roman}(X) \vee \text{loyal_to}(X, \text{caesar}) \vee \text{hates}(X, \text{caesar}))$
6. $\text{person}(f(X))$
7. $\text{loyal_to}(X, f(X))$
8. $(\neg \text{tried_to_kill}(X, Y) \vee \neg \text{loyal_to}(X, Y))$
9. $\text{tried_to_kill}(\text{marcus}, \text{caesar})$
10. $\neg \text{hates}(Z, \text{caesar}) \vee \text{answer}(Z)$
11. $R[9, 8a]\{X = \text{marcus}, Y = \text{caesar}\} \neg \text{loyal_to}(\text{marcus}, \text{caesar})$
12. $R[11, 5c]\{X = \text{marcus}\} \neg \text{roman}(\text{marcus}) \vee \text{hates}(\text{marcus}, \text{caesar})$
13. $R[12a, 2]\{\} \text{hates}(\text{marcus}, \text{caesar})$
14. $R[13, 10]\{Z = \text{marcus}\} \text{ answer}(\text{marcus})$

Tutorial Examples Uncertainty

March 24, 2015

Inference in Bayes Nets



$$P(E, S, B, W, G) = P(E)P(B)P(S|E, B)P(W|S)P(G|S)$$

Inference in Bayes Nets

$P(E)$	e	$\neg e$
	$1/10$	$9/10$

$P(B)$	b	$\neg b$
	$1/10$	$9/10$

$P(S E, B)$	s	$\neg s$
$e \wedge b$	$9/10$	$1/10$
$e \wedge \neg b$	$2/10$	$8/10$
$\neg e \wedge b$	$8/10$	$2/10$
$\neg e \wedge \neg b$	0	1

$P(W S)$	w	$\neg w$
s	$8/10$	$2/10$
$\neg s$	$2/10$	$8/10$

$P(G S)$	g	$\neg g$
s	$1/2$	$1/2$
$\neg s$	0	1

Inference in Bayes Nets

- ▶ Given the alarm went off (s) what is the probability that Mrs. Gibbons phones you (g)?

Inference in Bayes Nets

- Given the alarm went off (s) what is the probability that Mrs. Gibbons phones you (g)? probability that the alarm went off (s)?

$$P(g|s) = 1/2$$

Inference in Bayes Nets

- ▶ Given that Mrs. Gibbons phones you (g) what is the probability the alarm went off (s)?

Inference in Bayes Nets

- ▶ Given that Mrs. Gibbons phones you (g) what is the probability the alarm went off (s)?
1. Bayes Rule says: $P(S|g) = P(g|S) * P(S)/P(g)$
 2. $P(-s|g) = P(g|-s) * P(-s)/P(g) = 0 * P(-s)/P(g) = 0.$
 3. Therefore $P(s|g) = 1$ ($P(s|g) + P(-s|g)$ must sum to 1).

$$P(s|g) = 1 \quad P(-s|g) = 0$$

Alternatively: $-s \rightarrow -g$, so $g \rightarrow s$, so $P(s|g) = 1$.

Inference in Bayes Nets

- ▶ Given that Mrs. Gibbons phones you (g) what is the probability the alarm went off (s)?

$$\begin{aligned} P(g) &= P(g \wedge s) + P(g \wedge \neg s) \\ &= P(g|s) \times P(s) + P(g|\neg s) \times P(\neg s) \end{aligned}$$

Inference in Bayes Nets

- ▶ Say that there was a burglary (b) and but no earthquake ($\neg e$), what is the expression specifying the posterior probability of Dr. Watson phoning you (w) given the evidence. (You do not need to calculate a numeric answer, just give the probability expression).

Inference in Bayes Nets

- ▶ Say that there was a burglary (b) and but no earthquake (-e), what is the expression specifying the posterior probability of Dr. Watson phoning you (w) given the evidence. (You do not need to calculate a numeric answer, just give the probability expression).

$$P(w|b, -e)$$

Inference in Bayes Nets

- ▶ What is $P(G|S)$? (i.e., the four probability values) $P(g|s)$,
 $P(-g|s)$, $P(g|-s)$, $P(-g|-s)$.

Inference in Bayes Nets

- ▶ What is $P(G|S)$? (i.e., the four probability values $P(g|s)$, $P(-g|s)$, $P(g|-s)$, $P(-g|-s)$).

$$\begin{array}{ll} P(g|s) = 1/2 & P(-g|s) = 1/2 \\ P(-g|-s) = 0 & P(g|-s) = 1 \end{array}$$

Inference in Bayes Nets

- ▶ What is $P(G|S \wedge W)$? (i.e., the 8 probability values $P(g|s \wedge w)$, $P(g|s \wedge \neg w)$, \dots , $P(\neg g| \neg s \wedge \neg w)$).

Inference in Bayes Nets

- ▶ What is $P(G|S \wedge W)$? (i.e., the 8 probability values $P(g|s \wedge w)$, $P(g|s \wedge \neg w)$, \dots , $P(\neg g| \neg s \wedge \neg w)$).

$$\begin{array}{lllll} P(g|s, \neg w) & = & P(g|s, w) & = & P(g|s) & = & 1/2 \\ P(\neg g|s, \neg w) & = & P(\neg g|s, w) & = & P(\neg g|s) & = & 1/2 \\ P(g| \neg s, \neg w) & = & P(g| \neg s, w) & = & P(g| \neg s) & = & 0 \\ P(\neg g| \neg s, \neg w) & = & P(\neg g| \neg s, w) & = & P(\neg g| \neg s) & = & 1 \end{array}$$

Inference in Bayes Nets

- ▶ What do these values tell us about the relationship between G , W and S ?

G is conditionally independent of W given S

Inference in Bayes Nets

- ▶ What do these values tell us about the relationship between G , W and S ?

Inference in Bayes Nets

- ▶ What is $P(G|W)$? (i.e., the four probability values $P(g|w)$, $P(-g|w)$, $P(g|-w)$, and $P(-g|-w)$).

Inference in Bayes Nets

- ▶ What is $P(G|W)$? (i.e., the four probability values $P(g|w)$, $P(-g|w)$, $P(g|-w)$, and $P(-g|-w)$).

Must do variable elimination.

Inference in Bayes Nets

- ▶ What is $P(G|W)$? (i.e., the four probability values $P(g|w)$, $P(-g|w)$, $P(g|-w)$, and $P(-g|-w)$).
- ▶ Query variable is G .
- ▶ First run of VE, evidence is $W = w$.
- ▶ Second run of VE, evidence is $W = -w$.
- ▶ Use same ordering for both runs of VE: E, B, S, G .
- ▶ With same ordering some factors can be reused between the two runs of VE.

Inference in Bayes Nets

- ▶ What is $P(G|W)$? (i.e., the four probability values $P(g|w)$, $P(-g|w)$, $P(g|-w)$, and $P(-g|-w)$).
 1. E : $P(E)$, $P(S|E, B)$
 2. B : $P(B)$,
 3. S : $P(w|S)$, $P(S|G)$
 4. G :

Inference in Bayes Nets

- ▶ What is $P(G|W)$? (i.e., the four probability values $P(g|w)$, $P(-g|w)$, $P(g|-w)$, and $P(-g|-w)$).
 1. E : $P(E)$, $P(S|E, B)$
 2. B : $P(B)$,
 3. S : $P(w|S)$, $P(S|G)$
 4. G :

$$\begin{aligned}F_1(S, B) &= \sum_E P(E) \times P(S|E, B) \\&= P(e) \times P(S|e, B) + P(-e) \times P(S|-e, B)\end{aligned}$$

$$\begin{aligned}F_1(-s, -b) &= P(e)P(-s, e, -b) + P(-e)P(-s, -e, -b) \\&= 0.1 \times 0.8 + 0.9 \times 1 = 0.98\end{aligned}$$

$$\begin{aligned}F_1(-s, b) &= P(e)P(-s, e, b) + P(-e)P(-s, -e, b) \\&= 0.1 \times 0.1 + 0.9 \times 0.2 = 0.19\end{aligned}$$

$$\begin{aligned}F_1(s, -b) &= P(e)P(s, e, -b) + P(-e)P(s, -e, -b) \\&= 0.1 \times 0.2 + 0.9 \times 0 = 0.02\end{aligned}$$

$$\begin{aligned}F_1(s, b) &= P(e)P(s, e, b) + P(-e)P(s, -e, b) \\&= 0.1 \times 0.9 + 0.9 \times 0.8 = 0.81\end{aligned}$$

Inference in Bayes Nets

1. E : $P(E)$, $P(S|E, B)$
2. B : $P(B)$, $F_1(S, B)$
3. S : $P(w|S)$, $P(S|G)$
4. G :

$$\begin{aligned}F_2(S) &= \sum_B P(B) \times F_1(S, B) \\&= P(b)F_1(S, b) + P(-b)F_1(S, -b)\end{aligned}$$

$$\begin{aligned}F_2(-s) &= P(b)F_1(-s, b) + P(-b)F_1(-s, -b) \\&= 0.1 \times 0.19 + 0.9 \times 0.98 = 0.901\end{aligned}$$

$$\begin{aligned}F_2(s) &= P(b)F_1(s, b) + P(-b)F_1(s, -b) \\&= 0.1 \times 0.81 + 0.9 \times 0.02 = 0.099\end{aligned}$$

Inference in Bayes Nets

1. E : $P(E)$, $P(S|E, B)$
2. B : $P(B)$, $F_1(S, B)$
3. S : $P(w|S)$, $P(S|G)$, $F_2(S)$
4. G :

$$\begin{aligned}F_3(G) &= \sum_S P(w|S) \times P(S|G) \times F_2(S) \\&= P(w|s)P(s|G)F_2(s) + P(w|-s)P(-s|G)F_2(-s)\end{aligned}$$

$$\begin{aligned}F_3(-g) &= P(w|s)P(s|-g)F_2(s) + P(w|-s)P(-s|-g)F_2(-s) \\&= 0.8 \times 0.5 \times 0.099 + 0.2 \times 1 \times 0.901 = 0.2198 \\F_3(g) &= P(w|s)P(s|g)F_2(s) + P(w|-s)P(-s|g)F_2(-s) \\&= 0.8 \times 0.5 \times 0.099 + 0.2 \times 0 \times 0.901 = 0.0396\end{aligned}$$

Inference in Bayes Nets

1. E : $P(E)$, $P(S|E, B)$
2. B : $P(B)$, $F_1(S, B)$
3. S : $P(w|S)$, $P(S|G)$, $F_2(S)$
4. G : $F_3(G)$

Normalize $F_3(G)$:

$$P(-g|w) = \frac{0.2198}{0.2198+0.0396} = 0.8473$$

$$P(g|w) = \frac{0.0396}{0.2198+0.0396} = 0.1527$$

Inference in Bayes Nets

- ▶ Now $P(G| -w)$?
 1. E : $P(E)$, $P(S|E, B)$
 2. B : $P(B)$,
 3. S : $P(-w|S)$, $P(S|G)$
 4. G :

Already computed as $F_1(S, B)$

Inference in Bayes Nets

1. E : $P(E)$, $P(S|E, B)$
2. B : $P(B)$, $F_1(S, B)$
3. S : $P(-w|S)$, $P(S|G)$
4. G :

Already computed as $F_2(S)$

Inference in Bayes Nets

1. E : $P(E)$, $P(S|E, B)$
2. B : $P(B)$, $F_1(S, B)$
3. S : $P(-w|S)$, $P(S|G)$, $F_2(S)$
4. G :

$$\begin{aligned}F_3(G) &= \sum_S P(-w|S) \times P(S|G) \times F_2(S) \\&= P(-w|s)P(s|G)F_2(s) + P(-w|-s)P(-s|G)F_2(-s)\end{aligned}$$

$$\begin{aligned}F_3(-g) &= P(-w|s)P(s|-g)F_2(s) + P(-w|-s)P(-s|-g)F_2(-s) \\&= 0.2 \times 0.5 \times 0.099 + 0.8 \times 1 \times 0.901 = 0.7307\end{aligned}$$

$$\begin{aligned}F_3(g) &= P(-w|s)P(s|g)F_2(s) + P(-w|-s)P(-s|g)F_2(-s) \\&= 0.2 \times 0.5 \times 0.099 + 0.8 \times 0 \times 0.901 = 0.0099\end{aligned}$$

Inference in Bayes Nets

1. E : $P(E)$, $P(S|E, B)$
2. B : $P(B)$, $F_1(S, B)$
3. S : $P(-w|S)$, $P(S|G)$, $F_2(S)$
4. G : $F_3(G)$

Normalize $F_3(G)$:

$$P(-g|w) = \frac{0.7307}{0.7307+0.0099} = 0.9866$$

$$P(g|w) = \frac{0.0099}{0.2198+0.00099} = 0.0134$$

Inference in Bayes Nets

- ▶ What do these values tell us about the relationship between G and W , and why does this relationship differ when we know S ?

Inference in Bayes Nets

- ▶ What do these values tell us about the relationship between G and W , and why does this relationship differ when we know S ?

G and W are not independent of each other. But when S is known they become independent.

Uniform Cost Search Overview

- Paths in the State Space ordered by cost

Path	Cost
$\langle S \rangle$	0
$\langle S, a \rangle$	1.0
$\langle S, b \rangle$	1.0
$\langle S, a, c \rangle$	1.5
$\langle S, d \rangle$	2.0
...	...

Note cost is non-decreasing

Uniform Cost Search Overview

- Uniform Cost Search expands paths in non-decreasing order of cost. So it only goes down this list.

LEMMA 1

- It does not miss any paths on this list. LEMMA 2

Path	Cost
<S>	0
<S, a>	1.0
<S,a,b>	1.0
<S,a,b,c>	3.0
<S,a,b,d,e>	4.0
...	...

Uniform Cost Search Overview

It does not miss any paths on this list. LEMMA 2

Path	Cost
<S>	0
<S, a>	1.0
<S,a,b>	1.0
<S,a,b,c>	3.0
<S,a,b,d,e>	4.0
...	...

- If $\langle S, a, b, d, e \rangle$ is expanded next, $\langle S, a, b, c \rangle$ must already been expanded. If not it or $\langle S, a, b, c \rangle$ or $\langle S, a, b \rangle$ or $\langle S, a \rangle$ or $\langle S \rangle$ must be available for expansion (on OPEN) and would be expanded next as they all have lower cost than $\langle S, a, b, d, e \rangle$

Uniform Cost Search Overview

Thus working its way down such a list of paths in this order the first path achieving the goal that Uniform cost search finds will be the cheapest way of achieving the goal.