Lecture overview

- What is artificial intelligence?
- Agents acting in an environment

Learning objectives: at the end of the class, you should be able to

- describe what an intelligent agent is
- identify the goals of Artificial Intelligence
- classify the inputs and the outputs of various agents

What is artificial intelligence?

- Artificial intelligence is the synthesis and analysis of computational agents that act intelligently.
- An agent is something that acts in an environment.
- An agent acts intelligently if:
  - its actions are appropriate for its goals and circumstances
  - it is flexible to changing environments and goals
  - it learns from experience
  - it makes appropriate choices given perceptual and computational limitations

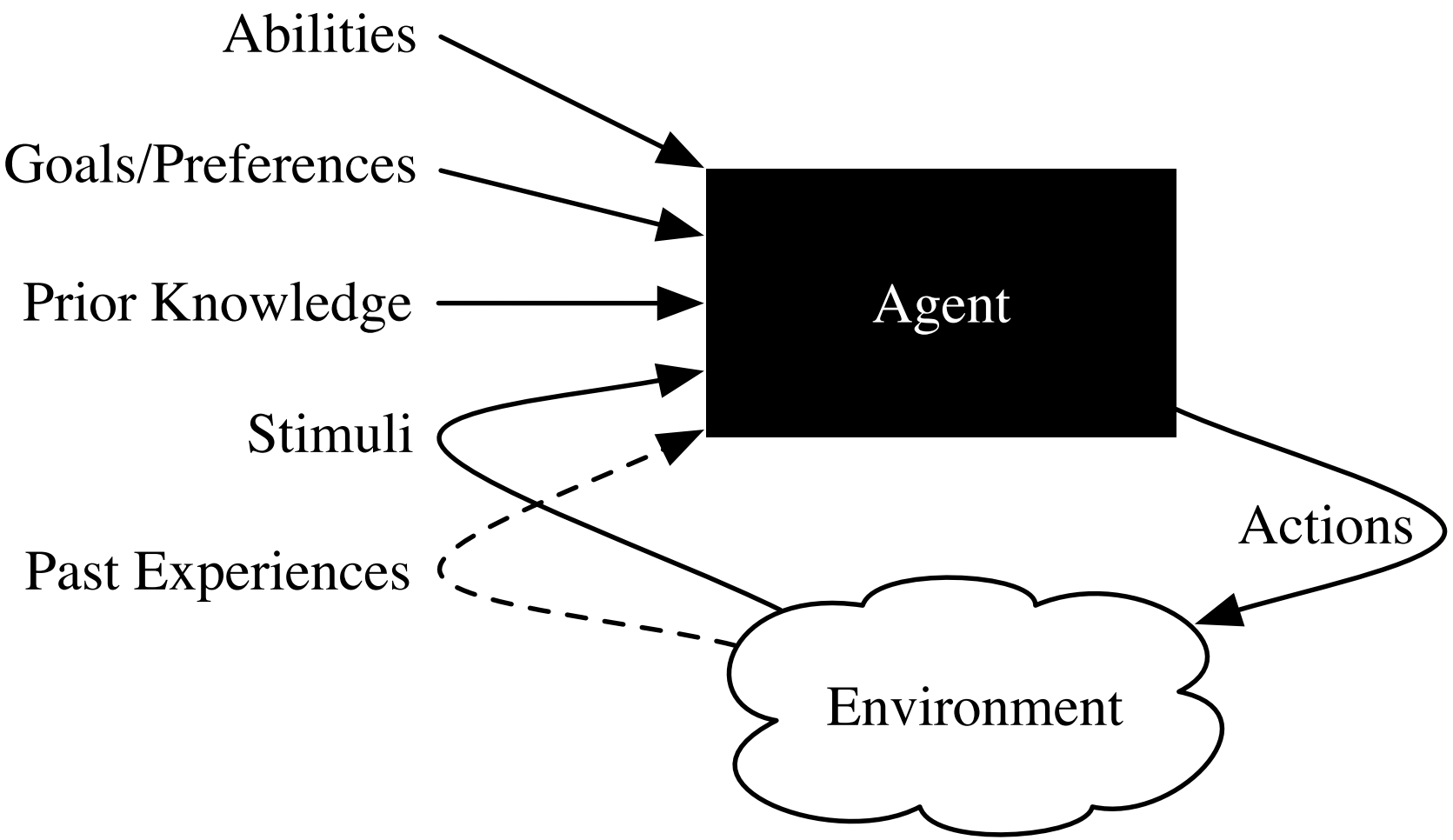
Examples of agents

- Organisations Microsoft, Al Qaeda, Government of Canada, UBC, CS Dept,...
- People teacher, physician, stock trader, engineer, researcher, travel agent, farmer, waiter...
- Computers/devices thermostat, user interface, airplane controller, network controller, game, advising system, tutoring system, diagnostic assistant, robot, Google car, Mars rover...
- Animals dog, mouse, bird, insect, worm, bacterium, bacteria...
- book(?), sentence(?), word(?), letter(?)
  Can a book or article *do* things?
  Convince? Argue? Inspire? Cause people to act differently?

Goals of artificial intelligence

- Scientific goal: to understand the principles that make intelligent behavior possible in natural or artificial systems.
  - analyze natural and artificial agents
  - formulate and test hypotheses about what it takes to construct intelligent agents
  - design, build, and experiment with computational systems that perform tasks that require intelligence
- Engineering goal: design useful, intelligent artifacts.
- Analogy between studying flying machines and thinking machines.

Agents acting in an environment: inputs and output



Inputs to an agent

- Abilities — the set of possible actions it can perform
- Goals/Preferences — what it wants, its desires, its values,...
- Prior Knowledge — what it comes into being knowing, what it doesn't get from experience,...
- History of stimuli
  - (current) stimuli — what it receives from environment now (observations, percepts)
  - past experiences — what it has received in the past

Example agent: autonomous car

- abilities: steer, accelerate, brake
- goals: safety, get to destination, timeliness ...
- prior knowledge: street maps, what signs mean, what to stop for ...
- stimuli: vision, laser, GPS, voice commands ...
- past experiences: how breaking and steering affects direction and speed...

Example agent: robot

- abilities: movement, grippers, speech, facial expressions,. . .
- goals: deliver food, rescue people, score goals, explore,. . .
- prior knowledge: what is important feature, categories of objects, what a sensor tell us,. . .
- stimuli: vision, sonar, sound, speech recognition, gesture recognition,. . .
- past experiences: effect of steering, slipperiness, how people move,. . .

Example agent: teacher

- abilities: present new concept, drill, give test, explain concept,. . .
- goals: particular knowledge, skills, inquisitiveness, social skills,. . .
- prior knowledge: subject material, teaching strategies,. . .
- stimuli: test results, facial expressions, errors, focus,. . .
- past experiences: prior test results, effects of teaching strategies, . . .

Example agent: thermostat for heater

- abilities: turn heater on or off
- goals: conformable temperature, save fuel, save money
- prior knowledge: 24 hour cycle, weekends
- stimuli: temperature, set temperature, who is home, outside temperature
- past experiences: when people come and go, who likes what temperature

Example agent: medical doctor

- abilities:
- goals:
- prior knowledge:
- stimuli:
- past experiences:

Example agent: Apple Inc.

- abilities:
- goals:
- prior knowledge:
- stimuli:
- past experiences:

Other Agents

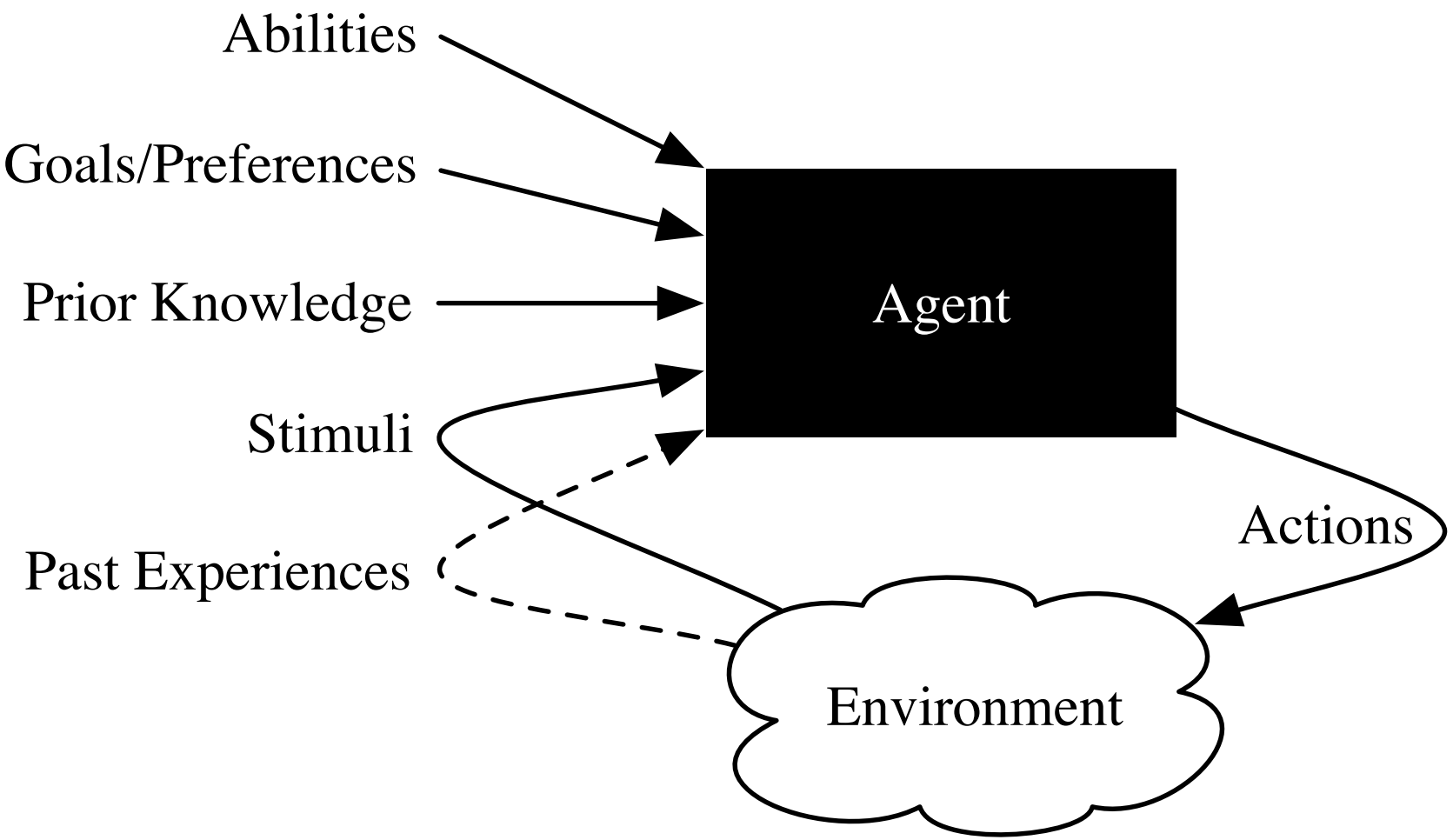
- user interface
- bee
- smart home
- ...

- abilities:
- goals:
- prior knowledge:
- stimuli:
- past experiences:

Example agent:

- abilities:
- goals:
- prior knowledge:
- stimuli:
- past experiences:

Agents acting in an environment



Learning objectives

At the end of the class you should be able to:

- characterize simplifying assumptions made in building AI systems
- determine what simplifying assumptions particular AI systems are making
- suggest what assumptions to lift to build a more intelligent system than an existing one

Dimensions

- Research proceeds by making simplifying assumptions, and gradually reducing them.
- Each simplifying assumption gives a dimension of complexity
  - multiple values in a dimension: from simple to complex
  - simplifying assumptions can be relaxed in various combinations

Dimensions of complexity

Dimension	Values
Modularity	flat, modular, hierarchical
Planning horizon	non-planning, finite stage, indefinite stage, infinite stage
Representation	states, features, relations
Computational limits	perfect rationality, bounded rationality
Learning	knowledge is given, knowledge is learned
Sensing uncertainty	fully observable, partially observable
Effect uncertainty	deterministic, stochastic
Preference	goals, complex preferences
Number of agents	single agent, multiple agents
Interaction	offline, online

Modularity

- Model at one level of abstraction: flat
- Model with interacting modules that can be understood separately: modular
- Model with modules that are (recursively) decomposed into modules: hierarchical
- Example: Planning a trip from here to a see the Mona Lisa in Paris.
- Flat representations are adequate for simple systems.
- Complex biological systems, computer systems, organizations are all hierarchical
- A flat description is either continuous or discrete. Hierarchical reasoning is often a hybrid of continuous and discrete.

Planning horizon

...how far the agent looks into the future when deciding what to do.

- Static: world does not change
- Finite stage: agent reasons about a fixed finite number of time steps
- Indefinite stage: agent reasons about a finite, but not predetermined, number of time steps
- Infinite stage: the agent plans for going on forever (process oriented)

Representation

Much of modern AI is about finding compact representations and exploiting the compactness for computational gains.
A agent can reason in terms of:

- Explicit states — a state is one way the world could be
- Features or propositions.
  - States can be described using features.
  - 30 binary features can represent $2^{30} = 1,073,741,824$ states.
- Individuals and relations
  - There is a feature for each relationship on each tuple of individuals.
  - Often an agent can reason without knowing the individuals or when there are infinitely many individuals.

Computational limits

- Perfect rationality: the agent can determine the best course of action, without taking into account its limited computational resources.
- Bounded rationality: the agent must make good decisions based on its perceptual, computational and memory limitations.

Learning from experience

Whether the model is fully specified a priori:

- Knowledge is given.
- Knowledge is learned from data or past experience.

. . . always some mix of prior (innate, programmed) knowledge and learning (nature vs nurture).

- Learning is impossible without prior knowledge (bias).

Uncertainty

There are two dimensions for uncertainty. In each dimension an agent can have

- No uncertainty: the agent knows what is true
- Disjunctive uncertainty: there is a set of states that are possible
- Probabilistic uncertainty: a probability distribution over the worlds.

Why probability?

- Agents need to act even if they are uncertain.
- Predictions are needed to decide what to do:
  - definitive predictions: you will be run over tomorrow
  - disjunctions: be careful or you will be run over
  - point probabilities: probability you will be run over tomorrow is 0.002 if you are careful and 0.05 if you are not careful
- Acting is gambling: agents who don't use probabilities will lose to those who do.
- Probabilities can be learned from data and prior knowledge.

Sensing uncertainty

Whether an agent can determine the state from its stimuli:

- Fully-observable: the agent can observe the state of the world.
- Partially-observable: there can be a number states that are possible given the agent's stimuli.

Effect uncertainty

If an agent knew the initial state and its action, could it predict the resulting state?
The dynamics can be:

- Deterministic: the resulting state is determined from the action and the state
- Stochastic: there is uncertainty about the resulting state.

Preference

What does the agent try to achieve?

- achievement goal is a goal to achieve. This can be a complex logical formula.
- complex preferences may involve tradeoffs between various desiderata, perhaps at different times.
  - ordinal only the order matters
  - cardinal absolute values also matter

Examples: coffee delivery robot, medical doctor

Number of agents

Are there multiple reasoning agents that need to be taken into account?

- Single agent reasoning: any other agents are part of the environment.
- Multiple agent reasoning: an agent reasons strategically about the reasoning of other agents.

Agents can have their own goals: cooperative, competitive, or goals can be independent of each other

Interaction

When does the agent reason to determine what to do?

- reason offline: before acting
- reason online: while interacting with environment

Dimensions of complexity

Dimension	Values
Modularity	flat, modular, hierarchical
Planning horizon	non-planning, finite stage, indefinite stage, infinite stage
Representation	states, features, relations
Computational limits	perfect rationality, bounded rationality
Learning	knowledge is given, knowledge is learned
Sensing uncertainty	fully observable, partially observable
Effect uncertainty	deterministic, stochastic
Preference	goals, complex preferences
Number of agents	single agent, multiple agents
Interaction	offline, online

State-space search

Dimension	Values
Modularity	flat, modular, hierarchical
Planning horizon	non-planning, finite stage, indefinite stage, infinite stage
Representation	states, features, relations
Computational limits	perfect rationality, bounded rationality
Learning	knowledge is given, knowledge is learned
Sensing uncertainty	fully observable, partially observable
Effect uncertainty	deterministic, stochastic
Preference	goals, complex preferences
Number of agents	single agent, multiple agents
Interaction	offline, online

Deterministic planning

Dimension	Values
Modularity	flat, modular, hierarchical
Planning horizon	non-planning, finite stage, indefinite stage, infinite stage
Representation	states, features, relations
Computational limits	perfect rationality, bounded rationality
Learning	knowledge is given, knowledge is learned
Sensing uncertainty	fully observable, partially observable
Effect uncertainty	deterministic, stochastic
Preference	goals, complex preferences
Number of agents	single agent, multiple agents
Interaction	offline, online

Decision networks

Dimension	Values
Modularity	flat, modular, hierarchical
Planning horizon	non-planning, finite stage, indefinite stage, infinite stage
Representation	states, features, relations
Computational limits	perfect rationality, bounded rationality
Learning	knowledge is given, knowledge is learned
Sensing uncertainty	fully observable, partially observable
Effect uncertainty	deterministic, stochastic
Preference	goals, complex preferences
Number of agents	single agent, multiple agents
Interaction	offline, online

Markov decision processes (MDPs)

Dimension	Values
Modularity	flat, modular, hierarchical
Planning horizon	non-planning, finite stage, indefinite stage, infinite stage
Representation	states, features, relations
Computational limits	perfect rationality, bounded rationality
Learning	knowledge is given, knowledge is learned
Sensing uncertainty	fully observable, partially observable
Effect uncertainty	deterministic, stochastic
Preference	goals, complex preferences
Number of agents	single agent, multiple agents
Interaction	offline, online

Decision-theoretic planning

Dimension	Values
Modularity	flat, modular, hierarchical
Planning horizon	non-planning, finite stage, indefinite stage, infinite stage
Representation	states, features, relations
Computational limits	perfect rationality, bounded rationality
Learning	knowledge is given, knowledge is learned
Sensing uncertainty	fully observable, partially observable
Effect uncertainty	deterministic, stochastic
Preference	goals, complex preferences
Number of agents	single agent, multiple agents
Interaction	offline, online

Reinforcement learning

Dimension	Values
Modularity	flat, modular, hierarchical
Planning horizon	non-planning, finite stage, indefinite stage, infinite stage
Representation	states, features, relations
Computational limits	perfect rationality, bounded rationality
Learning	knowledge is given, knowledge is learned
Sensing uncertainty	fully observable, partially observable
Effect uncertainty	deterministic, stochastic
Preference	goals, complex preferences
Number of agents	single agent, multiple agents
Interaction	offline, online

Classical game theory

Dimension	Values
Modularity	flat, modular, hierarchical
Planning horizon	non-planning, finite stage, indefinite stage, infinite stage
Representation	states, features, relations
Computational limits	perfect rationality, bounded rationality
Learning	knowledge is given, knowledge is learned
Sensing uncertainty	fully observable, partially observable
Effect uncertainty	deterministic, stochastic
Preference	goals, complex preferences
Number of agents	single agent, multiple agents
Interaction	offline, online

Humans

Dimension	Values
Modularity	flat, modular, hierarchical
Planning horizon	non-planning, finite stage, indefinite stage, infinite stage
Representation	states, features, relations
Computational limits	perfect rationality, bounded rationality
Learning	knowledge is given, knowledge is learned
Sensing uncertainty	fully observable, partially observable
Effect uncertainty	deterministic, stochastic
Preference	goals, complex preferences
Number of agents	single agent, multiple agents
Interaction	offline, online

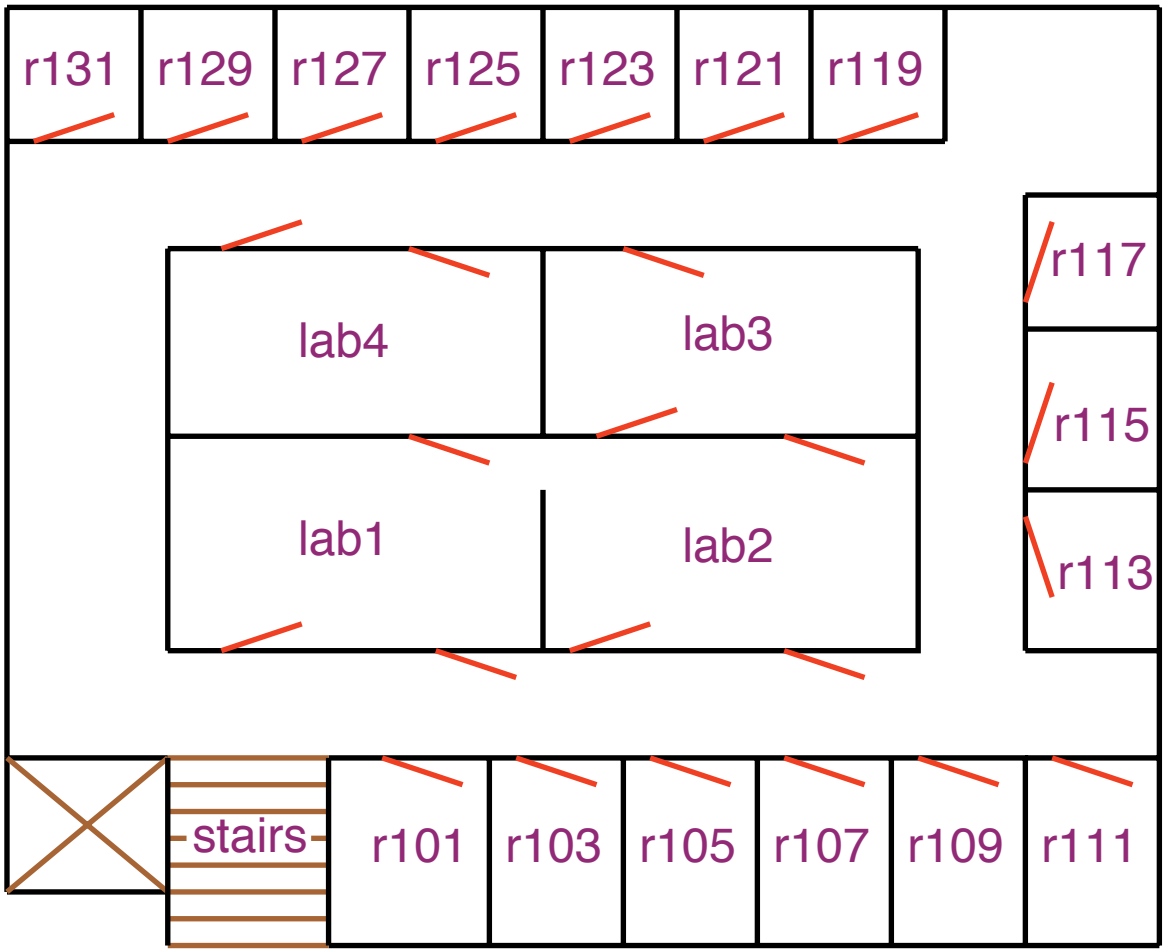
The dimensions interact in complex ways

- Partial observability makes multi-agent and indefinite horizon reasoning more complex
- Modularity interacts with uncertainty and succinctness: some levels may be fully observable, some may be partially observable
- Three values of dimensions promise to make reasoning simpler for the agent:
  - Hierarchical reasoning
  - Individuals and relations
  - Bounded rationality

Four Example Application Domains

- Autonomous delivery robot roams around an office environment and delivers coffee, parcels,. . .
- Diagnostic assistant helps a human troubleshoot problems and suggests repairs or treatments. E.g., electrical problems, medical diagnosis.
- Intelligent tutoring system teaches students in some subject area.
- Trading agent buys goods and services on your behalf.

Domain for Delivery Robot



Autonomous Delivery Robot

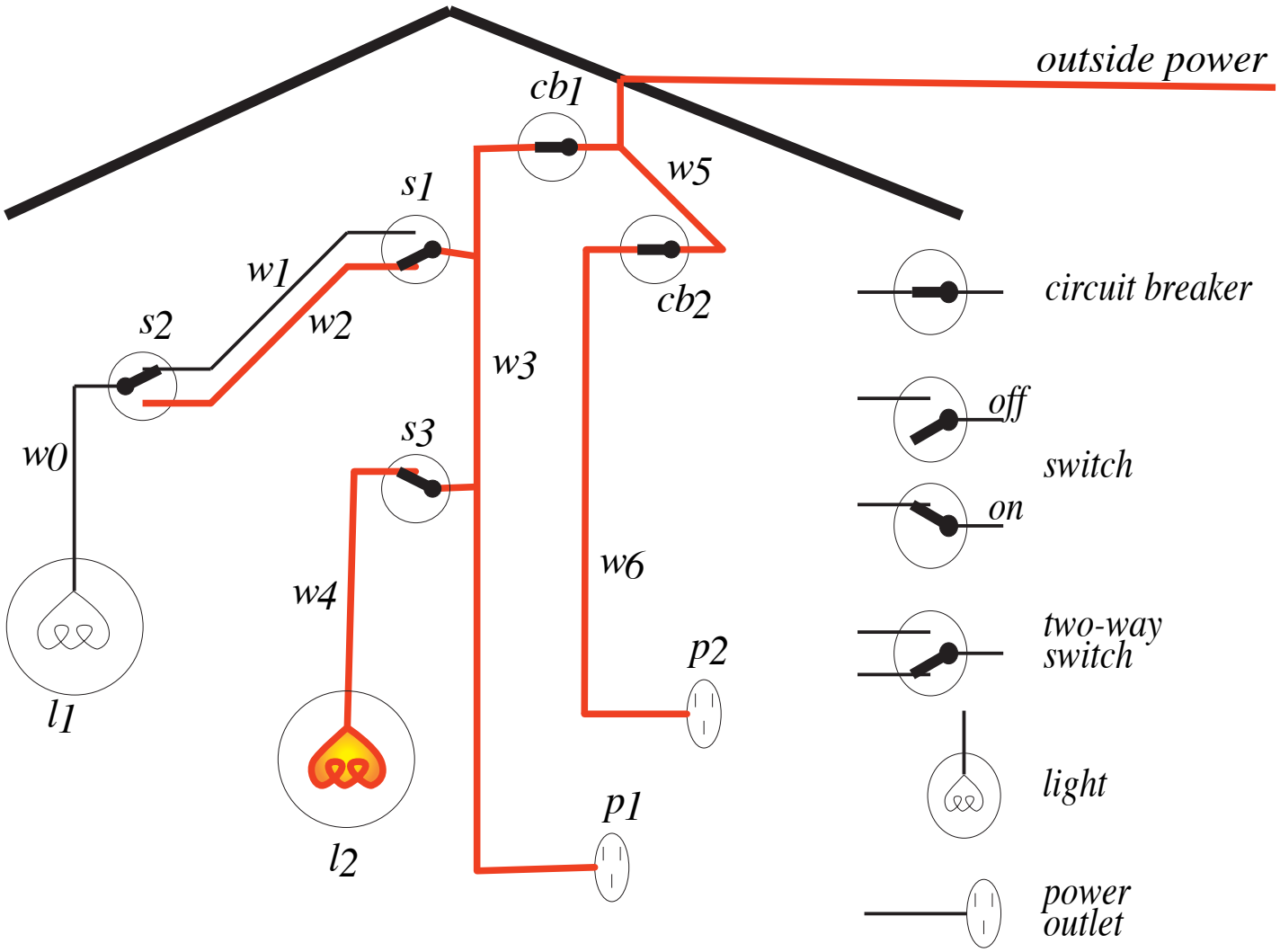
Example inputs:

- Abilities: movement, speech, pickup and place objects.
- Prior knowledge: its capabilities, objects it may encounter, maps.
- Past experience: which actions are useful and when, what objects are there, how its actions affect its position.
- Goals: what it needs to deliver and when, tradeoffs between acting quickly and acting safely.
- Observations: about its environment from cameras, sonar, sound, laser range finders, or keyboards.

What does the Delivery Robot need to do?

- Determine where Craig's office is. Where coffee is...
- Find a path between locations.
- Plan how to carry out multiple tasks.
- Make default assumptions about where Craig is.
- Make tradeoffs under uncertainty: should it go near the stairs?
- Learn from experience.
- Sense the world, avoid obstacles, pickup and put down coffee.

Domain for Diagnostic Assistant in a Smart Home



Diagnostic Assistant

Example inputs:

- Abilities: recommends fixes, ask questions.
- Prior knowledge: how switches and lights work, how malfunctions manifest themselves, what information tests provide, the side effects of repairs.
- Past experience: the effects of repairs or treatments, the prevalence of faults or diseases.
- Goals: fixing the device and tradeoffs between fixing or replacing different components.
- Observations: symptoms of a device or patient.

Subtasks for the diagnostic assistant

- Derive the effects of faults and interventions.
- Search through the space of possible fault complexes.
- Explain its reasoning to the human who is using it.
- Derive possible causes for symptoms; rule out other causes.
- Plan courses of tests and treatments to address the problems.
- Reason about the uncertainties/ambiguities given symptoms.
- Trade off alternate courses of action.
- Learn what symptoms are associated with faults, the effects of treatments, and the accuracy of tests.

Trading Agent

Trading agent interacts with an information environment to purchase goods and services.

- It acquires a users needs, desires and preferences.
- It finds what is available.
- It purchases goods and services that fit together to fulfill your preferences.
- It is difficult because users preferences and what is available can change dynamically, and some items may be useless without other items.

Trading Agent Inputs

- Abilities: acquire information, make recommendations, purchase items.
- Prior knowledge: the ontology of what things are available, where to purchase items, how to decompose a complex item.
- Past experience: how long special last, how long items take to sell out, who has good deals, what your competitors do.
- Goals: what the person wants, their tradeoff.
- Observations: what items are available, prices, number in stock,

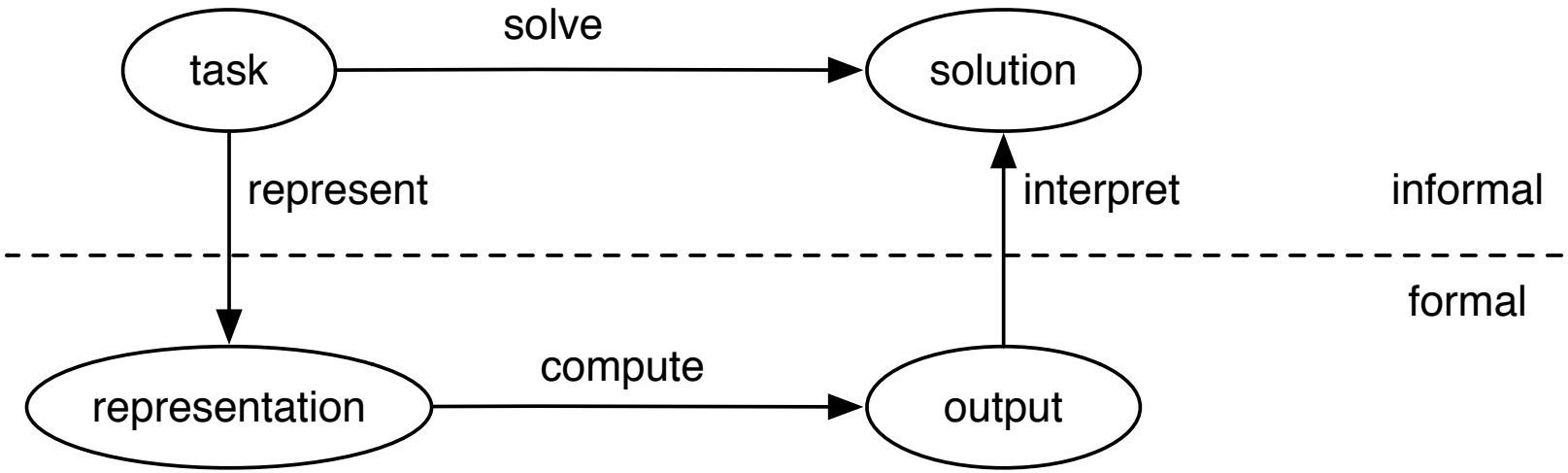
Intelligent Tutoring System

- Abilities: Present information, give tests
- Prior knowledge: subject material, primitive strategies
- Past experience: common errors, effects of teaching strategies
- Goals: the students should master subject material, gain social skills, study skills, inquisitiveness, interest
- Observations: test results, facial expressions, questions, what the student is concentrating on

Common Tasks of the Domains

- Modeling the environment Build models of the physical environment, patient, or information environment.
- Evidential reasoning or perception Given observations, determine what the world is like.
- Action Given a model of the world and a goal, determine what should be done.
- Learning from past experiences Learn about the specific case and the population of cases.

Representations



What do we want in a representation?

We want a representation to be

- rich enough to express the knowledge needed to solve the problem;
- as close to the problem as possible: compact, natural and maintainable;
- amenable to efficient computation
  - able to express features of the problem that can be exploited for computational gain
  - able to trade off accuracy and computation time and/or space
- able to be acquired from people, data and past experiences.

Defining a Solution

- Given an informal description of a problem, what is a solution?
- Typically much is left unspecified, but the unspecified parts can't be filled in arbitrarily.
- Much work in AI is motivated by common-sense reasoning. The computer needs to make common-sense conclusions about the unstated assumptions.

Quality of Solutions

- Does it matter if the answer is wrong or answers are missing?

Classes of solution:

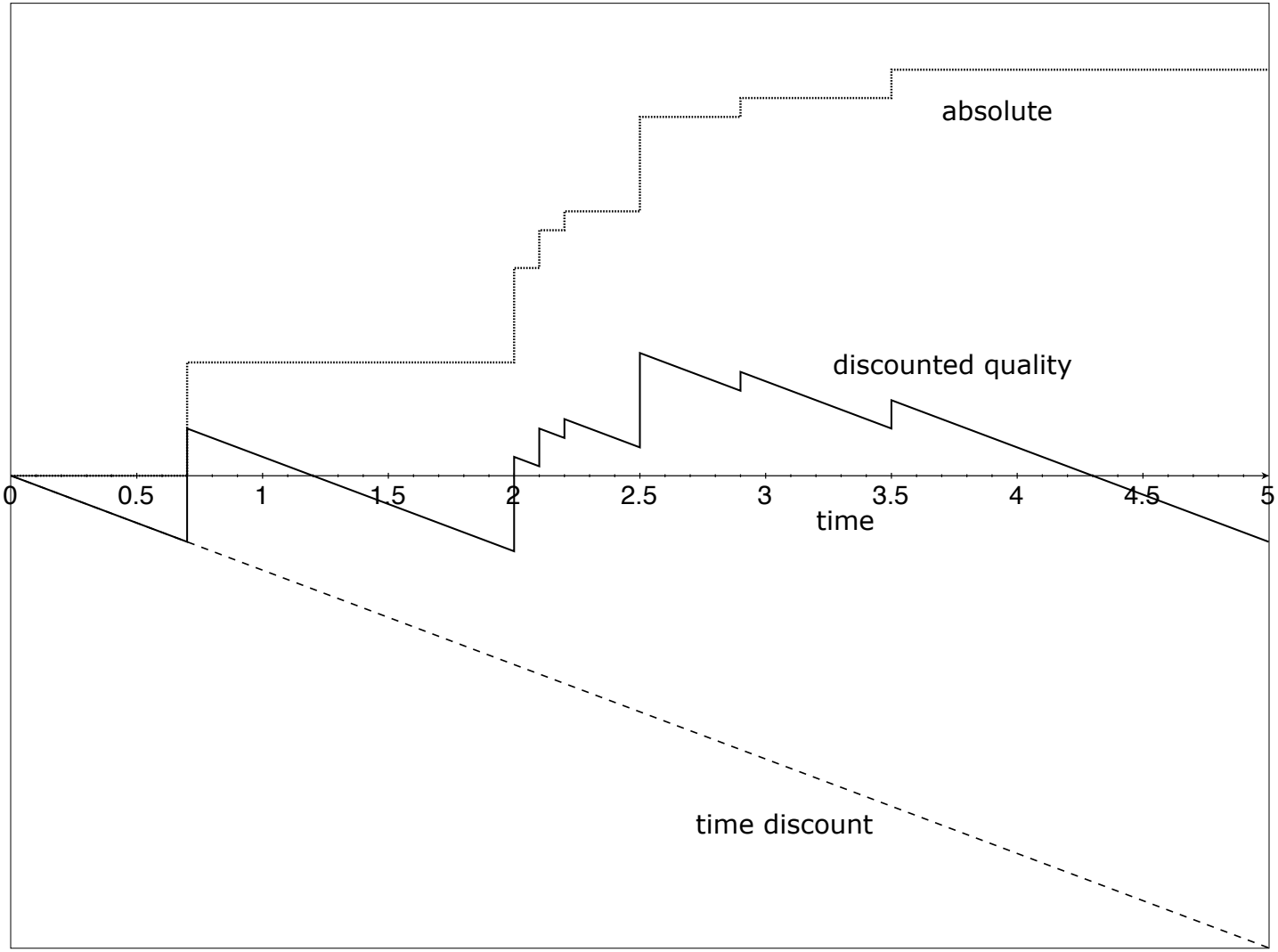
- An optimal solution is a best solution according some measure of solution quality.
- A satisficing solution is one that is good enough, according to some description of which solutions are adequate.
- An approximately optimal solution is one whose measure of quality is close to the best theoretically possible.
- A probable solution one that is likely to be a solution.

Decisions and Outcomes

- Good decisions can have bad outcomes. Bad decisions can have good outcomes.
- Information can be valuable because it leads to better decisions: value of information.
- We can often trade off computation time and solution quality. An anytime algorithm can provide a solution at any time; given more time it can produce better solutions.

An agent isn't just concerned about finding the right answer, but about acquiring the appropriate information, and computing it in a timely manner.

Solution quality and computation time



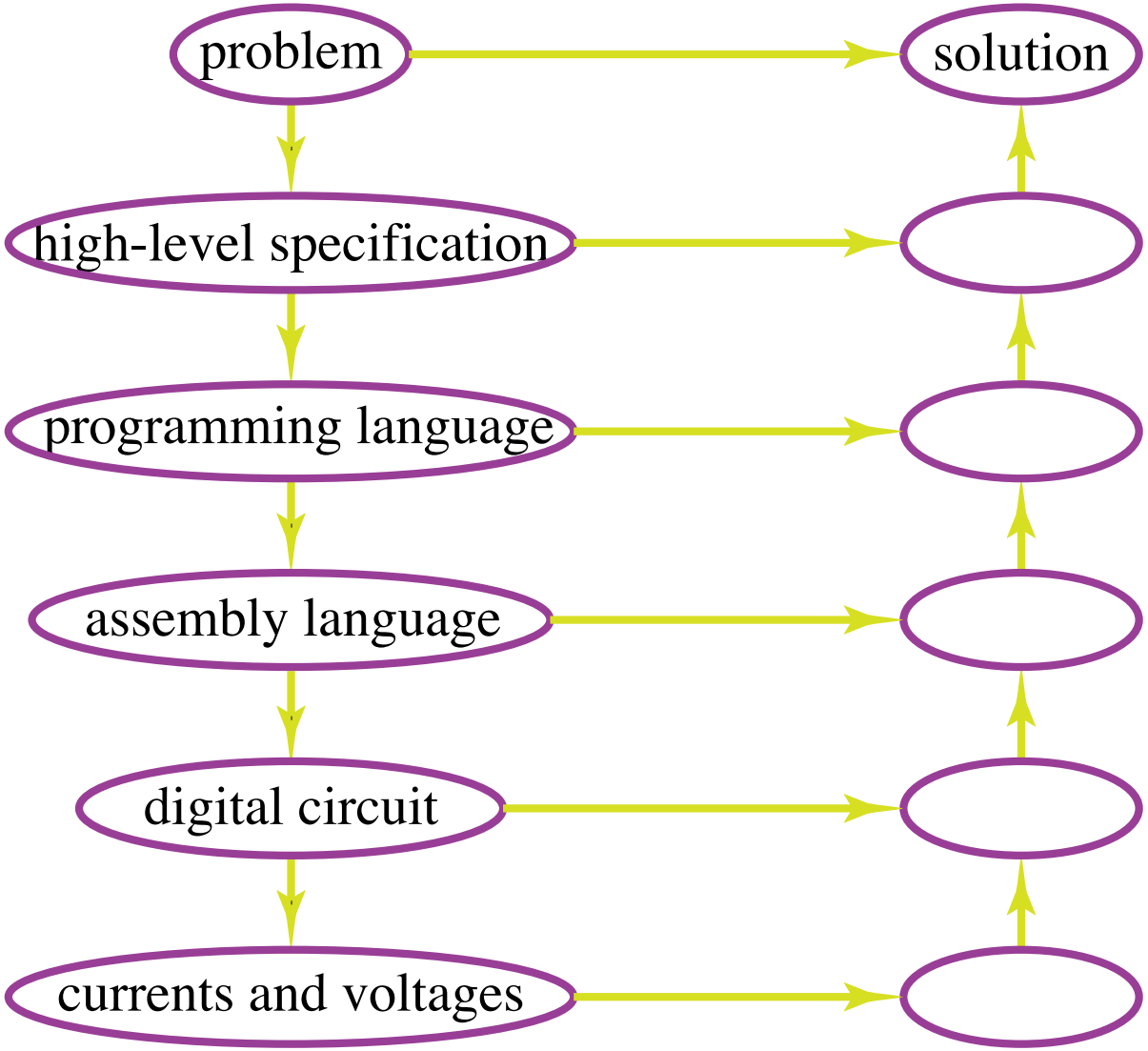
Choosing a Representation Language

We need to represent a problem to solve it on a computer.

$$\left[\begin{array}{l} \text{problem} \\ \quad \rightarrow \text{ specification of problem} \\ \qquad \rightarrow \text{ appropriate computation} \end{array}\right]$$

Example representation languages: Machine Language, C++, Java, Prolog, English

Hierarchy of representations



Physical symbol system hypothesis

- A symbol is a meaningful physical pattern that can be manipulated.
- A symbol system creates, copies, modifies and destroys symbols.

Physical symbol system hypothesis:

- A physical symbol system has the necessary and sufficient means for general intelligent action.

Knowledge & Symbol Levels

Two levels of abstraction seem to be common among biological and computational entities:

- The knowledge level is in terms of what an agent knows and what its goals are.
- The symbol level is a level of description of an agent in terms of what reasoning it is doing.

The knowledge level is about the external world to the agent. The symbol level is about what symbols an agent uses to implement the knowledge level.

Mapping from Problem to Representation

- What level of abstraction of the problem to represent?
- What individuals and relations in the world to represent?
- How can an agent represent the knowledge to ensure that the representation is natural, modular, and maintainable?
- How can an agent acquire the information from data, sensing, experience, or other agents?

Choosing a level of abstraction

- A high-level description is easier for a human to specify and understand.
- A low-level description can be more accurate and more predictive. High-level descriptions abstract away details that may be important for actually solving the problem.
- The lower the level, the more difficult it is to reason with.
- You may not know the information needed for a low-level description.

It is sometime possible to use multiple levels of abstraction.

Reasoning and acting

Reasoning is the computation required to determine what an agent should do.

- Design time reasoning and computation is carried out by the designer the agent.
- Offline computation is the computation done by the agent before it has to act.
  Background knowledge and data $\rightsquigarrow$ knowledge base.
- Online computation is the computation that's done by an agent between receiving information and acting.

Agent architectures and hierarchical control

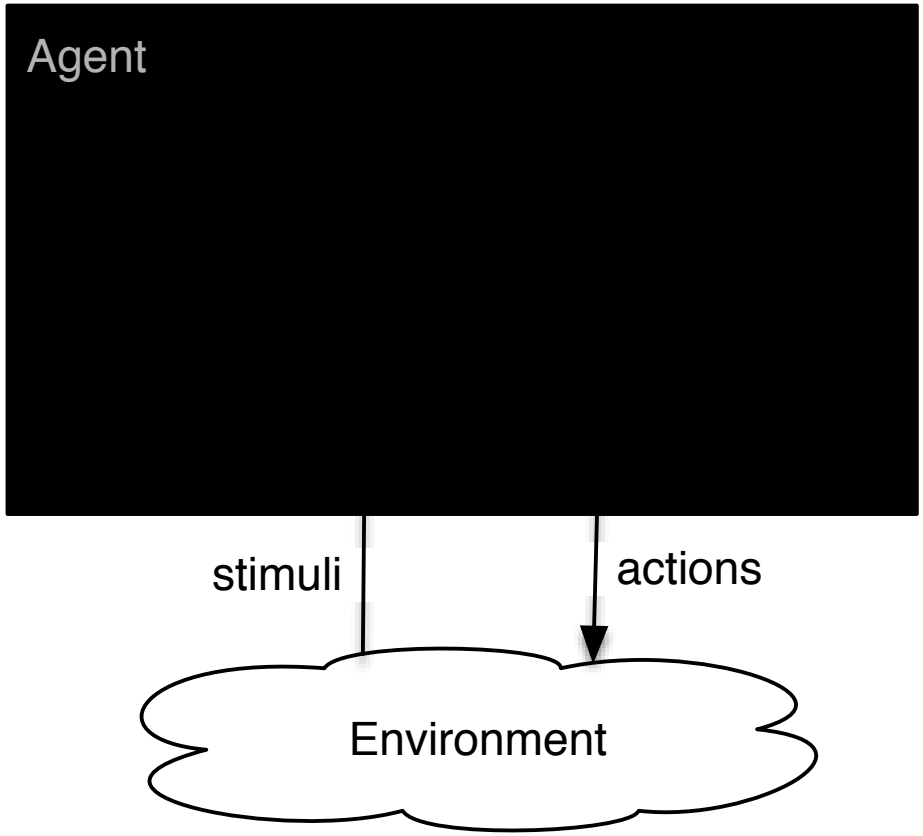
Overview:

- Agents and Robots
- Agent systems and architectures
- Agent controllers
- Hierarchical controllers

Example: smart house

- A smart house will monitor your use of essentials, and buy them before you run out.
  Example: snack buying agent that ensures you have a supply of chips:
  - abilities: buy chips (and have them delivered)
  - goals:
  - stimuli:
  - prior knowledge:

Agent Systems

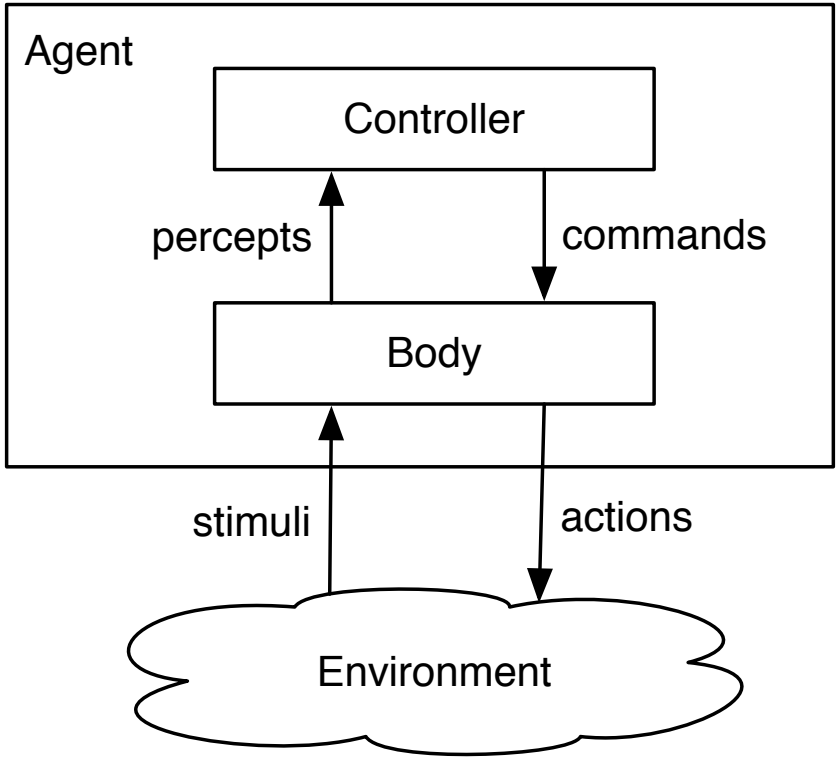


A agent system is made up of a agent and an environment.

- An agent receives stimuli from the environment
- An agent carries out actions in the environment.

Agent System Architecture

An agent is made up of a body and a controller.



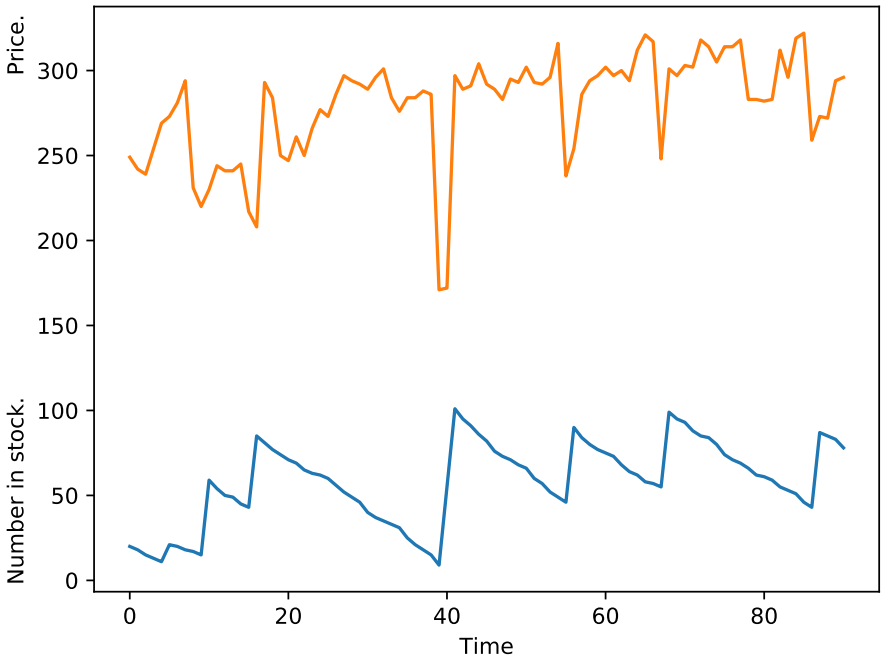
- An agent interacts with the environment through its body.
- The body is made up of:
  - sensors that interpret stimuli
  - actuators that carry out actions
- The controller receives percepts from the body.
- The controller sends commands to the body.
- The body can also have reactions that are not controlled.

Implementing a controller

- A controller is the brains of the agent.
- Agents are situated in time, they receive sensory data in time, and do actions in time.
- Controllers have (limited) memory and (limited) computational capabilities.
- The controller specifies the command at every time.
- The command at any time can depend on the current and previous percepts.

The Agent Functions

- A percept trace is a sequence of all past, present, and future percepts received by the controller.



- A command trace is a sequence of all past, present, and future commands output by the controller.

The Agent Functions

- A percept trace is a sequence of all past, present, and future percepts received by the controller.
- A command trace is a sequence of all past, present, and future commands output by the controller.

The Agent Functions

- A percept trace is a sequence of all past, present, and future percepts received by the controller.
- A command trace is a sequence of all past, present, and future commands output by the controller.
- A transduction is a function from percept traces into command traces.

The Agent Functions

- A percept trace is a sequence of all past, present, and future percepts received by the controller.
- A command trace is a sequence of all past, present, and future commands output by the controller.
- A transduction is a function from percept traces into command traces.
- A transduction is causal if the command trace up to time t depends only on percepts up to t.

The Agent Functions

- A percept trace is a sequence of all past, present, and future percepts received by the controller.
- A command trace is a sequence of all past, present, and future commands output by the controller.
- A transduction is a function from percept traces into command traces.
- A transduction is causal if the command trace up to time t depends only on percepts up to t.
- A controller is an implementation of a causal transduction.

The Agent Functions

- A percept trace is a sequence of all past, present, and future percepts received by the controller.
- A command trace is a sequence of all past, present, and future commands output by the controller.
- A transduction is a function from percept traces into command traces.
- A transduction is causal if the command trace up to time t depends only on percepts up to t.
- A controller is an implementation of a causal transduction.
- An agent's history at time t is sequence of past and present percepts and past commands.

The Agent Functions

- A percept trace is a sequence of all past, present, and future percepts received by the controller.
- A command trace is a sequence of all past, present, and future commands output by the controller.
- A transduction is a function from percept traces into command traces.
- A transduction is causal if the command trace up to time t depends only on percepts up to t.
- A controller is an implementation of a causal transduction.
- An agent's history at time t is sequence of past and present percepts and past commands.
- A causal transduction specifies a function from an agent's history at time t into its action at time t.

Belief States

- An agent doesn't have access to its entire history. It only has access to what it has remembered.

Belief States

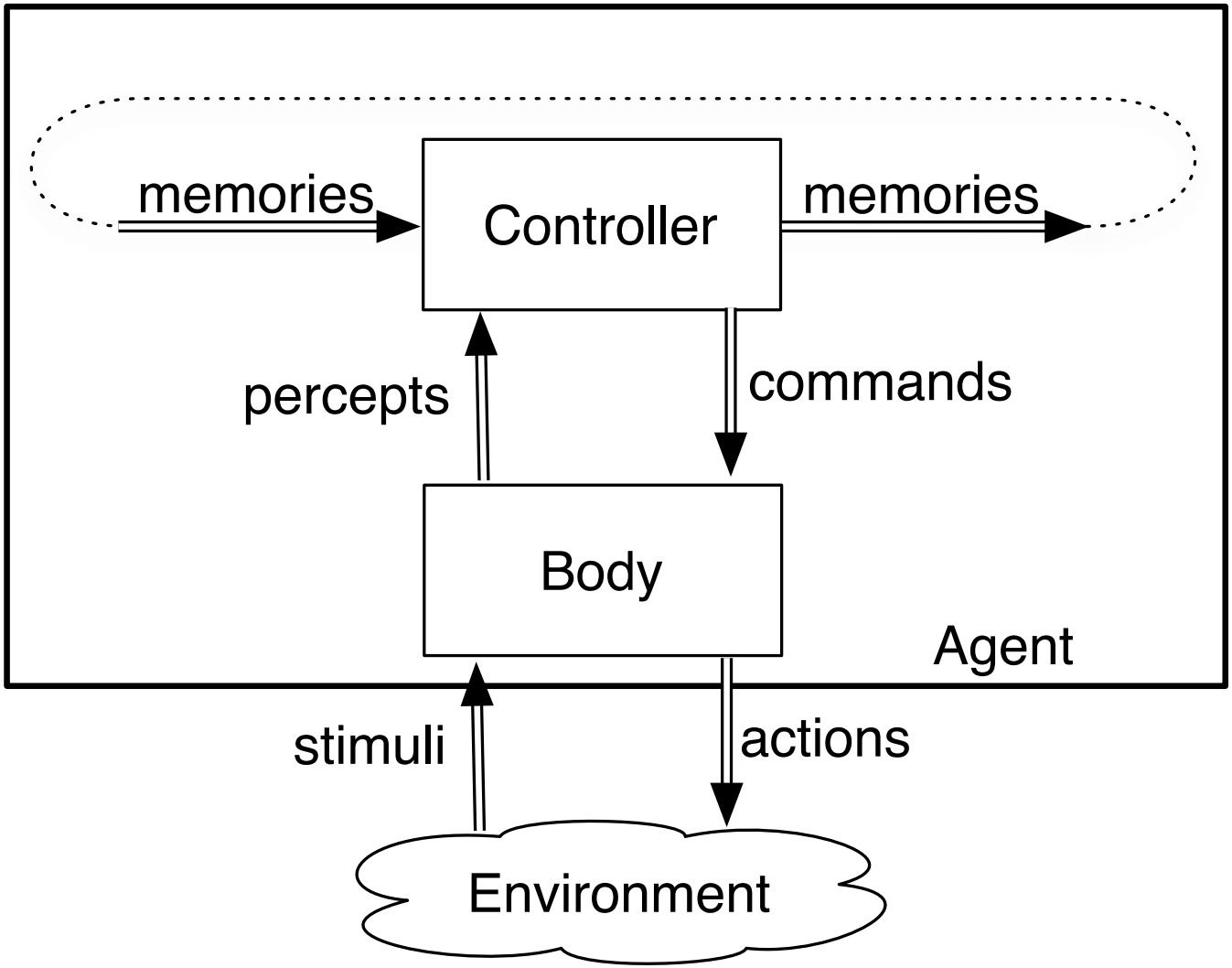
- An agent doesn't have access to its entire history. It only has access to what it has remembered.
- The memory or belief state of an agent at time t encodes all of the agent's history that it has access to.
- The belief state of an agent encapsulates the information about its past that it can use for current and future actions.

Belief States

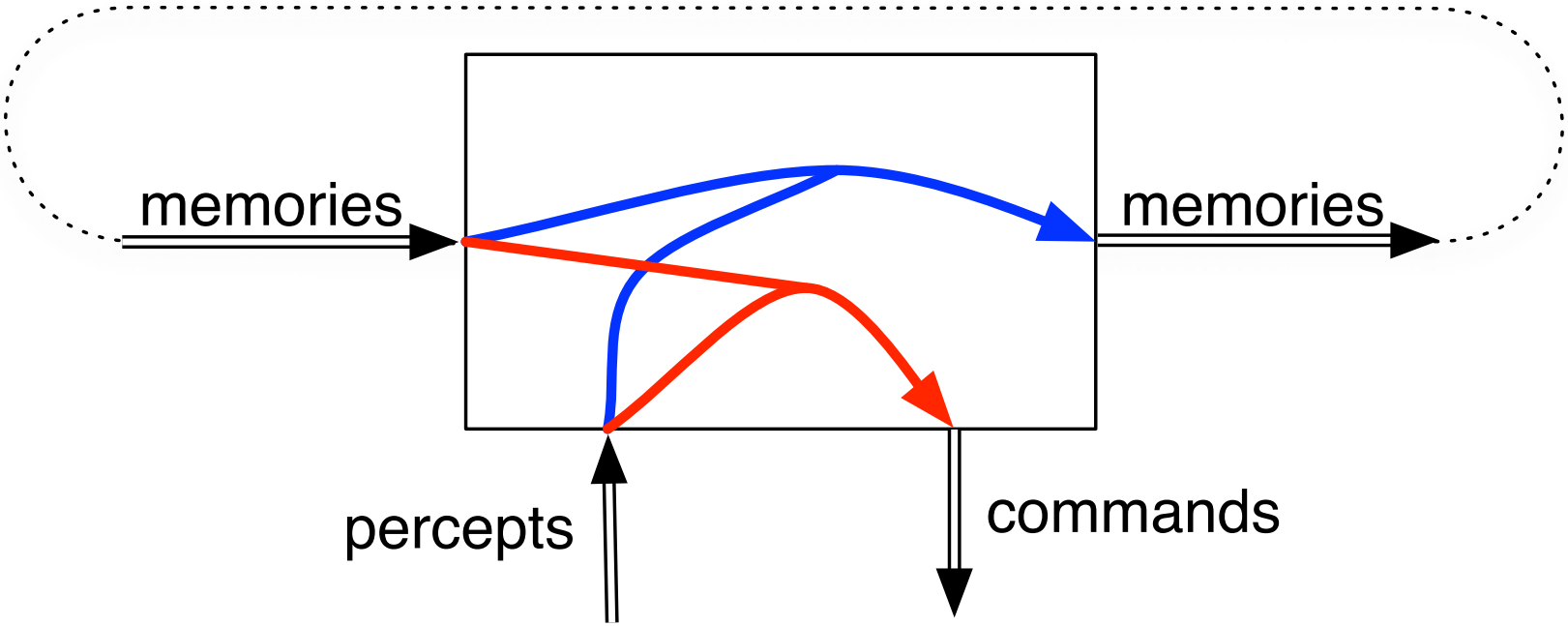
- An agent doesn't have access to its entire history. It only has access to what it has remembered.
- The memory or belief state of an agent at time t encodes all of the agent's history that it has access to.
- The belief state of an agent encapsulates the information about its past that it can use for current and future actions.
- At every time a controller has to decide on:
  - What should it do?
  - What should it remember?
    (How should it update its memory?)

  — as a function of its percepts and its memory.

Controller


memories
Controller
memories
percepts
commands
Body
Agent
stimuli
actions
Environment

Functions implemented in a controller



For discrete time, a controller implements:

- belief state function *remember*(*belief_state*, *percept*), returns the next belief state.
- command function *command*(*memory*, *percept*) returns the command for the agent.

Example: smart house

- A smart house will monitor your use of essentials, and buy them before you run out.
  Example: snack buying agent:
  - abilities: buy chips (and have them delivered)
  - goals: mimimize price, don't run out of chips
  - stimuli: price, number in stock
  - prior knowledge: ??
- Percept trace:
- Control trace:
- Transduction:
- Belief state:
- Belief state transition function:
- Control Function:

Implemented Example

- Percepts: price, number in stock
- Action: number to buy
- Belief state: (approximate) running average
- controller:
  - if $price < 0.9 * average$ and $instock < 60$ buy 48
  - else if $instock < 12$ buy 12
  - else buy 0

Implemented Example

- Percepts: price, number in stock
- Action: number to buy
- Belief state: (approximate) running average
- controller:
  - if $price < 0.9 * average$ and $instock < 60$ buy 48
  - else if $instock < 12$ buy 12
  - else buy 0
- Belief state transition function:

$$average \;:=\; average + (price - average) * 0.05$$

Implemented Example

- Percepts: price, number in stock
- Action: number to buy
- Belief state: (approximate) running average
- controller:
  - if $price < 0.9 * average$ and $instock < 60$ buy 48
  - else if $instock < 12$ buy 12
  - else buy 0
- Belief state transition function:

$$average \;:=\; average + (price - average) * 0.05$$

This maintains a discouning rolling avergage that (eventually) weights more recent prices more.

(see `agents.py` in AIPython distribution `http://aipython.org`)

By a hierarchic system, or hierarchy, I mean a system that is composed of interrelated subsystems, each of the latter being in turn hierarchic in structure until we reach some lowest level of elementary subsystem. In most systems of nature it is somewhat arbitrary as to where we leave off the partitioning and what subsystems we take as elementary. Physics makes much use of the concept of "elementary particle," although the particles have a disconcerting tendency not to remain elementary very long ...

Empirically a large proportion of the complex systems we observe in nature exhibit hierarchic structure. On theoretical grounds we would expect complex systems to be hierarchies in a world in which complexity had to evolve from simplicity.

– *Herbert A. Simon, 1996*

Agent Architectures

You don't need to implement an intelligent agent as:

as three independent modules, each feeding into the the next.

Agent Architectures

You don't need to implement an intelligent agent as:

as three independent modules, each feeding into the the next.

- It's too slow.

Agent Architectures

You don't need to implement an intelligent agent as:

as three independent modules, each feeding into the the next.

- It's too slow.
- High-level strategic reasoning takes more time than the reaction time needed (e.g. to avoid obstacles).

Agent Architectures

You don't need to implement an intelligent agent as:

as three independent modules, each feeding into the the next.

- It's too slow.
- High-level strategic reasoning takes more time than the reaction time needed (e.g. to avoid obstacles).
- The output of the perception depends on what you will do with it.

Hierarchical Control

- A better architecture is a hierarchy of controllers.
- Each controller sees the controllers below it as a virtual body from which it gets percepts and sends commands.

Hierarchical Control

- A better architecture is a hierarchy of controllers.
- Each controller sees the controllers below it as a virtual body from which it gets percepts and sends commands.
- The lower-level controllers can

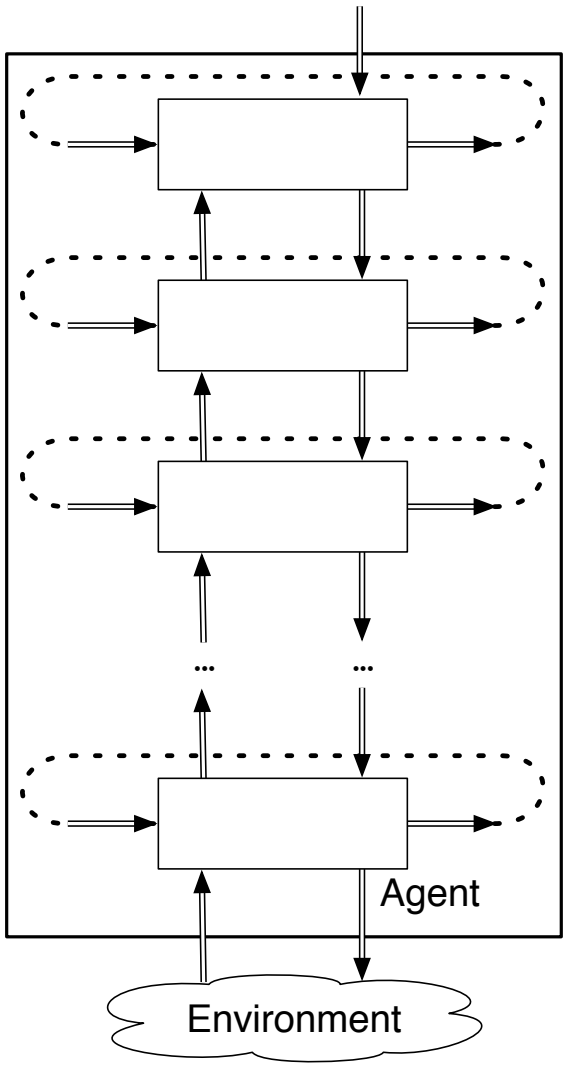
Hierarchical Control

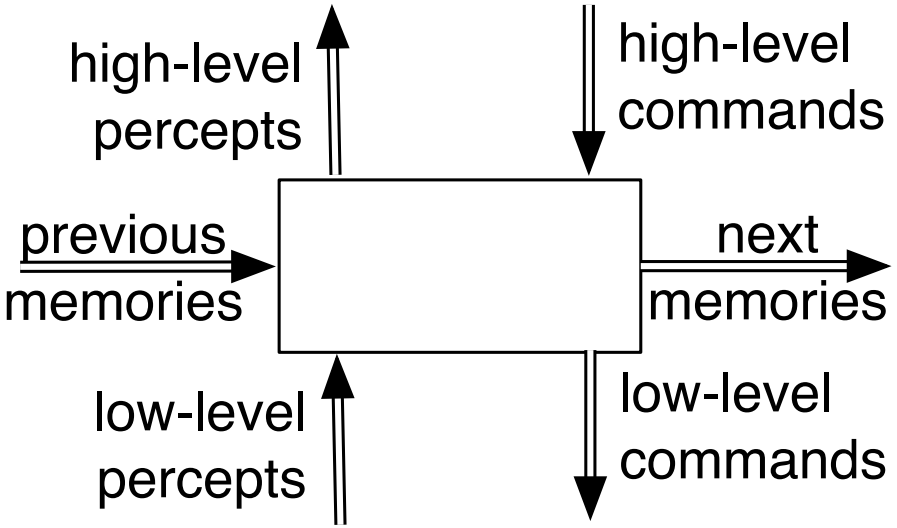
- A better architecture is a hierarchy of controllers.
- Each controller sees the controllers below it as a virtual body from which it gets percepts and sends commands.
- The lower-level controllers can
  - run much faster, and react to the world more quickly

Hierarchical Control

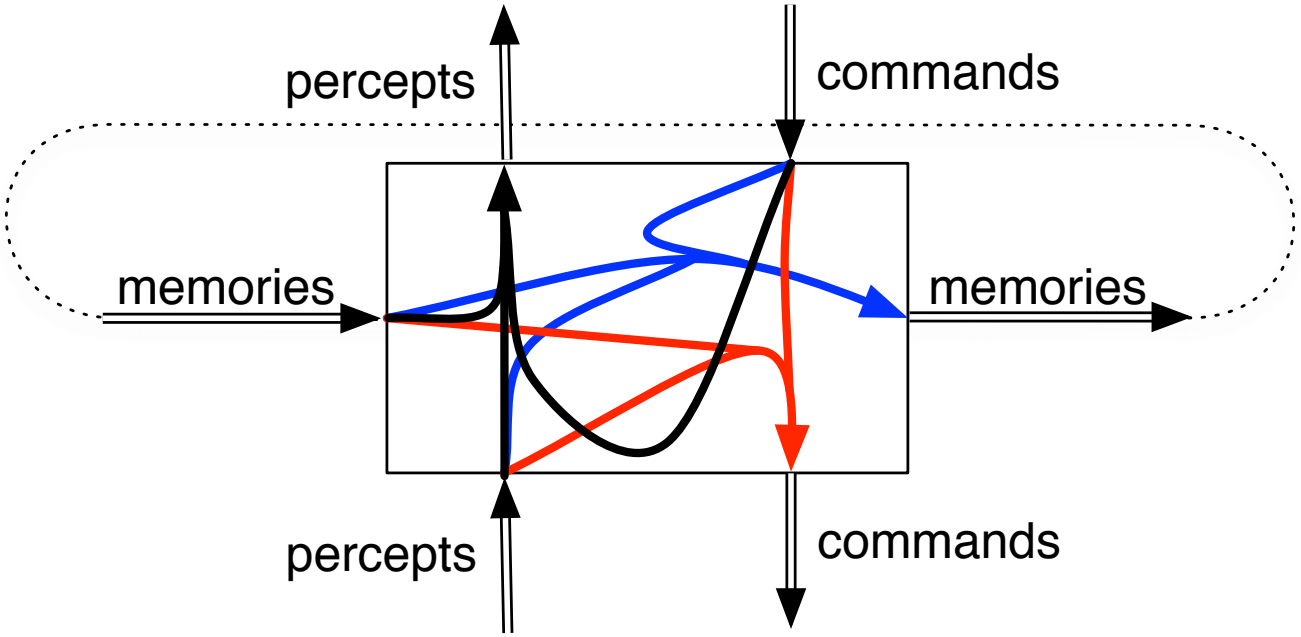
- A better architecture is a hierarchy of controllers.
- Each controller sees the controllers below it as a virtual body from which it gets percepts and sends commands.
- The lower-level controllers can
  - run much faster, and react to the world more quickly
  - deliver a simpler view of the world to the higher-level controllers.

Hierarchical Robotic System Architecture





Functions implemented in a layer



- memory function $remember(memory, percept, command)$
- command function
  $do(memory, percept, command)$
- percept function
  $higher_percept(memory, percept, command)$

Example: delivery robot

- The robot has three actions: go straight, go right, go left. (Its velocity doesn't change).

Example: delivery robot

- The robot has three actions: go straight, go right, go left. (Its velocity doesn't change).
- It can be given a plan consisting of sequence of named locations for the robot to go to in turn.

Example: delivery robot

- The robot has three actions: go straight, go right, go left. (Its velocity doesn't change).
- It can be given a plan consisting of sequence of named locations for the robot to go to in turn.
- The robot must avoid obstacles.

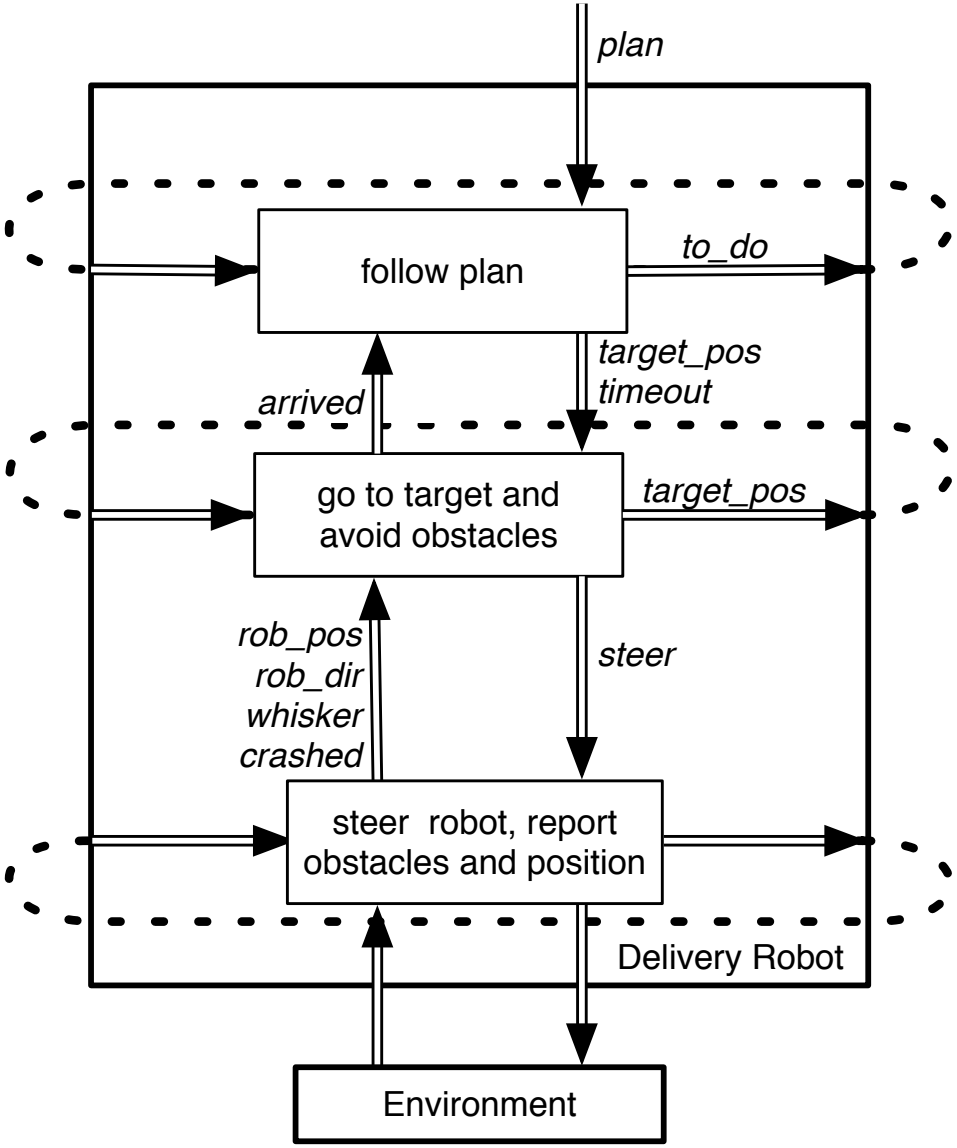
Example: delivery robot

- The robot has three actions: go straight, go right, go left. (Its velocity doesn't change).
- It can be given a plan consisting of sequence of named locations for the robot to go to in turn.
- The robot must avoid obstacles.
- It has a single whisker sensor pointing forward and to the right. The robot can detect if the whisker hits an object. The robot knows where it is.

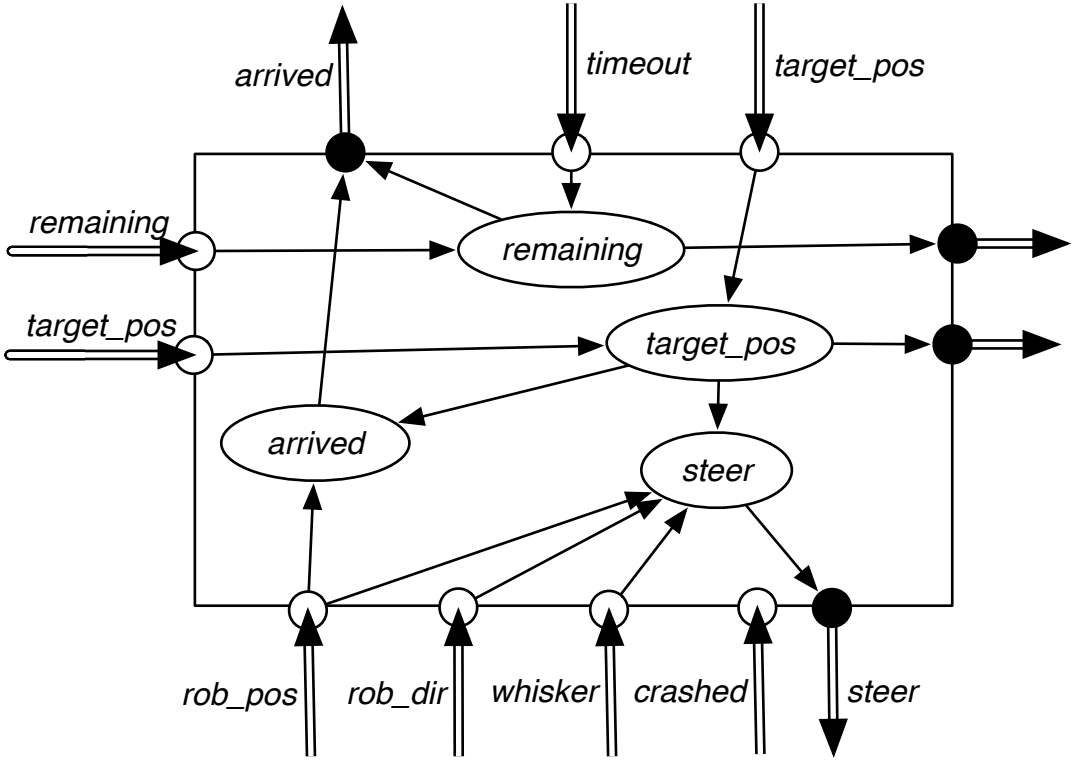
Example: delivery robot

- The robot has three actions: go straight, go right, go left. (Its velocity doesn't change).
- It can be given a plan consisting of sequence of named locations for the robot to go to in turn.
- The robot must avoid obstacles.
- It has a single whisker sensor pointing forward and to the right. The robot can detect if the whisker hits an object. The robot knows where it is.
- The obstacles and locations can be moved dynamically. Obstacles and new locations can be created dynamically.

A Decomposition of the Delivery Robot



Middle Layer



Middle Layer of the Delivery Robot

given *timeout* and *target_pos*:

  remaining := *timeout*

  while not *arrived*() and *remaining* $\neq 0$

    if *whisker_sensor* = *on*

      then *steer* := *left*

    else if *straight_ahead*(*rob_pos*, *robot_dir*, *target_pos*)

      then *steer* := *straight*

    else if *left_of*(*rob_pos*, *robot_dir*, *target_pos*)

      then *steer* := *left*

    else *steer* := *right*

    do(*steer*)

    remaining := *remaining* $- 1$

  tell upper layer *arrived*()

Top Layer of the Delivery Robot

- The top layer is given a plan which is a sequence of named locations.

Top Layer of the Delivery Robot

- The top layer is given a plan which is a sequence of named locations.
- The top layer tells the middle layer the goal position of the current location.

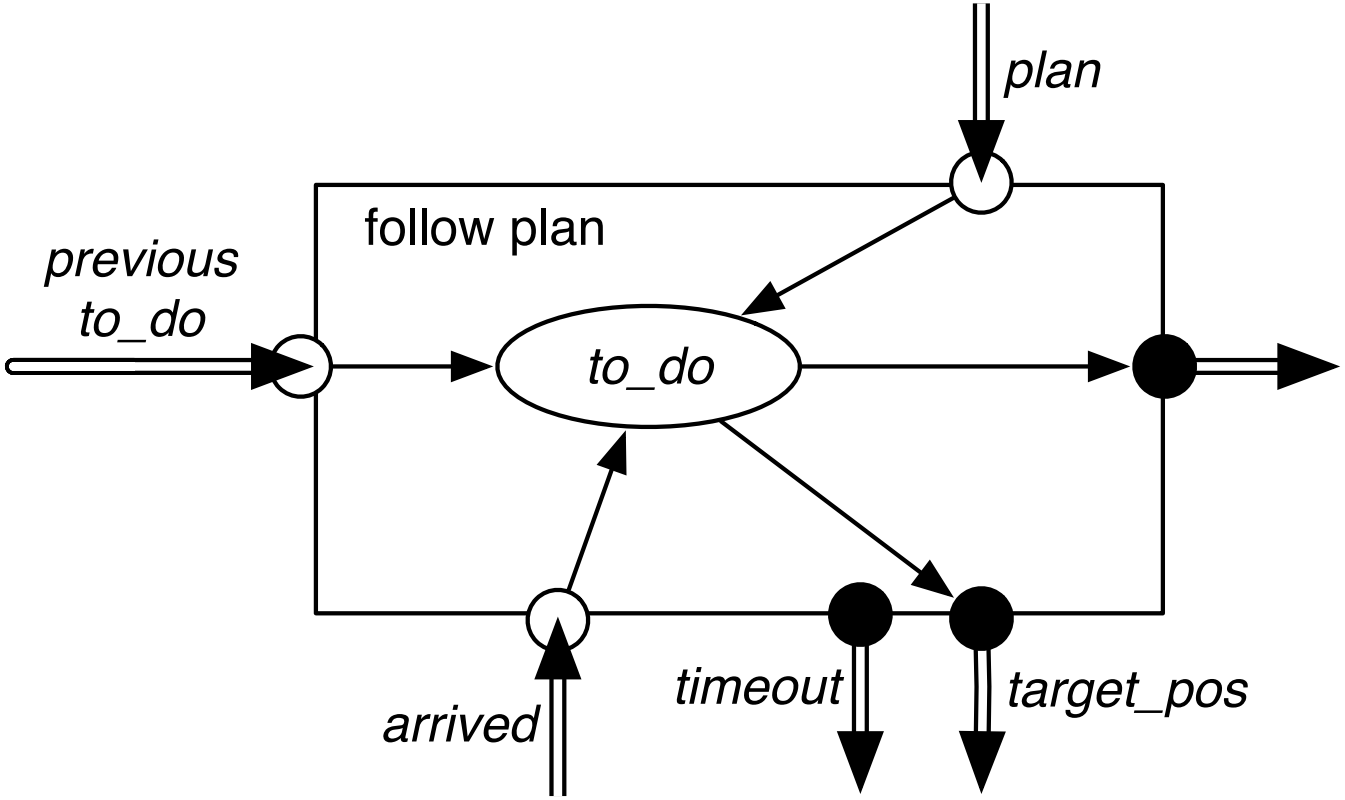
Top Layer of the Delivery Robot

- The top layer is given a plan which is a sequence of named locations.
- The top layer tells the middle layer the goal position of the current location.
- It has to remember the current goal position and the locations still to visit.

Top Layer of the Delivery Robot

- The top layer is given a plan which is a sequence of named locations.
- The top layer tells the middle layer the goal position of the current location.
- It has to remember the current goal position and the locations still to visit.
- When the middle layer reports the robot has arrived, the top layer takes the next location from the list of positions to visit, and there is a new goal position.

Top Layer

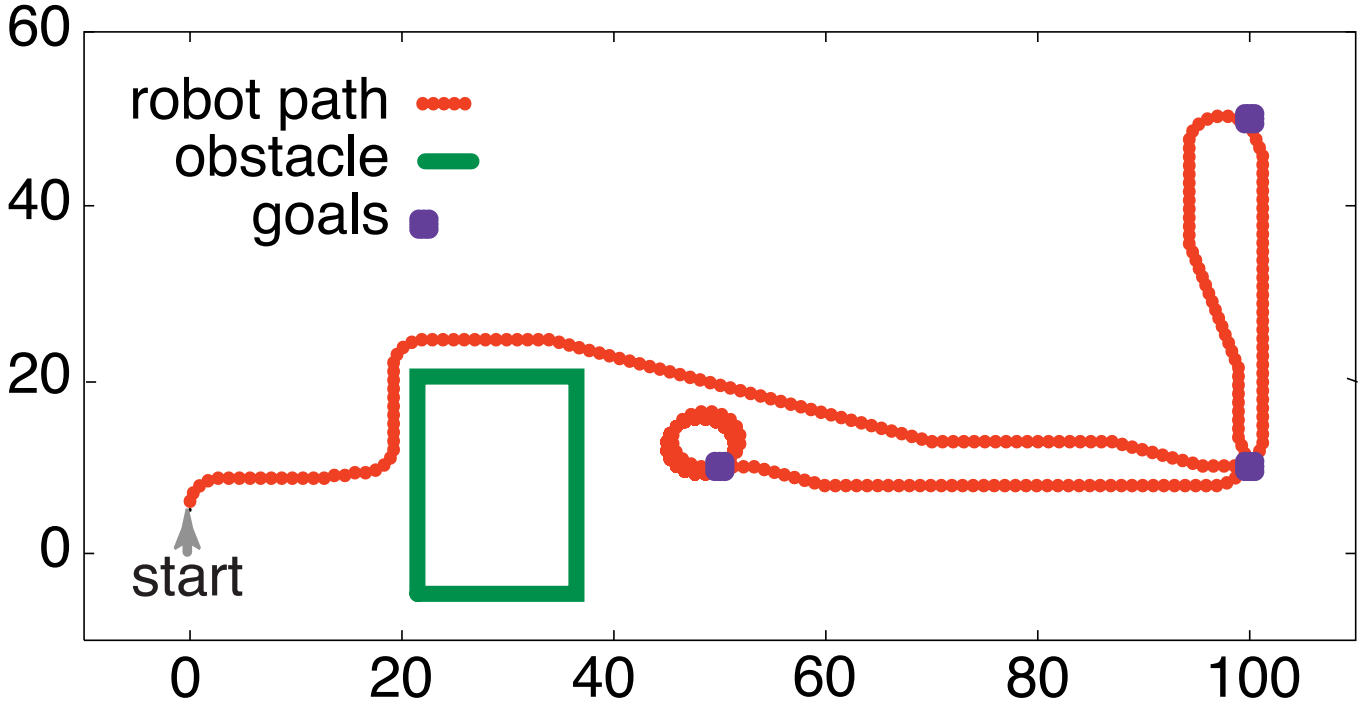

plan
follow plan
previous
to_do
to_do
arrived
timeout
target_pos

Code for the top layer

given *plan*:

$$
\begin{aligned}
&\quad to_do := plan \\
&\quad timeout := 200 \\
&\quad \text{while not } empty(to_do) \\
&\qquad\qquad target_pos := coordinates(first(to_do)) \\
&\qquad\qquad do(timeout, target_pos) \\
&\qquad\qquad to_do := rest(to_do)
\end{aligned}
$$

Simulation of the Robot



$$to_do = [goto(o109), goto(storage), goto(o109), goto(o103)]$$

$$arrived = true$$

What should be in an agent's belief state?

- An agent decides what to do based on its belief state and what it observes.

What should be in an agent's belief state?

- An agent decides what to do based on its belief state and what it observes.
- A purely reactive agent doesn't have a belief state.

What should be in an agent's belief state?

- An agent decides what to do based on its belief state and what it observes.
- A purely reactive agent doesn't have a belief state.
  A dead reckoning agent doesn't perceive the world.
  — neither work very well in complicated domains.

What should be in an agent's belief state?

- An agent decides what to do based on its belief state and what it observes.
- A purely reactive agent doesn't have a belief state.
  A dead reckoning agent doesn't perceive the world.
  — neither work very well in complicated domains.
- It is often useful for the agent's belief state to be a model of the world (itself and the environment).

Learning Objectives

At the end of the class you should be able to:

- define a directed graph
- represent a problem as a state-space graph

Searching

- Often we are not given an algorithm to solve a problem, but only a specification of what is a solution — we have to search for a solution.

Searching

- Often we are not given an algorithm to solve a problem, but only a specification of what is a solution — we have to search for a solution.
- A typical problem is when the agent is in one state, it has a set of deterministic actions it can carry out, and wants to get to a goal state.

Searching

- Often we are not given an algorithm to solve a problem, but only a specification of what is a solution — we have to search for a solution.
- A typical problem is when the agent is in one state, it has a set of deterministic actions it can carry out, and wants to get to a goal state.
- Many AI problems can be abstracted into the problem of finding a path in a directed graph.

Searching

- Often we are not given an algorithm to solve a problem, but only a specification of what is a solution — we have to search for a solution.
- A typical problem is when the agent is in one state, it has a set of deterministic actions it can carry out, and wants to get to a goal state.
- Many AI problems can be abstracted into the problem of finding a path in a directed graph.
- Often there is more than one way to represent a problem as a graph.

State-space Search

- flat or modular or hierarchical
- explicit states or features or individuals and relations
- static or finite stage or indefinite stage or infinite stage
- fully observable or partially observable
- deterministic or stochastic dynamics
- goals or complex preferences
- single agent or multiple agents
- knowledge is given or knowledge is learned
- perfect rationality or bounded rationality

State-space Problem

A state-space problem consists of

- a set of states
- a subset of states called the start states

State-space Problem

A state-space problem consists of

- a set of states
- a subset of states called the start states
- a set of actions

State-space Problem

A state-space problem consists of

- a set of states
- a subset of states called the start states
- a set of actions
- an action function: given a state and an action, returns a new state

State-space Problem

A state-space problem consists of

- a set of states
- a subset of states called the start states
- a set of actions
- an action function: given a state and an action, returns a new state
- a set of goal states, specified as function, $goal(s)$

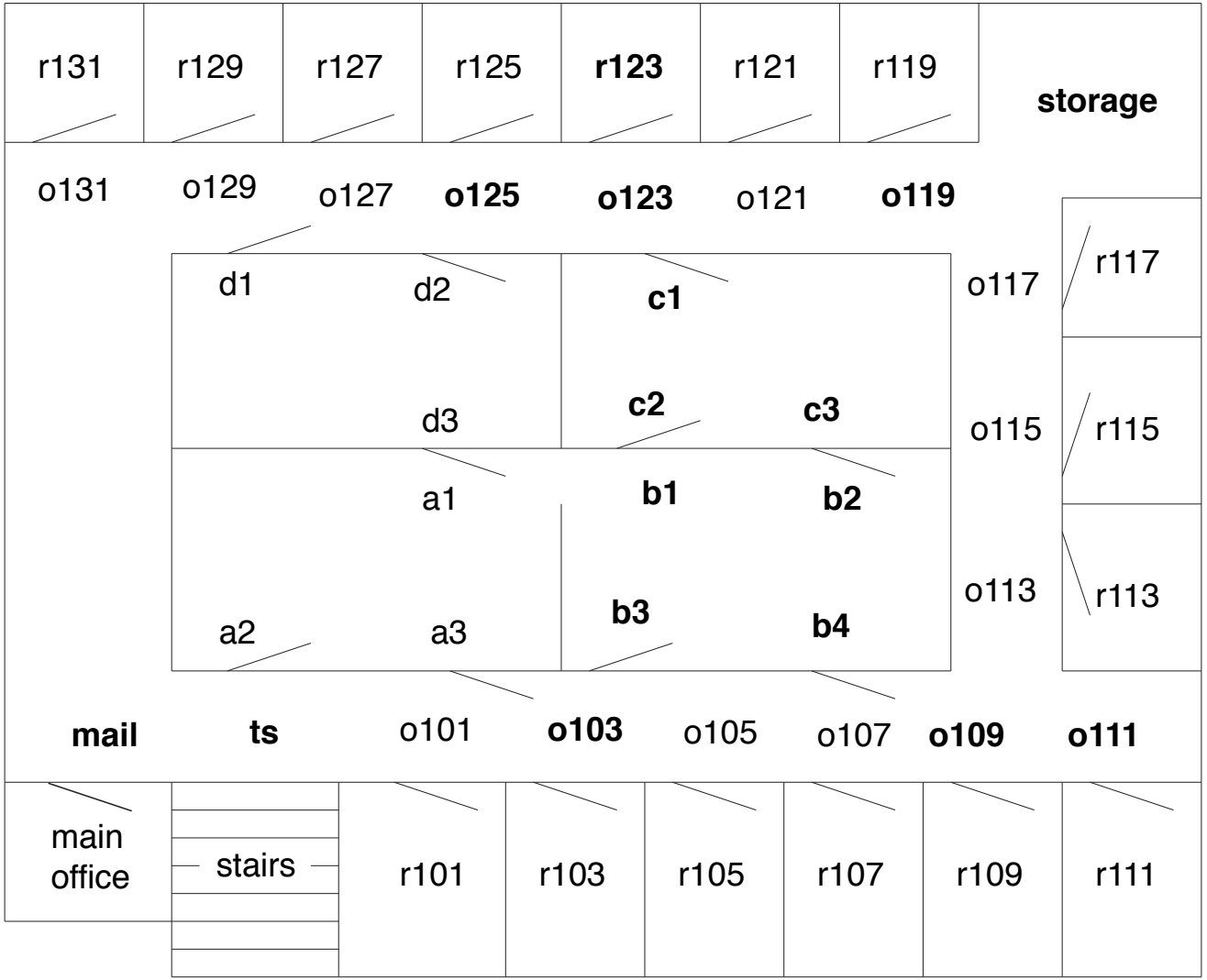
State-space Problem

A state-space problem consists of

- a set of states
- a subset of states called the start states
- a set of actions
- an action function: given a state and an action, returns a new state
- a set of goal states, specified as function, $goal(s)$
- a criterion that specifies the quality of an acceptable solution.

Example Problem for Delivery Robot

The robot wants to get from outside room 103 to the inside of room 123.



Directed Graphs

- A (directed) graph consists of a set N of nodes and a set A of ordered pairs of nodes, called arcs.

Directed Graphs

- A (directed) graph consists of a set N of nodes and a set A of ordered pairs of nodes, called arcs.
- Node n_2 is a neighbor of n_1 if there is an arc from n_1 to n_2. That is, if $\langle n_1, n_2 \rangle \in A$.

Directed Graphs

- A (directed) graph consists of a set N of nodes and a set A of ordered pairs of nodes, called arcs.
- Node n_2 is a neighbor of n_1 if there is an arc from n_1 to n_2. That is, if $\langle n_1, n_2 \rangle \in A$.
- A path is a sequence of nodes $\langle n_0, n_1, \ldots, n_k \rangle$ such that $\langle n_{i-1}, n_i \rangle \in A$.

Directed Graphs

- A (directed) graph consists of a set N of nodes and a set A of ordered pairs of nodes, called arcs.
- Node n_2 is a neighbor of n_1 if there is an arc from n_1 to n_2. That is, if $\langle n_1, n_2 \rangle \in A$.
- A path is a sequence of nodes $\langle n_0, n_1, \ldots, n_k \rangle$ such that $\langle n_{i-1}, n_i \rangle \in A$.
- Given start nodes and goal nodes, a solution is a path from a start node to a goal node.

Directed Graphs

- A (directed) graph consists of a set N of nodes and a set A of ordered pairs of nodes, called arcs.
- Node n_2 is a neighbor of n_1 if there is an arc from n_1 to n_2. That is, if $\langle n_1, n_2 \rangle \in A$.
- A path is a sequence of nodes $\langle n_0, n_1, \ldots, n_k \rangle$ such that $\langle n_{i-1}, n_i \rangle \in A$.
- Given start nodes and goal nodes, a solution is a path from a start node to a goal node.
- When there is a cost associated with arcs, the cost of a path is the sum of the costs of the arcs in the path:

$$cost(\langle n_0, n_1, \ldots, n_k \rangle) = \sum_{i=1}^{k} cost(\langle n_{i-1}, n_i \rangle)$$

An optimal solution is one with minimum cost.

What is a state?

What is a state?

A state needs to include enough information to

- determine what is the next state

What is a state?

A state needs to include enough information to

- determine what is the next state
- determine whether the goal is achieved

What is a state?

A state needs to include enough information to

- determine what is the next state
- determine whether the goal is achieved
- determine the cost.

What is a state?

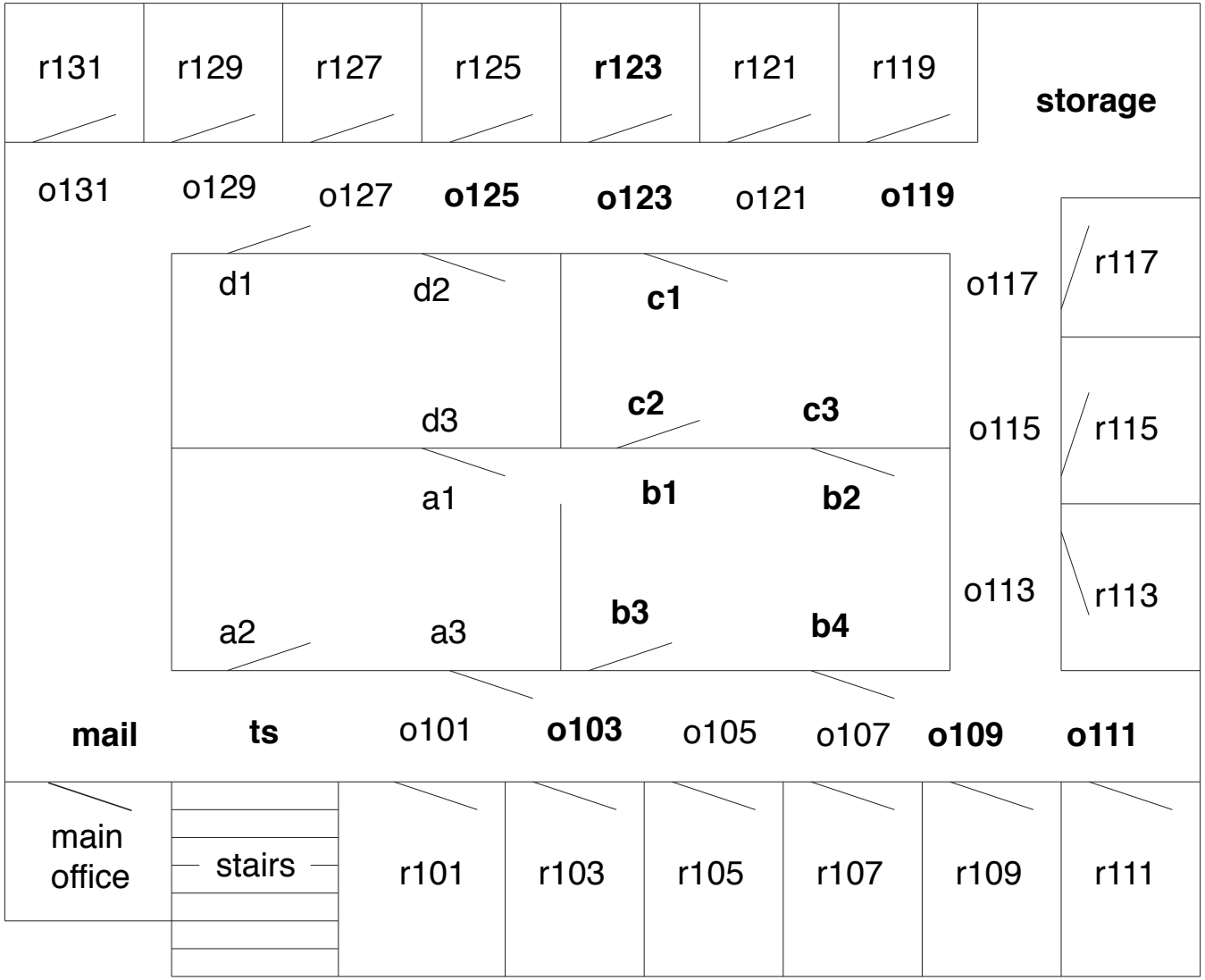
A state needs to include enough information to

- determine what is the next state
- determine whether the goal is achieved
- determine the cost.

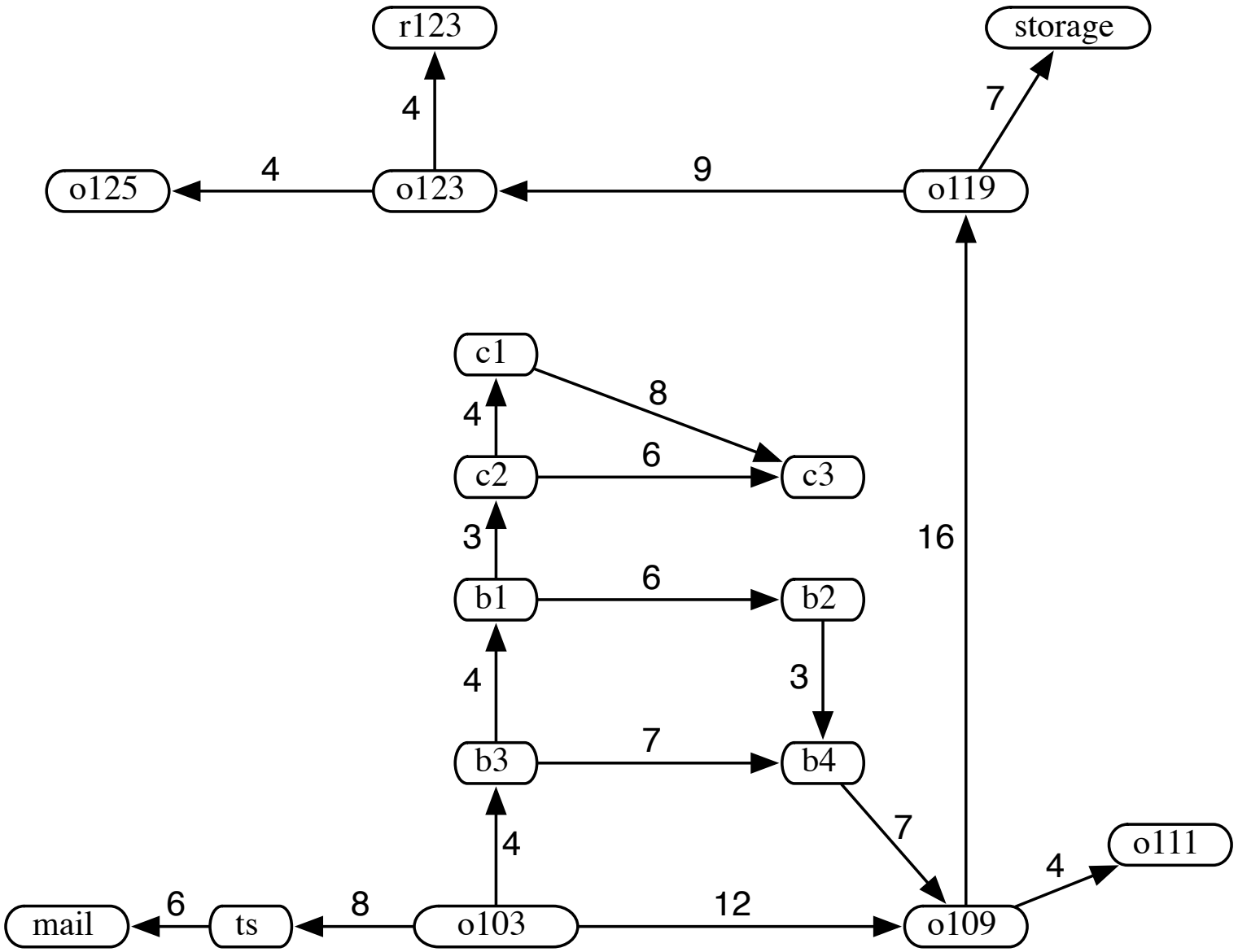
Often there are many options for what to include in the state.
Keep the states as simple as possible but no simpler.

Example Problem for Delivery Robot

The robot wants to get from outside room 103 to the inside of room 123.



State-Space Graph for the Delivery Robot



Example: Google Maps

Example: Google Maps

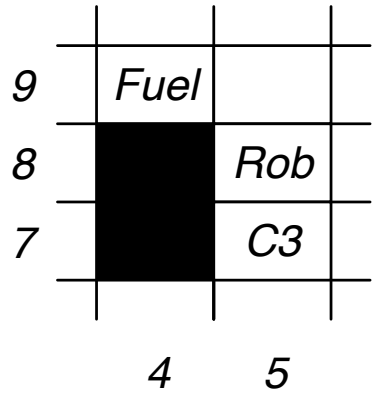
- single start location and goal location
- cost is estimated time
- state

Example: Google Maps

- single start location and goal location
- cost is estimated time
- state needs to include direction because the cost depends on directions (e.g., turning left).

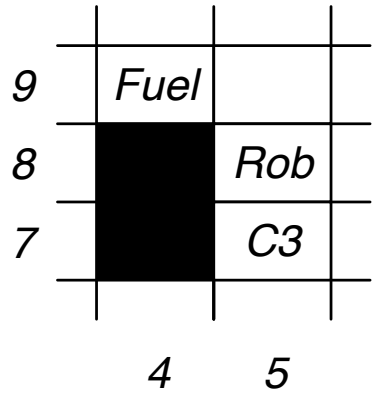
Partial Search Space for a Video Game

Grid game: Rob is on a grid and can move up, down, left or right and needs to collect coins C_1, C_2, C_3, C_4, without running out of fuel, and end up at location $(1, 1)$:



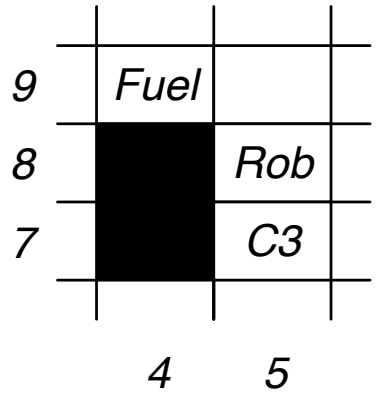
Partial Search Space for a Video Game

Grid game: Rob is on a grid and can move up, down, left or right and needs to collect coins C_1, C_2, C_3, C_4, without running out of fuel, and end up at location $(1, 1)$:
State:



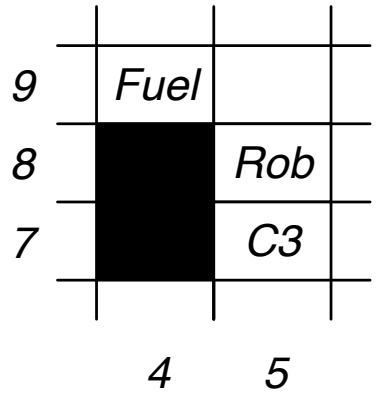
Partial Search Space for a Video Game

Grid game: Rob is on a grid and can move up, down, left or right and needs to collect coins C_1, C_2, C_3, C_4, without running out of fuel, and end up at location $(1, 1)$:
State: $< X\text{-}pos,\ Y\text{-}pos,$



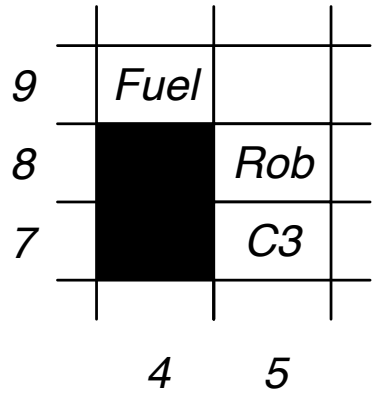
Partial Search Space for a Video Game

Grid game: Rob is on a grid and can move up, down, left or right and needs to collect coins C_1, C_2, C_3, C_4, without running out of fuel, and end up at location $(1, 1)$:
State: $< X\text{-}pos,\ Y\text{-}pos,\ \ Fuel,$



Partial Search Space for a Video Game

Grid game: Rob is on a grid and can move up, down, left or right and needs to collect coins C_1, C_2, C_3, C_4, without running out of fuel, and end up at location $(1, 1)$:
State: $< X\text{-}pos,\ Y\text{-}pos,\ \ Fuel, C_1,\ C_2,\ C_3,\ C_4 >$

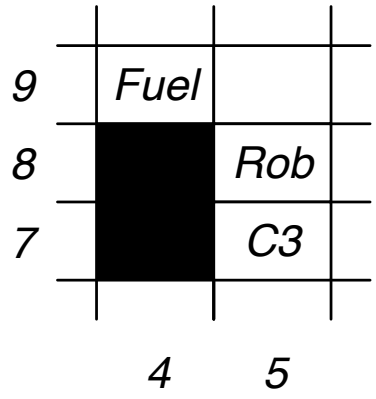


Partial Search Space for a Video Game

Grid game: Rob is on a grid and can move up, down, left or right and needs to collect coins C_1, C_2, C_3, C_4, without running out of fuel, and end up at location $(1, 1)$:

State: $< X\text{-}pos,\ Y\text{-}pos,\ \ Fuel, C_1,\ C_2,\ C_3,\ C_4 >$

Goal:

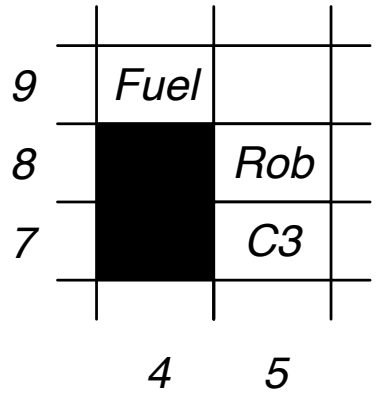


Partial Search Space for a Video Game

Grid game: Rob is on a grid and can move up, down, left or right and needs to collect coins C_1, C_2, C_3, C_4, without running out of fuel, and end up at location $(1, 1)$:

State: $< X\text{-}pos,\ Y\text{-}pos,\ \ Fuel, C_1,\ C_2,\ C_3,\ C_4 >$

Goal: $< 1,\ 1,\ ?,\ \mathrm{t},\ \mathrm{t},\ \mathrm{t},\ \mathrm{t} >$

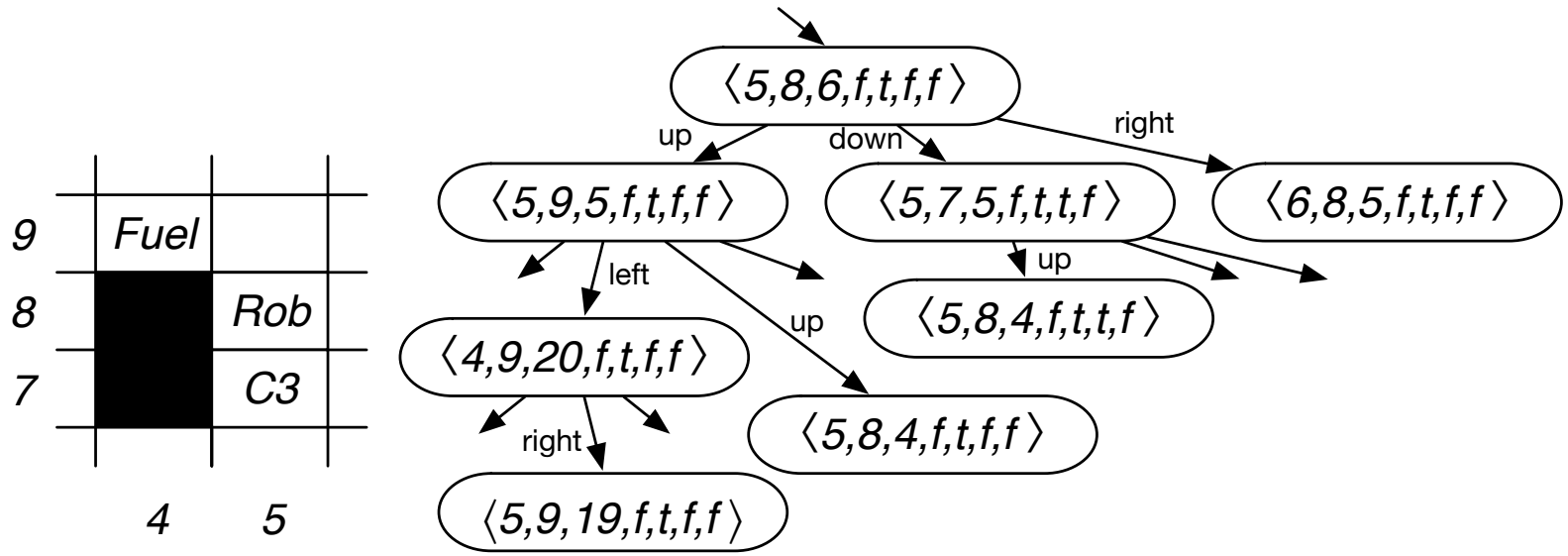


Partial Search Space for a Video Game

Grid game: Rob is on a grid and can move up, down, left or right and needs to collect coins C_1, C_2, C_3, C_4, without running out of fuel, and end up at location $(1, 1)$:

State: $< X\text{-}pos,\ Y\text{-}pos,\ \ Fuel, C_1,\ C_2,\ C_3,\ C_4 >$

Goal: $< 1,\ 1,\ ?,\ \mathrm{t},\ \mathrm{t},\ \mathrm{t},\ \mathrm{t} >$



Robot Cleaner

- 2 rooms, one cleaning robot
- rooms can be clean or dirty
- robot actions:
  suck: makes the room that the robot is in clean
  move: move to other room
- Goal: have both rooms clean

Robot Cleaner

- 2 rooms, one cleaning robot
- rooms can be clean or dirty
- robot actions:
  suck: makes the room that the robot is in clean
  move: move to other room
- Goal: have both rooms clean
- How many states are there? What are they?

Learning Objectives

At the end of the class you should be able to:

- explain how a generic searching algorithm works
- demonstrate how depth-first search will work on a graph
- demonstrate how breadth-first search will work on a graph
- predict the space and time requirements for depth-first and breadth-first searches

Graph Searching

- Generic search algorithm: given a graph, start nodes, and goal nodes, incrementally explore paths from the start nodes.

Graph Searching

- Generic search algorithm: given a graph, start nodes, and goal nodes, incrementally explore paths from the start nodes.
- Maintain a frontier of paths from the start node that have been explored.

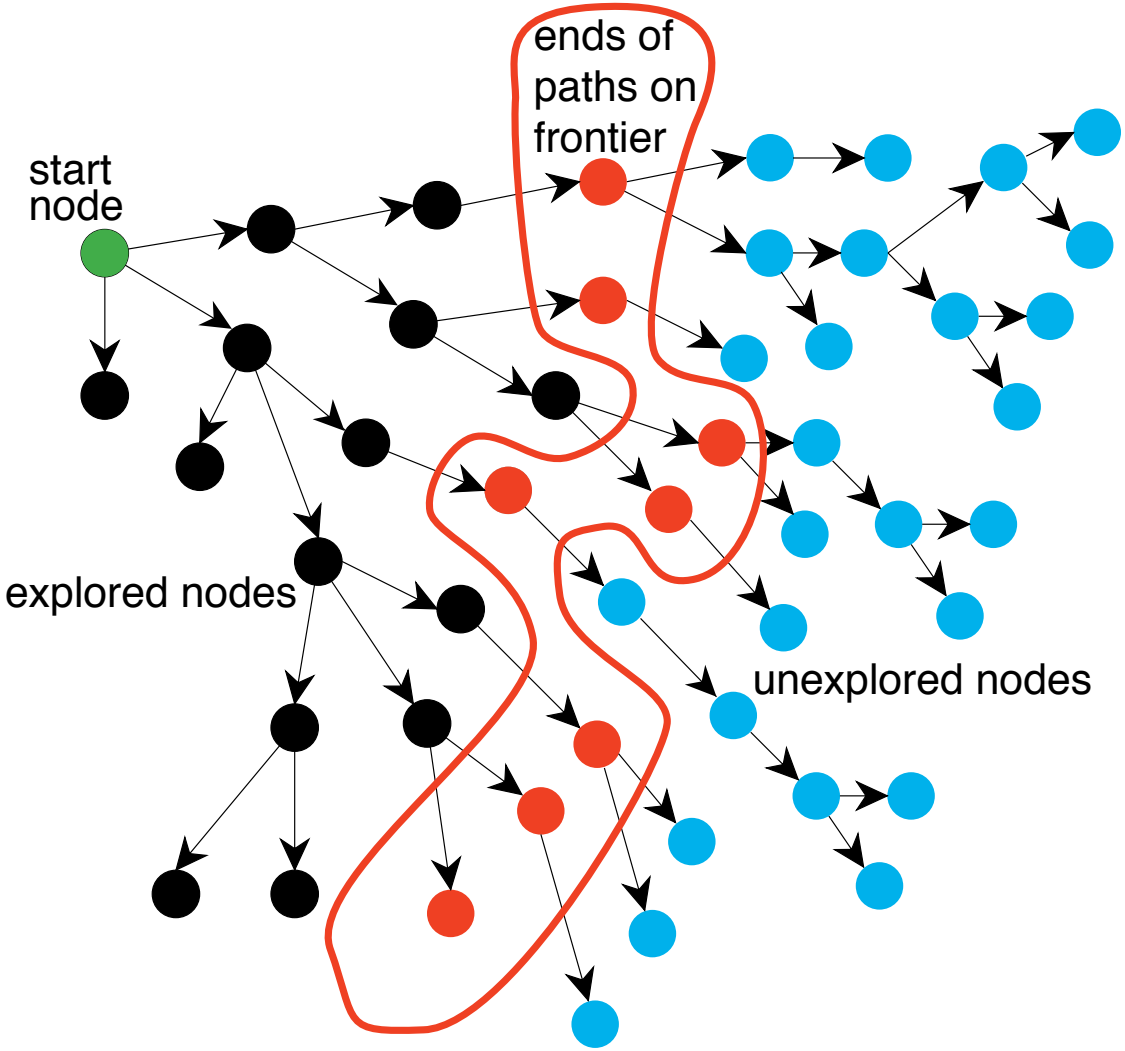
Graph Searching

- Generic search algorithm: given a graph, start nodes, and goal nodes, incrementally explore paths from the start nodes.
- Maintain a frontier of paths from the start node that have been explored.
- As search proceeds, the frontier expands into the unexplored nodes until a goal node is encountered.

Graph Searching

- Generic search algorithm: given a graph, start nodes, and goal nodes, incrementally explore paths from the start nodes.
- Maintain a frontier of paths from the start node that have been explored.
- As search proceeds, the frontier expands into the unexplored nodes until a goal node is encountered.
- The way in which the frontier is expanded defines the search strategy.

Problem Solving by Graph Searching



Graph Search Algorithm

Input: a graph,
a set of start nodes,
Boolean procedure $goal(n)$ that tests if n is a goal node.

$frontier := \{\langle s \rangle : s \text{ is a start node}\}$
while *frontier* is not empty:
  select and **remove** path $\langle n_0, \ldots, n_k \rangle$ from *frontier*
  if $goal(n_k)$
    return $\langle n_0, \ldots, n_k \rangle$
  for every neighbor n of n_k
    add $\langle n_0, \ldots, n_k, n \rangle$ to *frontier*
end while

Graph Search Algorithm

- Which value is selected from the frontier at each stage defines the search strategy.
- The neighbors define the graph.
- *goal* defines what is a solution.
- If more than one answer is required, the search can continue from the return.

Optimality Criteria

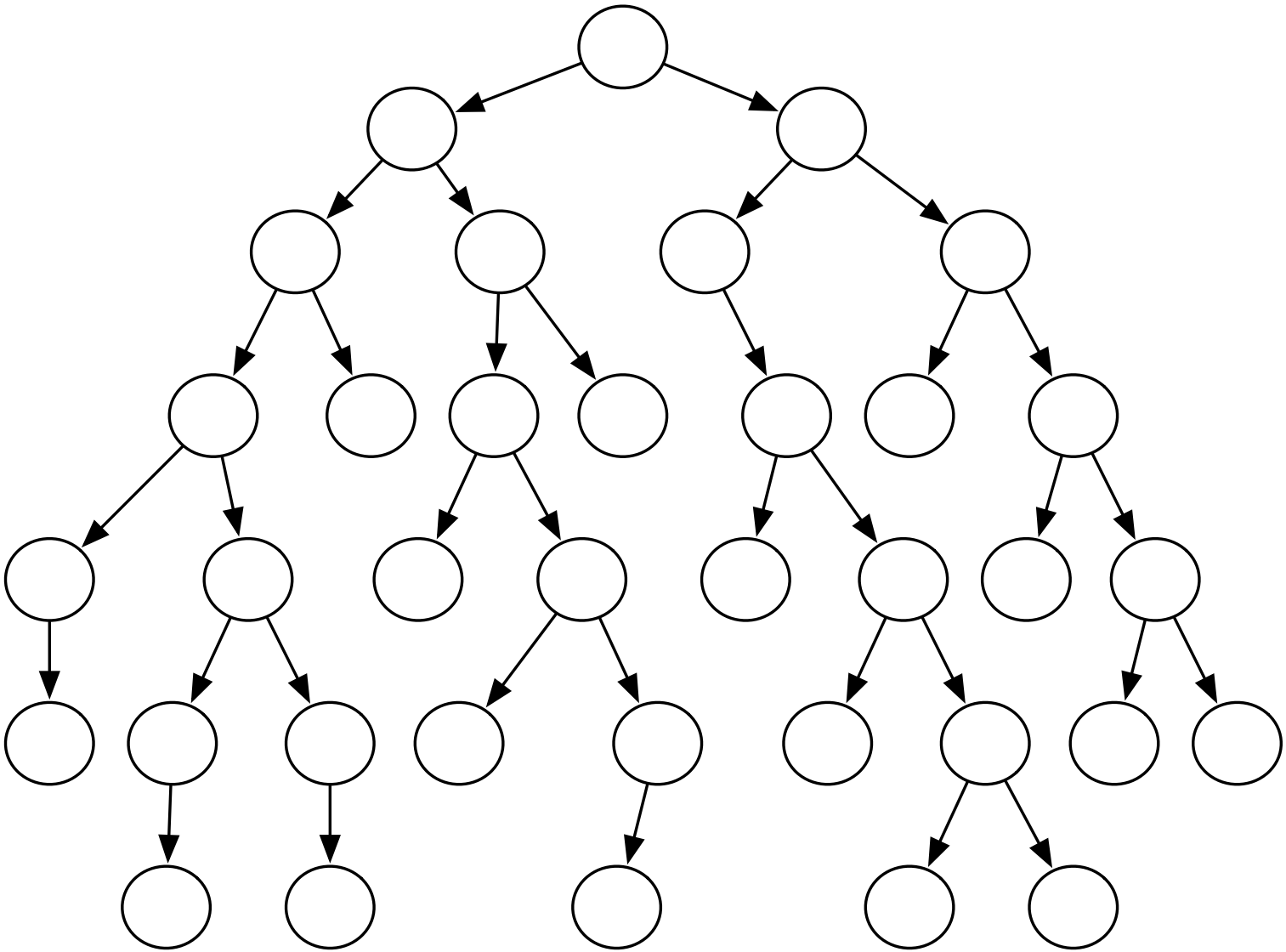
- Often we don't want any solution, but the best solution or optimal solution.
- Costs on arcs give costs on paths. We want the least-cost path to a goal.

Depth-first Search

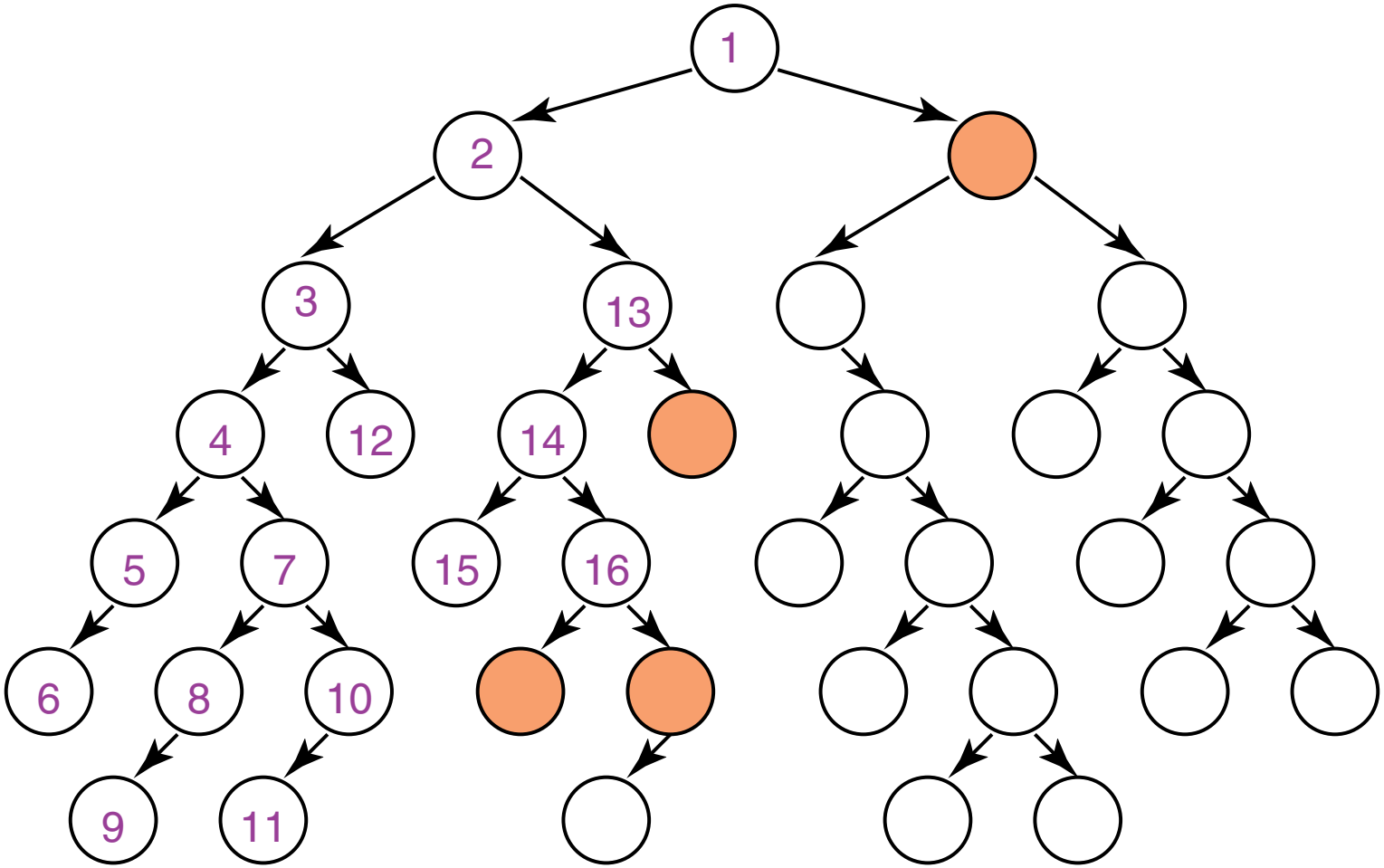
- Depth-first search treats the frontier as a stack
- It always selects one of the last elements added to the frontier.
- If the list of paths on the frontier is $[p_1, p_2, \ldots]$
  - p_1 is selected. Paths that extend p_1 are added to the front of the stack (in front of p_2).
  - p_2 is only selected when all paths from p_1 have been explored.

Illustrative Graph - Depth-first search

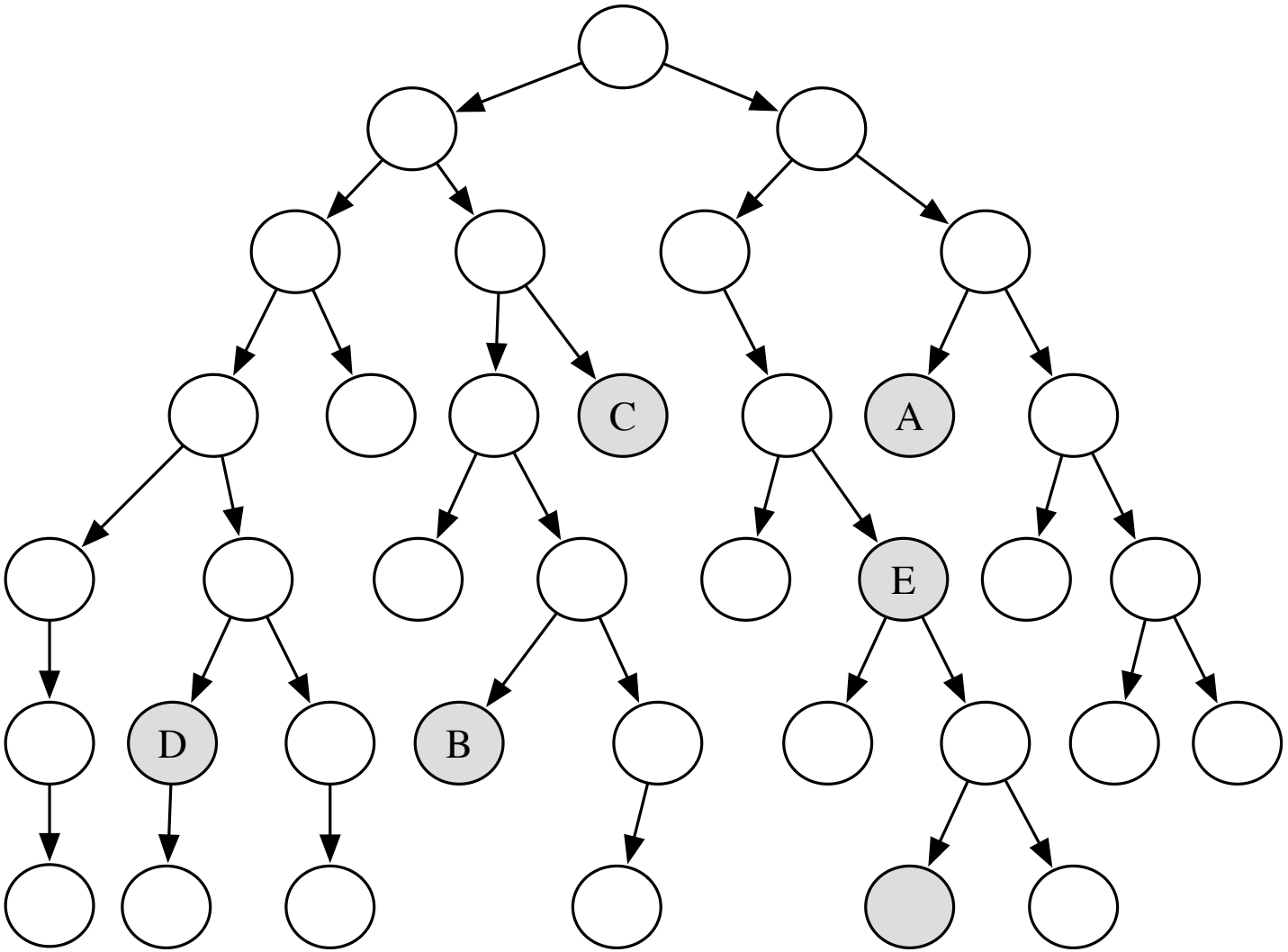
Start node is at the top.



Illustrative Graph — Depth-first Search



Which shaded goal will depth-first search find first?



Complexity of Depth-first Search

- Does depth-first search guarantee to find the path with fewest arcs?

Complexity of Depth-first Search

- Does depth-first search guarantee to find the path with fewest arcs?
- What happens on infinite graphs or on graphs with cycles if there is a solution?

Complexity of Depth-first Search

- Does depth-first search guarantee to find the path with fewest arcs?
- What happens on infinite graphs or on graphs with cycles if there is a solution?
- What is the time complexity as a function of length of the path selected?

Complexity of Depth-first Search

- Does depth-first search guarantee to find the path with fewest arcs?
- What happens on infinite graphs or on graphs with cycles if there is a solution?
- What is the time complexity as a function of length of the path selected?
- What is the space complexity as a function of length of the path selected?

Complexity of Depth-first Search

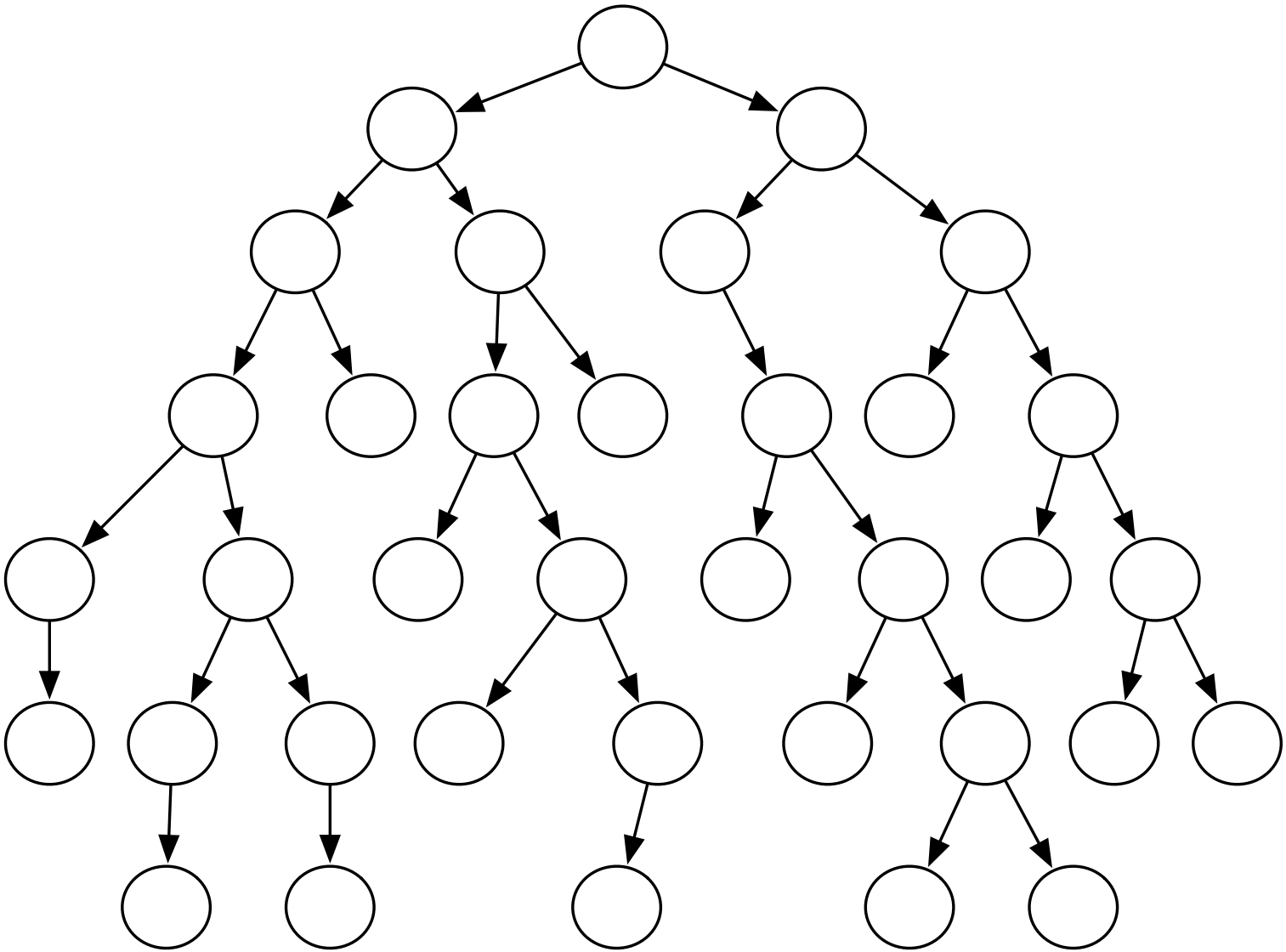
- Does depth-first search guarantee to find the path with fewest arcs?
- What happens on infinite graphs or on graphs with cycles if there is a solution?
- What is the time complexity as a function of length of the path selected?
- What is the space complexity as a function of length of the path selected?
- How does the goal affect the search?

Breadth-first Search

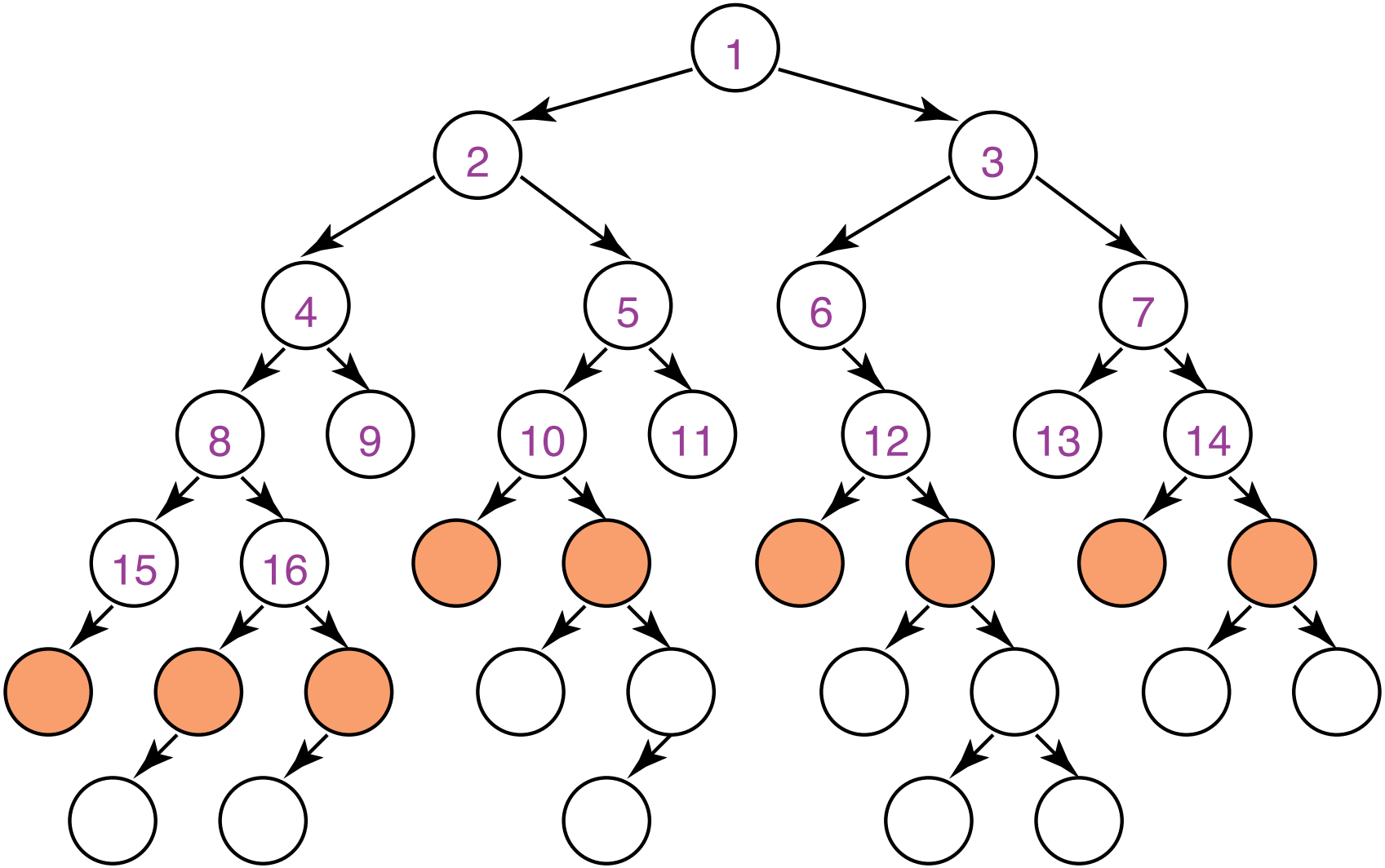
- Breadth-first search treats the frontier as a queue.
- It always selects one of the earliest elements added to the frontier.
- If the list of paths on the frontier is $[p_1, p_2, \ldots, p_r]$:
  - p_1 is selected. Its neighbors are added to the end of the queue, after p_r.
  - p_2 is selected next.

Illustrative Graph - Breadth-first search

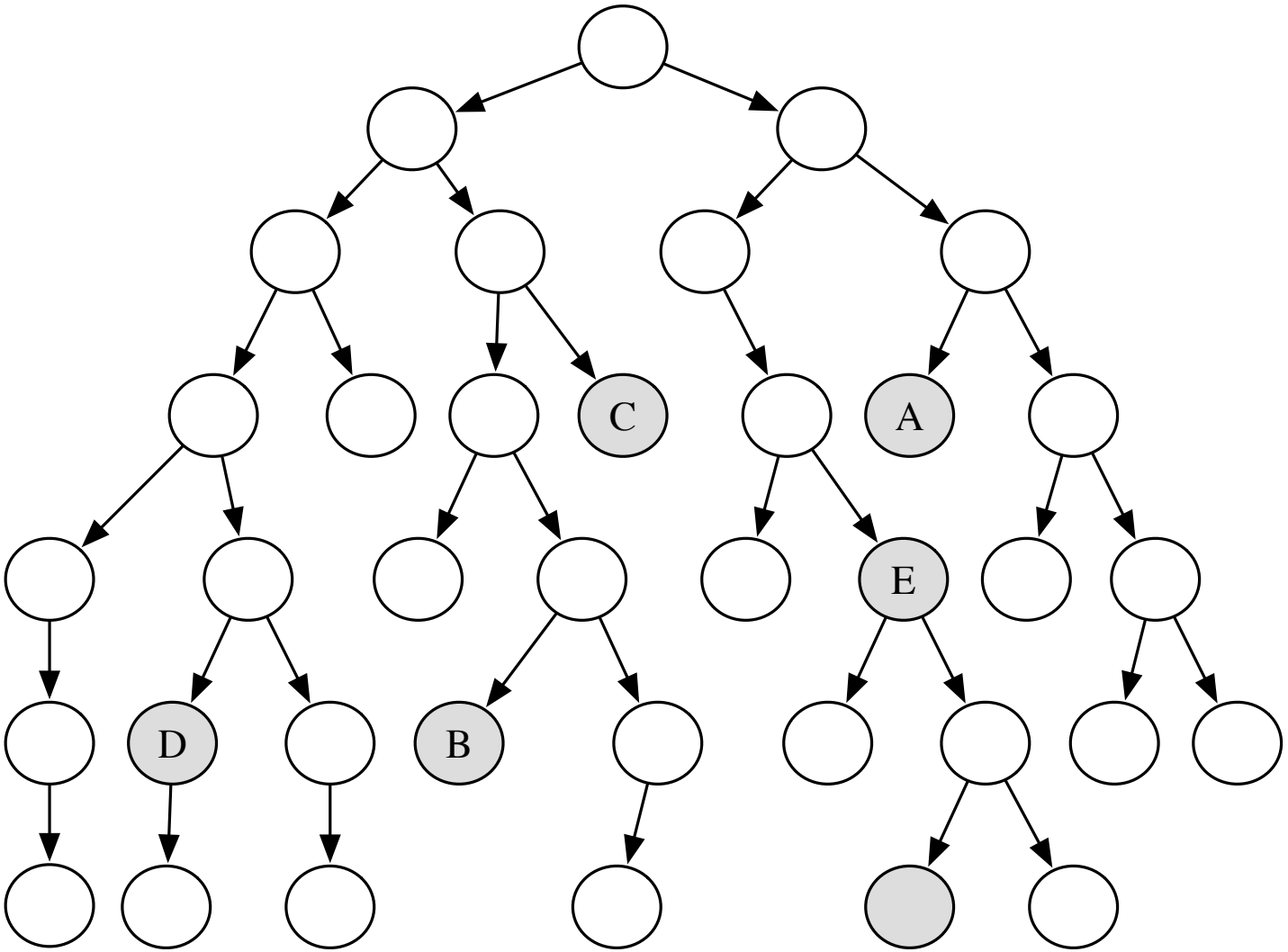
Start node is at the top.



Illustrative Graph — Breadth-first Search



Which shaded goal will breadth-first search find first?



Complexity of Breadth-first Search

- Does breadth-first search guarantee to find the path with fewest arcs?

Complexity of Breadth-first Search

- Does breadth-first search guarantee to find the path with fewest arcs?
- What happens on infinite graphs or on graphs with cycles if there is a solution?

Complexity of Breadth-first Search

- Does breadth-first search guarantee to find the path with fewest arcs?
- What happens on infinite graphs or on graphs with cycles if there is a solution?
- What is the time complexity as a function of the length of the path selected?

Complexity of Breadth-first Search

- Does breadth-first search guarantee to find the path with fewest arcs?
- What happens on infinite graphs or on graphs with cycles if there is a solution?
- What is the time complexity as a function of the length of the path selected?
- What is the space complexity as a function of the length of the path selected?

Complexity of Breadth-first Search

- Does breadth-first search guarantee to find the path with fewest arcs?
- What happens on infinite graphs or on graphs with cycles if there is a solution?
- What is the time complexity as a function of the length of the path selected?
- What is the space complexity as a function of the length of the path selected?
- How does the goal affect the search?

Lowest-cost-first Search

- Sometimes there are costs associated with arcs. The cost of a path is the sum of the costs of its arcs.

$$cost(\langle n_0, \ldots, n_k \rangle) = \sum_{i=1}^{k} cost(\langle n_{i-1}, n_i \rangle)$$

An optimal solution is one with minimum cost.

Lowest-cost-first Search

- Sometimes there are costs associated with arcs. The cost of a path is the sum of the costs of its arcs.

$$cost(\langle n_0, \ldots, n_k \rangle) = \sum_{i=1}^{k} cost(\langle n_{i-1}, n_i \rangle)$$

  An optimal solution is one with minimum cost.
- At each stage, lowest-cost-first search selects a path on the frontier with lowest cost.
- The frontier is a priority queue ordered by path cost.
- The first path to a goal is

Lowest-cost-first Search

- Sometimes there are costs associated with arcs. The cost of a path is the sum of the costs of its arcs.

$$cost(\langle n_0, \ldots, n_k \rangle) = \sum_{i=1}^{k} cost(\langle n_{i-1}, n_i \rangle)$$

  An optimal solution is one with minimum cost.
- At each stage, lowest-cost-first search selects a path on the frontier with lowest cost.
- The frontier is a priority queue ordered by path cost.
- The first path to a goal is a least-cost path to a goal node.
- When arc costs are equal $\Longrightarrow$

Lowest-cost-first Search

- Sometimes there are costs associated with arcs. The cost of a path is the sum of the costs of its arcs.

$$cost(\langle n_0, \ldots, n_k \rangle) = \sum_{i=1}^{k} cost(\langle n_{i-1}, n_i \rangle)$$

  An optimal solution is one with minimum cost.
- At each stage, lowest-cost-first search selects a path on the frontier with lowest cost.
- The frontier is a priority queue ordered by path cost.
- The first path to a goal is a least-cost path to a goal node.
- When arc costs are equal $\Longrightarrow$breadth-first search.

Summary of Search Strategies

Strategy	Frontier Selection	Complete	Halts	Space
Depth-first	Last node added			
Breadth-first	First node added			
Lowest-cost-first	Minimal $cost(p)$			

Complete — guaranteed to find a solution if there is one (for graphs with finite number of neighbours, even on infinite graphs)
Halts — on finite graph (perhaps with cycles).
Space — as a function of the length of current path

Summary of Search Strategies

Strategy	Frontier Selection	Complete	Halts	Space
Depth-first	Last node added	No	No	Linear
Breadth-first	First node added	Yes	No	Exp
Lowest-cost-first	Minimal $cost(p)$	Yes	No	Exp

Complete — guaranteed to find a solution if there is one (for graphs with finite number of neighbours, even on infinite graphs)
Halts — on finite graph (perhaps with cycles).
Space — as a function of the length of current path

Learning Objectives

At the end of the class you should be able to:

- devise an useful heuristic function for a problem
- demonstrate how best-first and A^* search will work on a graph
- predict the space and time requirements for best-first and A^* search

Heuristic Search

- Idea: don't ignore the goal when selecting paths.

Heuristic Search

- Idea: don't ignore the goal when selecting paths.
- Often there is extra knowledge that can be used to guide the search: heuristics.

Heuristic Search

- Idea: don't ignore the goal when selecting paths.
- Often there is extra knowledge that can be used to guide the search: heuristics.
- A heuristic function $h(n)$ is a nonnegative estimate of the cost of the least-cost path from node n to a goal node.

Heuristic Search

- Idea: don't ignore the goal when selecting paths.
- Often there is extra knowledge that can be used to guide the search: heuristics.
- A heuristic function $h(n)$ is a nonnegative estimate of the cost of the least-cost path from node n to a goal node.
- $h(n)$ needs to be efficient to compute.

Heuristic Search

- Idea: don't ignore the goal when selecting paths.
- Often there is extra knowledge that can be used to guide the search: heuristics.
- A heuristic function $h(n)$ is a nonnegative estimate of the cost of the least-cost path from node n to a goal node.
- $h(n)$ needs to be efficient to compute.
- h can be extended to paths: $h(\langle n_0, \ldots, n_k \rangle) = h(n_k)$.

Heuristic Search

- Idea: don't ignore the goal when selecting paths.
- Often there is extra knowledge that can be used to guide the search: heuristics.
- A heuristic function $h(n)$ is a nonnegative estimate of the cost of the least-cost path from node n to a goal node.
- $h(n)$ needs to be efficient to compute.
- h can be extended to paths: $h(\langle n_0, \ldots, n_k \rangle) = h(n_k)$.
- $h(n)$ is an underestimate if there is no path from n to a goal with cost less than $h(n)$.

Heuristic Search

- Idea: don't ignore the goal when selecting paths.
- Often there is extra knowledge that can be used to guide the search: heuristics.
- A heuristic function $h(n)$ is a nonnegative estimate of the cost of the least-cost path from node n to a goal node.
- $h(n)$ needs to be efficient to compute.
- h can be extended to paths: $h(\langle n_0, \ldots, n_k \rangle) = h(n_k)$.
- $h(n)$ is an underestimate if there is no path from n to a goal with cost less than $h(n)$.
- An admissible heuristic is a heuristic function that is an underestimate of the actual cost of a path to a goal.

Example Heuristic Functions

- If the nodes are points on a Euclidean plane and the cost is the distance, $h(n)$ can be

Example Heuristic Functions

- If the nodes are points on a Euclidean plane and the cost is the distance, $h(n)$ can be the straight-line distance from n to the closest goal.

Example Heuristic Functions

- If the nodes are points on a Euclidean plane and the cost is the distance, $h(n)$ can be the straight-line distance from n to the closest goal.
- If the nodes are locations and cost is time, we can use

Example Heuristic Functions

- If the nodes are points on a Euclidean plane and the cost is the distance, $h(n)$ can be the straight-line distance from n to the closest goal.
- If the nodes are locations and cost is time, we can use the distance to a goal divided by the maximum speed.
- If the goal is to collect all of the coins and not run out of fuel,

Example Heuristic Functions

- If the nodes are points on a Euclidean plane and the cost is the distance, $h(n)$ can be the straight-line distance from n to the closest goal.
- If the nodes are locations and cost is time, we can use the distance to a goal divided by the maximum speed.
- If the goal is to collect all of the coins and not run out of fuel, the cost is an estimate of how many steps it will take to collect the rest of the coins, refuel when necessary, and return to goal position.

Example Heuristic Functions

- If the nodes are points on a Euclidean plane and the cost is the distance, $h(n)$ can be the straight-line distance from n to the closest goal.
- If the nodes are locations and cost is time, we can use the distance to a goal divided by the maximum speed.
- If the goal is to collect all of the coins and not run out of fuel, the cost is an estimate of how many steps it will take to collect the rest of the coins, refuel when necessary, and return to goal position.
- A heuristic function can be found by solving a simpler (less constrained) version of the problem.

Heuristic depth-first Search

- Idea: in depth-first search select a neighbor that is closest to a goal according to the heuristic function.

Heuristic depth-first Search

- Idea: in depth-first search select a neighbor that is closest to a goal according to the heuristic function.
- It inherits all of the advantages/disadvantages of depth-first search, but locally heads towards a goal.

Best-first Search

- Idea: select a path whose end is closest to a goal according to the heuristic function.

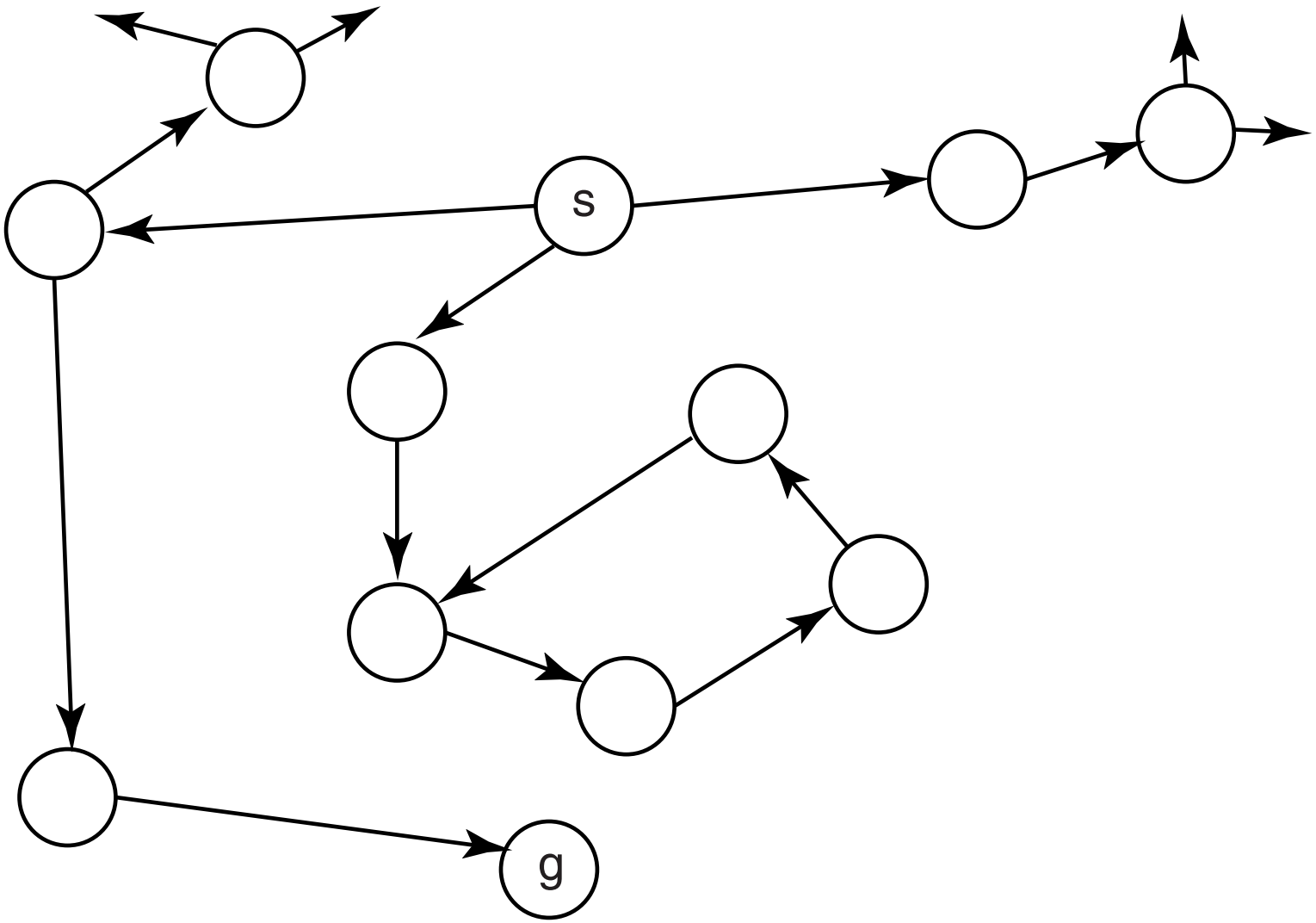
Best-first Search

- Idea: select a path whose end is closest to a goal according to the heuristic function.
- Best-first search selects a path on the frontier with minimal h-value.

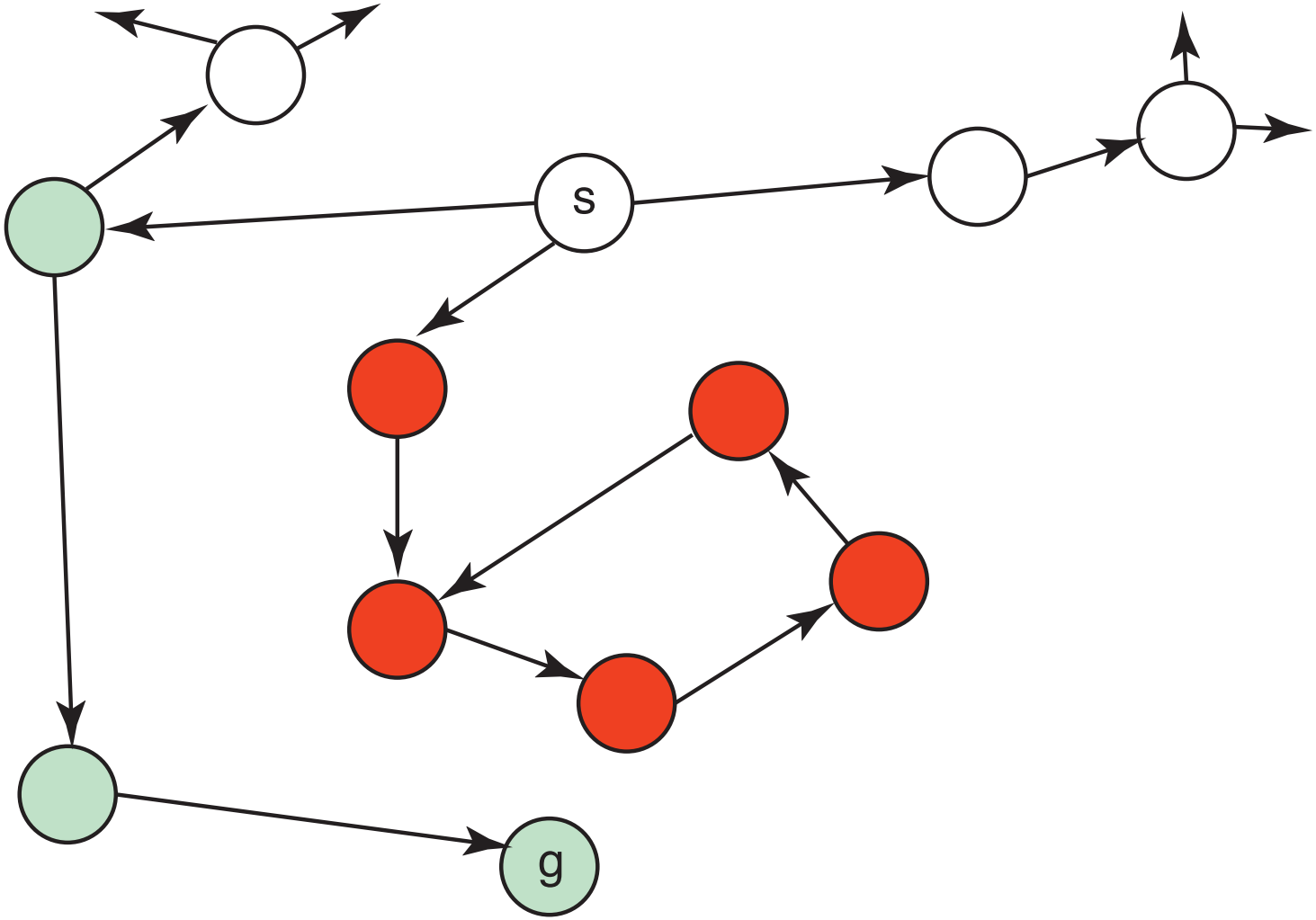
Best-first Search

- Idea: select a path whose end is closest to a goal according to the heuristic function.
- Best-first search selects a path on the frontier with minimal h-value.
- It treats the frontier as a priority queue ordered by h.

Illustrative Graph — Heuristic Search



Illustrative Graph — Heuristic Search



Complexity of Best-first Search

- Does best-first search guarantee to find a least-cost path?

Complexity of Best-first Search

- Does best-first search guarantee to find a least-cost path?
- Does best-first search guarantee to find a path with fewest arcs?

Complexity of Best-first Search

- Does best-first search guarantee to find a least-cost path?
- Does best-first search guarantee to find a path with fewest arcs?
- What happens on infinite graphs or on graphs with cycles if there is a solution?

Complexity of Best-first Search

- Does best-first search guarantee to find a least-cost path?
- Does best-first search guarantee to find a path with fewest arcs?
- What happens on infinite graphs or on graphs with cycles if there is a solution?
- What is the time complexity as a function of length of the path selected?

Complexity of Best-first Search

- Does best-first search guarantee to find a least-cost path?
- Does best-first search guarantee to find a path with fewest arcs?
- What happens on infinite graphs or on graphs with cycles if there is a solution?
- What is the time complexity as a function of length of the path selected?
- What is the space complexity as a function of length of the path selected?

Complexity of Best-first Search

- Does best-first search guarantee to find a least-cost path?
- Does best-first search guarantee to find a path with fewest arcs?
- What happens on infinite graphs or on graphs with cycles if there is a solution?
- What is the time complexity as a function of length of the path selected?
- What is the space complexity as a function of length of the path selected?
- How does the goal affect the search?

A^* Search

- A^* search uses both path cost and heuristic values

A^* Search

- A^* search uses both path cost and heuristic values
- $cost(p)$ is the cost of path p.
- $h(p)$ estimates the cost from the end of p to a goal.

A^* Search

- A^* search uses both path cost and heuristic values
- $cost(p)$ is the cost of path p.
- $h(p)$ estimates the cost from the end of p to a goal.
- Let $f(p) = cost(p) + h(p)$.
  $f(p)$ estimates the total path cost of going from a start node to a goal via p.

$$\underbrace{\underbrace{start \xrightarrow{\text{path } p} n}_{cost(p)} \underbrace{\xrightarrow{\text{estimate}} goal}_{h(p)}}_{f(p)}$$

A^* Search

- A^* search uses both path cost and heuristic values
- $cost(p)$ is the cost of path p.
- $h(p)$ estimates the cost from the end of p to a goal.
- Let $f(p) = cost(p) + h(p)$.
  $f(p)$ estimates the total path cost of going from a start node to a goal via p.

$$\underbrace{\underbrace{start \xrightarrow{\text{path } p} n}_{cost(p)} \underbrace{\xrightarrow{\text{estimate}} goal}_{h(p)}}_{f(p)}$$

- In A^* search, the frontier is a priority queue ordered by $f(p)$.

A^* Search

- A^* search uses both path cost and heuristic values
- $cost(p)$ is the cost of path p.
- $h(p)$ estimates the cost from the end of p to a goal.
- Let $f(p) = cost(p) + h(p)$.
  $f(p)$ estimates the total path cost of going from a start node to a goal via p.

$$\underbrace{\underbrace{start \xrightarrow{\text{path } p} n}_{cost(p)} \underbrace{\xrightarrow{\text{estimate}} goal}_{h(p)}}_{f(p)}$$

- In A^* search, the frontier is a priority queue ordered by $f(p)$.
- It always selects the path on the frontier with the lowest estimated cost from the start to a goal node constrained to go via that path.

A^* Search Algorithm

- A^* is a mix of lowest-cost-first and best-first search.

A^* Search Algorithm

- A^* is a mix of lowest-cost-first and best-first search.
- It treats the frontier as a priority queue ordered by $f(p)$.

A^* Search Algorithm

- A^* is a mix of lowest-cost-first and best-first search.
- It treats the frontier as a priority queue ordered by $f(p)$.
- It always selects the node on the frontier with the lowest estimated distance from the start to a goal node constrained to go via that node.

Complexity of A^* Search

- Does A^* search guarantee to find the path with fewest arcs?

Complexity of A^* Search

- Does A^* search guarantee to find the path with fewest arcs?
- Does A^* search guarantee to find the least-cost path?

Complexity of A^* Search

- Does A^* search guarantee to find the path with fewest arcs?
- Does A^* search guarantee to find the least-cost path?
- What happens on infinite graphs or on graphs with cycles if there is a solution?

Complexity of A^* Search

- Does A^* search guarantee to find the path with fewest arcs?
- Does A^* search guarantee to find the least-cost path?
- What happens on infinite graphs or on graphs with cycles if there is a solution?
- What is the time complexity as a function of length of the path selected?

Complexity of A^* Search

- Does A^* search guarantee to find the path with fewest arcs?
- Does A^* search guarantee to find the least-cost path?
- What happens on infinite graphs or on graphs with cycles if there is a solution?
- What is the time complexity as a function of length of the path selected?
- What is the space complexity as a function of length of the path selected?

Complexity of A^* Search

- Does A^* search guarantee to find the path with fewest arcs?
- Does A^* search guarantee to find the least-cost path?
- What happens on infinite graphs or on graphs with cycles if there is a solution?
- What is the time complexity as a function of length of the path selected?
- What is the space complexity as a function of length of the path selected?
- How does the goal affect the search?

Admissibility of A^*

If there is a solution, A^* always finds an optimal solution – -the first path to a goal selected — if

Admissibility of A^*

If there is a solution, A^* always finds an optimal solution – -the first path to a goal selected — if

- the branching factor is finite

Admissibility of A^*

If there is a solution, A^* always finds an optimal solution – -the first path to a goal selected — if

- the branching factor is finite
- arc costs are bounded above zero (there is some $\epsilon > 0$ such that all of the arc costs are greater than ϵ), and

Admissibility of A^*

If there is a solution, A^* always finds an optimal solution – -the first path to a goal selected — if

- the branching factor is finite
- arc costs are bounded above zero (there is some $\epsilon > 0$ such that all of the arc costs are greater than ϵ), and
- $h(n)$ is nonnegative and an underestimate of the cost of the shortest path from n to a goal node:

$$0 \leq h(n) \leq \text{cost of shortest path from } n \text{ to a goal}$$

Why is A^* admissible?

- If a path p to a goal is selected from the frontier, can there be a lower cost path to a goal?

Why is A^* admissible?

- If a path p to a goal is selected from the frontier, can there be a lower cost path to a goal?
- $h(p) =$

Why is A^* admissible?

- If a path p to a goal is selected from the frontier, can there be a lower cost path to a goal?
- $h(p) = 0$

Why is A^* admissible?

- If a path p to a goal is selected from the frontier, can there be a lower cost path to a goal?
- $h(p) = 0$
- Suppose path p' is on the frontier. Because p was chosen before p', and $h(p) = 0$:

Why is A^* admissible?

- If a path p to a goal is selected from the frontier, can there be a lower cost path to a goal?
- $h(p) = 0$
- Suppose path p' is on the frontier. Because p was chosen before p', and $h(p) = 0$:

$$cost(p) \leq cost(p') + h(p').$$

Why is A^* admissible?

- If a path p to a goal is selected from the frontier, can there be a lower cost path to a goal?
- $h(p) = 0$
- Suppose path p' is on the frontier. Because p was chosen before p', and $h(p) = 0$:

$$cost(p) \leq cost(p') + h(p').$$

- Because h is an underestimate:

for any path p'' to a goal that extends p'.

Why is A^* admissible?

- If a path p to a goal is selected from the frontier, can there be a lower cost path to a goal?
- $h(p) = 0$
- Suppose path p' is on the frontier. Because p was chosen before p', and $h(p) = 0$:

$$cost(p) \leq cost(p') + h(p').$$

- Because h is an underestimate:

$$cost(p') + h(p') \leq cost(p'')$$

for any path p'' to a goal that extends p'.

Why is A^* admissible?

- If a path p to a goal is selected from the frontier, can there be a lower cost path to a goal?
- $h(p) = 0$
- Suppose path p' is on the frontier. Because p was chosen before p', and $h(p) = 0$:

$$cost(p) \leq cost(p') + h(p').$$

- Because h is an underestimate:

$$cost(p') + h(p') \leq cost(p'')$$

  for any path p'' to a goal that extends p'.
- So $cost(p) \leq cost(p'')$ for any other path p'' to a goal.

Why is A^* admissible?

A^* can always find a solution if there is one:

- The frontier always contains the initial part of a path to a goal, before that goal is selected.

Why is A^* admissible?

A^* can always find a solution if there is one:

- The frontier always contains the initial part of a path to a goal, before that goal is selected.
- A^* halts, as the costs of the paths on the frontier keeps increasing, and will eventually exceed any finite number.

How do good heuristics help?

Suppose c is the cost of an optimal solution. What happens to a path p from a start node, where

- $cost(p) + h(p) < c$

How do good heuristics help?

Suppose c is the cost of an optimal solution. What happens to a path p from a start node, where

- $cost(p) + h(p) < c$
  It will be expanded
- $cost(p) + h(p) > c$

How do good heuristics help?

Suppose c is the cost of an optimal solution. What happens to a path p from a start node, where

- $cost(p) + h(p) < c$
  It will be expanded
- $cost(p) + h(p) > c$
  It will not be expanded
- $cost(p) + h(p) = c$

How do good heuristics help?

Suppose c is the cost of an optimal solution. What happens to a path p from a start node, where

- $cost(p) + h(p) < c$
  It will be expanded
- $cost(p) + h(p) > c$
  It will not be expanded
- $cost(p) + h(p) = c$
  It might or might not be expanded.

How do good heuristics help?

Suppose c is the cost of an optimal solution. What happens to a path p from a start node, where

- $cost(p) + h(p) < c$
  It will be expanded
- $cost(p) + h(p) > c$
  It will not be expanded
- $cost(p) + h(p) = c$
  It might or might not be expanded.

How can a better heuristic function help?

Summary of Search Strategies

Strategy	Frontier Selection	Complete	Halts	Space
Depth-first	Last node added			
Breadth-first	First node added			
Heuristic depth-first	Local min $h(p)$			
Best-first	Global min $h(p)$			
Lowest-cost-first	Minimal $cost(p)$			
A^*	Minimal $f(p)$			

Complete — if there a path to a goal, it can find one, even on infinite graphs.
Halts — on finite graph (perhaps with cycles).
Space — as a function of the length of current path

Summary of Search Strategies

Strategy	Frontier Selection	Complete	Halts	Space
Depth-first	Last node added	No	No	Linear
Breadth-first	First node added	Yes	No	Exp
Heuristic depth-first	Local min $h(p)$	No	No	Linear
Best-first	Global min $h(p)$	No	No	Exp
Lowest-cost-first	Minimal $cost(p)$	Yes	No	Exp
A^*	Minimal $f(p)$	Yes	No	Exp

Complete — if there a path to a goal, it can find one, even on infinite graphs.
Halts — on finite graph (perhaps with cycles).
Space — as a function of the length of current path

Learning Objectives

At the end of the class you should be able to:

- explain how cycle checking and multiple-path pruning can improve efficiency of search algorithms
- explain the complexity of cycle checking and multiple-path pruning for different search algorithms
- justify why the monotone restriction is useful for A^* search
- predict whether forward, backward, bidirectional or island-driven search is better for a particular problem
- demonstrate how dynamic programming works for a particular problem

Summary of Search Strategies

Strategy	Frontier Selection	Complete	Halts	Space
Depth-first	Last node added			
Breadth-first	First node added			
Best-first	Global min $h(p)$			
Lowest-cost-first	Minimal $cost(p)$			
A^*	Minimal $f(p)$			

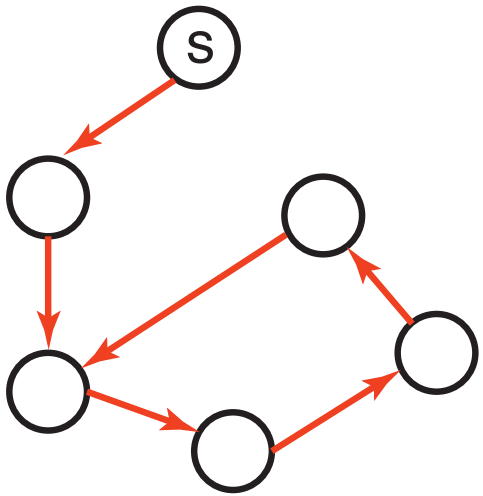
Complete — if there a path to a goal, it can find one, even on infinite graphs.
Halts — on finite graph (perhaps with cycles).
Space — as a function of the length of current path

Summary of Search Strategies

Strategy	Frontier Selection	Complete	Halts	Space
Depth-first	Last node added	No	No	Linear
Breadth-first	First node added	Yes	No	Exp
Best-first	Global min $h(p)$	No	No	Exp
Lowest-cost-first	Minimal $cost(p)$	Yes	No	Exp
A^*	Minimal $f(p)$	Yes	No	Exp

Complete — if there a path to a goal, it can find one, even on infinite graphs.
Halts — on finite graph (perhaps with cycles).
Space — as a function of the length of current path

Cycle Pruning



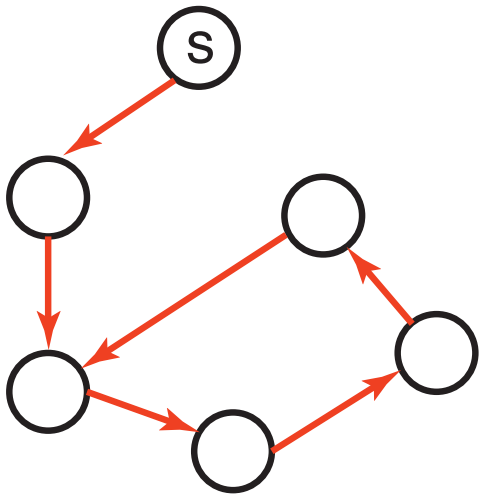
- A searcher can prune a path that ends in a node already on the path, without removing an optimal solution.

Graph searching with cycle pruning

Input: a graph,
a set of start nodes,
Boolean procedure $goal(n)$ that tests if n is a goal node.

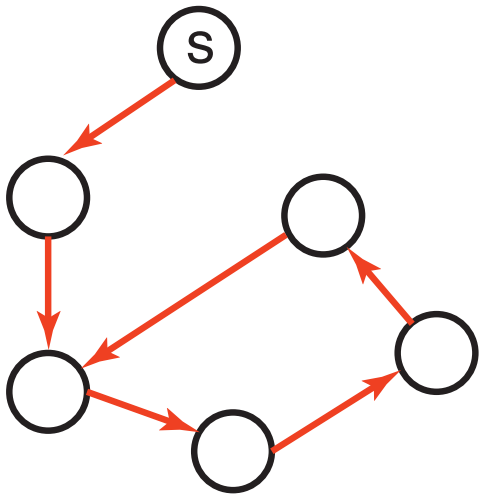
$frontier := \{\langle s \rangle : s \text{ is a start node}\}$
while *frontier* is not empty:
  select and **remove** path $\langle n_0, \ldots, n_k \rangle$ from *frontier*
  if $n_k \notin \{n_0, \ldots, n_{k-1}\}$:
    if $goal(n_k)$:
      return $\langle n_0, \ldots, n_k \rangle$
    $Frontier := Frontier \cup \{\langle n_0, \ldots, n_k, n \rangle : \langle n_k, n \rangle \in A\}$

Cycle Pruning



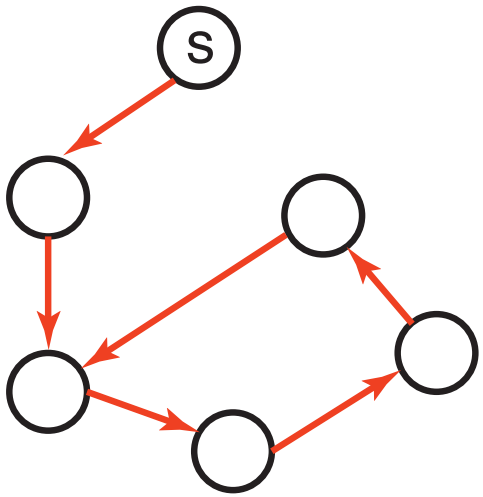
- In depth-first search, checking for cycles can be done in _____ time in path length.

Cycle Pruning



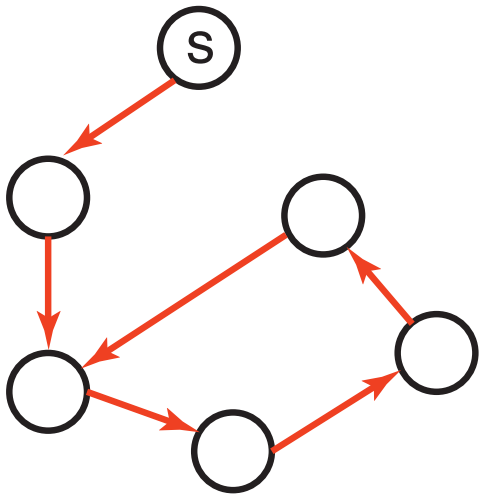
- In depth-first search, checking for cycles can be done in <u>constant</u> time in path length.

Cycle Pruning



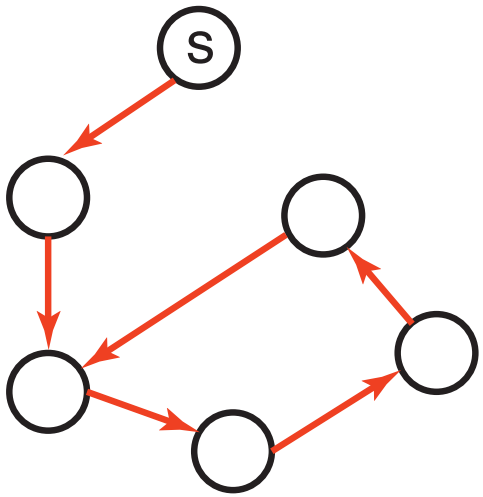
- In depth-first search, checking for cycles can be done in <u>constant</u> time in path length.
- For other methods, checking for cycles can be done in _____ time in path length.

Cycle Pruning



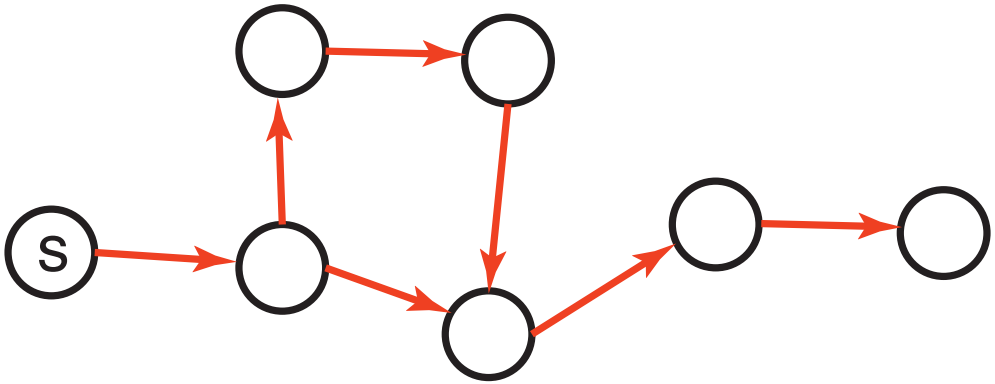
- In depth-first search, checking for cycles can be done in constant time in path length.
- For other methods, checking for cycles can be done in linear time in path length.

Cycle Pruning



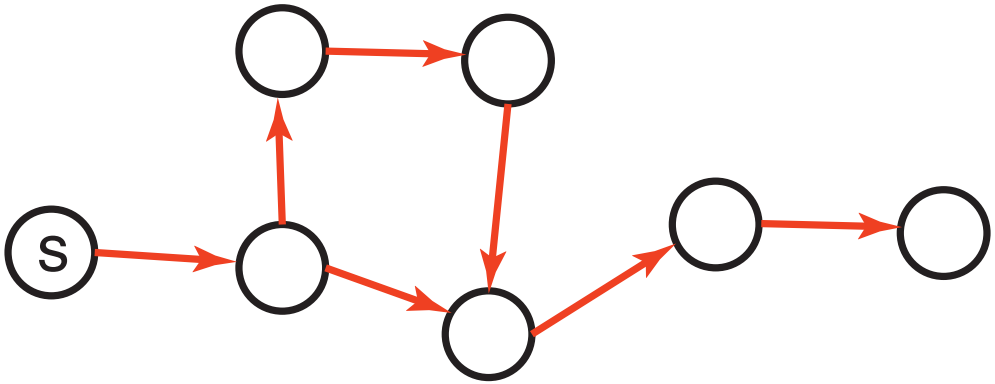
- In depth-first search, checking for cycles can be done in constant time in path length.
- For other methods, checking for cycles can be done in linear time in path length.
- With cycle pruning, which algorithms halt on finite graphs?

Multiple-Path Pruning



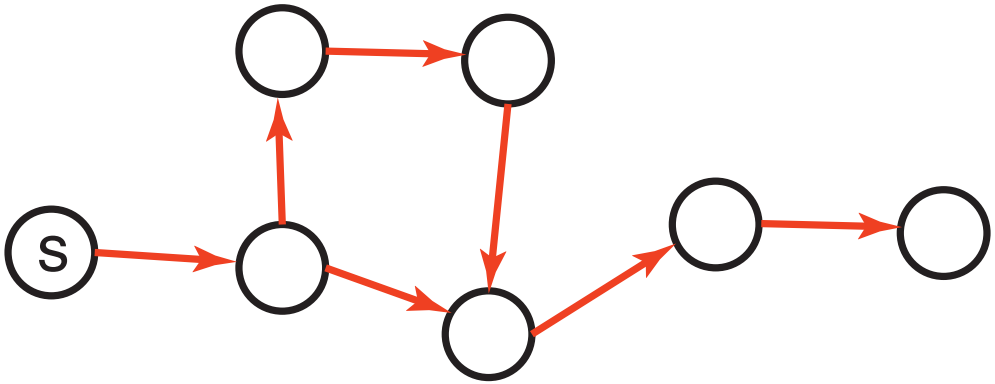
- Multiple path pruning: prune a path to node n that the searcher has already found a path to.

Multiple-Path Pruning



- Multiple path pruning: prune a path to node n that the searcher has already found a path to.
- What needs to be stored?

Multiple-Path Pruning



- Multiple path pruning: prune a path to node n that the searcher has already found a path to.
- What needs to be stored?
- Lowest-cost-first search with multiple-path pruning is Dijkstra's algorithm, and is the same as A^* with multiple-path pruning and a heuristic function of 0.

Graph searching with multiple-path pruning

Input: a graph,
    a set of start nodes,
    Boolean procedure $goal(n)$ that tests if n is a goal node.
$frontier := \{\langle s \rangle : s \text{ is a start node}\}$
$expanded := \{\}$
while *frontier* is not empty:
    select and **remove** path $\langle n_0, \ldots, n_k \rangle$ from *frontier*
    if $n_k \notin expanded$:
        add n_k to *expanded*
        if $goal(n_k)$:
            return $\langle n_0, \ldots, n_k \rangle$
        $Frontier := Frontier \cup \{\langle n_0, \ldots, n_k, n \rangle : \langle n_k, n \rangle \in A\}$

Multiple-Path Pruning

- How does multiple-path pruning compare to cycle pruning?

Multiple-Path Pruning

- How does multiple-path pruning compare to cycle pruning?
- Which search algorithms with multiple-path pruning always halt on finite graphs?

Multiple-Path Pruning

- How does multiple-path pruning compare to cycle pruning?
- Which search algorithms with multiple-path pruning always halt on finite graphs?
- What is the time overhead of multiple-path pruning?

Multiple-Path Pruning

- How does multiple-path pruning compare to cycle pruning?
- Which search algorithms with multiple-path pruning always halt on finite graphs?
- What is the time overhead of multiple-path pruning?
- What is the space overhead of multiple-path pruning?

Multiple-Path Pruning

- How does multiple-path pruning compare to cycle pruning?
- Which search algorithms with multiple-path pruning always halt on finite graphs?
- What is the time overhead of multiple-path pruning?
- What is the space overhead of multiple-path pruning?
- Is it better for depth-first or breadth-first searches?

Multiple-Path Pruning

- How does multiple-path pruning compare to cycle pruning?
- Which search algorithms with multiple-path pruning always halt on finite graphs?
- What is the time overhead of multiple-path pruning?
- What is the space overhead of multiple-path pruning?
- Is it better for depth-first or breadth-first searches?
- Can multiple-path pruning prevent an optimal solution being found?

Multiple-Path Pruning & Optimal Solutions

Problem: what if a subsequent path to n has a lower cost than the first path to n?

Multiple-Path Pruning & Optimal Solutions

Problem: what if a subsequent path to n has a lower cost than the first path to n?

- remove all paths from the frontier that use the longer path.
- change the initial segment of the paths on the frontier to use the lower-cost path.
- ensure this doesn't happen. Make sure that the lower-cost path to a node is expanded first.

Multiple-Path Pruning & A^*

- Suppose path p to n was selected, but there is a lower-cost path to n. Suppose this lower-cost path is via path p' on the frontier.
- Suppose path p' ends at node n'.

Multiple-Path Pruning & A^*

- Suppose path p to n was selected, but there is a lower-cost path to n. Suppose this lower-cost path is via path p' on the frontier.
- Suppose path p' ends at node n'.
- p was selected before p', so:

Multiple-Path Pruning & A^*

- Suppose path p to n was selected, but there is a lower-cost path to n. Suppose this lower-cost path is via path p' on the frontier.
- Suppose path p' ends at node n'.
- p was selected before p', so:
  $cost(p) + h(n) \leq cost(p') + h(n')$.
- Suppose $cost(n', n)$ is the actual cost of a path from n' to n. The path to n via p' has a lower cost that p so:

Multiple-Path Pruning & A^*

- Suppose path p to n was selected, but there is a lower-cost path to n. Suppose this lower-cost path is via path p' on the frontier.
- Suppose path p' ends at node n'.
- p was selected before p', so:
  $cost(p) + h(n) \leq cost(p') + h(n')$.
- Suppose $cost(n', n)$ is the actual cost of a path from n' to n. The path to n via p' has a lower cost that p so:
  $cost(p') + cost(n', n) < cost(p)$.

$$cost(n', n) <$$

Multiple-Path Pruning & A^*

- Suppose path p to n was selected, but there is a lower-cost path to n. Suppose this lower-cost path is via path p' on the frontier.
- Suppose path p' ends at node n'.
- p was selected before p', so:
  $cost(p) + h(n) \leq cost(p') + h(n')$.
- Suppose $cost(n', n)$ is the actual cost of a path from n' to n. The path to n via p' has a lower cost that p so:
  $cost(p') + cost(n', n) < cost(p)$.

$$cost(n', n) < cost(p) - cost(p') \leq$$

Multiple-Path Pruning & A^*

- Suppose path p to n was selected, but there is a lower-cost path to n. Suppose this lower-cost path is via path p' on the frontier.
- Suppose path p' ends at node n'.
- p was selected before p', so:
  $cost(p) + h(n) \leq cost(p') + h(n')$.
- Suppose $cost(n', n)$ is the actual cost of a path from n' to n. The path to n via p' has a lower cost that p so:
  $cost(p') + cost(n', n) < cost(p)$.

$$cost(n', n) < cost(p) - cost(p') \leq h(n') - h(n).$$

We can ensure this doesn't occur if
$|h(n') - h(n)| \leq cost(n', n)$.

Monotone Restriction

- Heuristic function h satisfies the monotone restriction if $|h(m) - h(n)| \leq cost(m, n)$ for every arc $\langle m, n \rangle$.

Monotone Restriction

- Heuristic function h satisfies the monotone restriction if $|h(m) - h(n)| \leq cost(m, n)$ for every arc $\langle m, n \rangle$.
- If h satisfies the monotone restriction, A^* with multiple path pruning always finds a least-cost path to a goal.

Monotone Restriction

- Heuristic function h satisfies the monotone restriction if $|h(m) - h(n)| \leq cost(m, n)$ for every arc $\langle m, n \rangle$.
- If h satisfies the monotone restriction, A^* with multiple path pruning always finds a least-cost path to a goal.
- This is a strengthening of the admissibility criterion.

Direction of Search

- The definition of searching is symmetric: find path from start nodes to goal node or from goal node to start nodes (with reversed arcs).

Direction of Search

- The definition of searching is symmetric: find path from start nodes to goal node or from goal node to start nodes (with reversed arcs).
- Forward branching factor: number of arcs out of a node.
- Backward branching factor: number of arcs into a node.

Direction of Search

- The definition of searching is symmetric: find path from start nodes to goal node or from goal node to start nodes (with reversed arcs).
- Forward branching factor: number of arcs out of a node.
- Backward branching factor: number of arcs into a node.
- Search complexity is b^n. Should use forward search if forward branching factor is less than backward branching factor, and vice versa.

Direction of Search

- The definition of searching is symmetric: find path from start nodes to goal node or from goal node to start nodes (with reversed arcs).
- Forward branching factor: number of arcs out of a node.
- Backward branching factor: number of arcs into a node.
- Search complexity is b^n. Should use forward search if forward branching factor is less than backward branching factor, and vice versa.
- Note: when graph is dynamically constructed, the backwards graph may not be available. One might be more difficult to compute than the other.

Bidirectional Search

- Idea: search backward from the goal and forward from the start simultaneously.

Bidirectional Search

- Idea: search backward from the goal and forward from the start simultaneously.
- This wins as $2b^{k/2} \ll b^k$.
  This can result in an exponential saving in time and space.

Bidirectional Search

- Idea: search backward from the goal and forward from the start simultaneously.
- This wins as $2b^{k/2} \ll b^k$.
  This can result in an exponential saving in time and space.
- The main problem is making sure the frontiers meet.

Bidirectional Search

- Idea: search backward from the goal and forward from the start simultaneously.
- This wins as $2b^{k/2} \ll b^k$.
  This can result in an exponential saving in time and space.
- The main problem is making sure the frontiers meet.
- This is often used with
  - a breadth-first method (e.g., least-cost-first search) that builds a set of states that can lead to the goal quickly.
  - in the other direction, another method (typically depth-first) can be used to find a path to these interesting states.

Bidirectional Search

- Idea: search backward from the goal and forward from the start simultaneously.
- This wins as $2b^{k/2} \ll b^k$.
  This can result in an exponential saving in time and space.
- The main problem is making sure the frontiers meet.
- This is often used with
  - a breadth-first method (e.g., least-cost-first search) that builds a set of states that can lead to the goal quickly.
  - in the other direction, another method (typically depth-first) can be used to find a path to these interesting states.
  - How much is stored in the breadth-first method, can be tuned depending on the space available.

Island Driven Search

- Idea: find a set of islands between s and g.

$$s \longrightarrow i_1 \longrightarrow i_2 \longrightarrow \ldots \longrightarrow i_{m-1} \longrightarrow g$$

  There are m smaller problems rather than 1 big problem.
- This can win as $mb^{k/m} \ll b^k$.

Island Driven Search

- Idea: find a set of islands between s and g.

$$s \longrightarrow i_1 \longrightarrow i_2 \longrightarrow \ldots \longrightarrow i_{m-1} \longrightarrow g$$

  There are m smaller problems rather than 1 big problem.
- This can win as $mb^{k/m} \ll b^k$.
- The problem is to identify the islands that the path must pass through. It is difficult to guarantee optimality.
- Requires more knowledge than just the graph and a heuristic function.

Island Driven Search

- Idea: find a set of islands between s and g.

$$s \longrightarrow i_1 \longrightarrow i_2 \longrightarrow \ldots \longrightarrow i_{m-1} \longrightarrow g$$

  There are m smaller problems rather than 1 big problem.
- This can win as $mb^{k/m} \ll b^k$.
- The problem is to identify the islands that the path must pass through. It is difficult to guarantee optimality.
- Requires more knowledge than just the graph and a heuristic function.
- The subproblems can be solved using islands $\Longrightarrow$ hierarchy of abstractions.

Dynamic Programming

Idea: for statically stored graphs, build a table of $dist(n)$ the actual distance of the shortest path from node n to a goal.
This can be built backwards from the goal:

$$dist(n) = \begin{cases} 0 & \text{if } is_goal(n), \\ \min_{\langle n,m\rangle \in A}(|\langle n, m\rangle| + dist(m)) & \text{otherwise.} \end{cases}$$

using least-cost-first search in the reverse graph.

Dynamic Programming

Idea: for statically stored graphs, build a table of $dist(n)$ the actual distance of the shortest path from node n to a goal.
This can be built backwards from the goal:

$$dist(n) = \begin{cases} 0 & \text{if } is_goal(n), \\ \min_{\langle n,m\rangle \in A}(|\langle n, m\rangle| + dist(m)) & \text{otherwise.} \end{cases}$$

using least-cost-first search in the reverse graph.

- This can be used locally to determine what to do from any state.

Dynamic Programming

Idea: for statically stored graphs, build a table of $dist(n)$ the actual distance of the shortest path from node n to a goal.
This can be built backwards from the goal:

$$dist(n) = \begin{cases} 0 & \text{if } is_goal(n), \\ \min_{\langle n,m\rangle \in A}(|\langle n, m\rangle| + dist(m)) & \text{otherwise.} \end{cases}$$

using least-cost-first search in the reverse graph.

- This can be used locally to determine what to do from any state.
- Why not use A^*?

Dynamic Programming

Idea: for statically stored graphs, build a table of $dist(n)$ the actual distance of the shortest path from node n to a goal.
This can be built backwards from the goal:

$$dist(n) = \begin{cases} 0 & \text{if } is_goal(n), \\ \min_{\langle n,m\rangle \in A}(|\langle n, m\rangle| + dist(m)) & \text{otherwise.} \end{cases}$$

using least-cost-first search in the reverse graph.

- This can be used locally to determine what to do from any state.
- Why not use A^*?
- There are two main problems:

Dynamic Programming

Idea: for statically stored graphs, build a table of $dist(n)$ the actual distance of the shortest path from node n to a goal.
This can be built backwards from the goal:

$$dist(n) = \begin{cases} 0 & \text{if } is_goal(n), \\ \min_{\langle n,m\rangle \in A}(|\langle n, m\rangle| + dist(m)) & \text{otherwise.} \end{cases}$$

using least-cost-first search in the reverse graph.

- This can be used locally to determine what to do from any state.
- Why not use A^*?
- There are two main problems:
  - It requires enough space to store the graph.
  - The *dist* function needs to be recomputed for each goal.

Learning Objectives

At the end of the class you should be able to:

- justify why depth-bounded search is useful
- demonstrate how iterative-deepening works for a particular problem
- demonstrate how depth-first branch-and-bound works for a particular problem

Summary of Search Strategies

Strategy	Frontier	Complete	Halts	Space
Depth-first w/o CP	Last added	No	No	Linear
Depth-first w CP	Last added	No	Yes	Linear
Depth-first w MPP	Last added	No	Yes	Exp
Breadth-first w/o MPP	First added	Yes	No	Exp
Breadth-first w MPP	First added	Yes	Yes	Exp
Best-first w/o MPP	Min $h(p)$	No	No	Exp
Best-first w MPP	Min $h(p)$	No	Yes	Exp
A^* w/o MPP	Min $f(p)$	Yes	No	Exp
A^* w MPP	Min $f(p)$	Yes	Yes	Exp

Complete — if there a path to a goal, it can find one, even on infinite graphs.

Halts — on finite graph (perhaps with cycles).

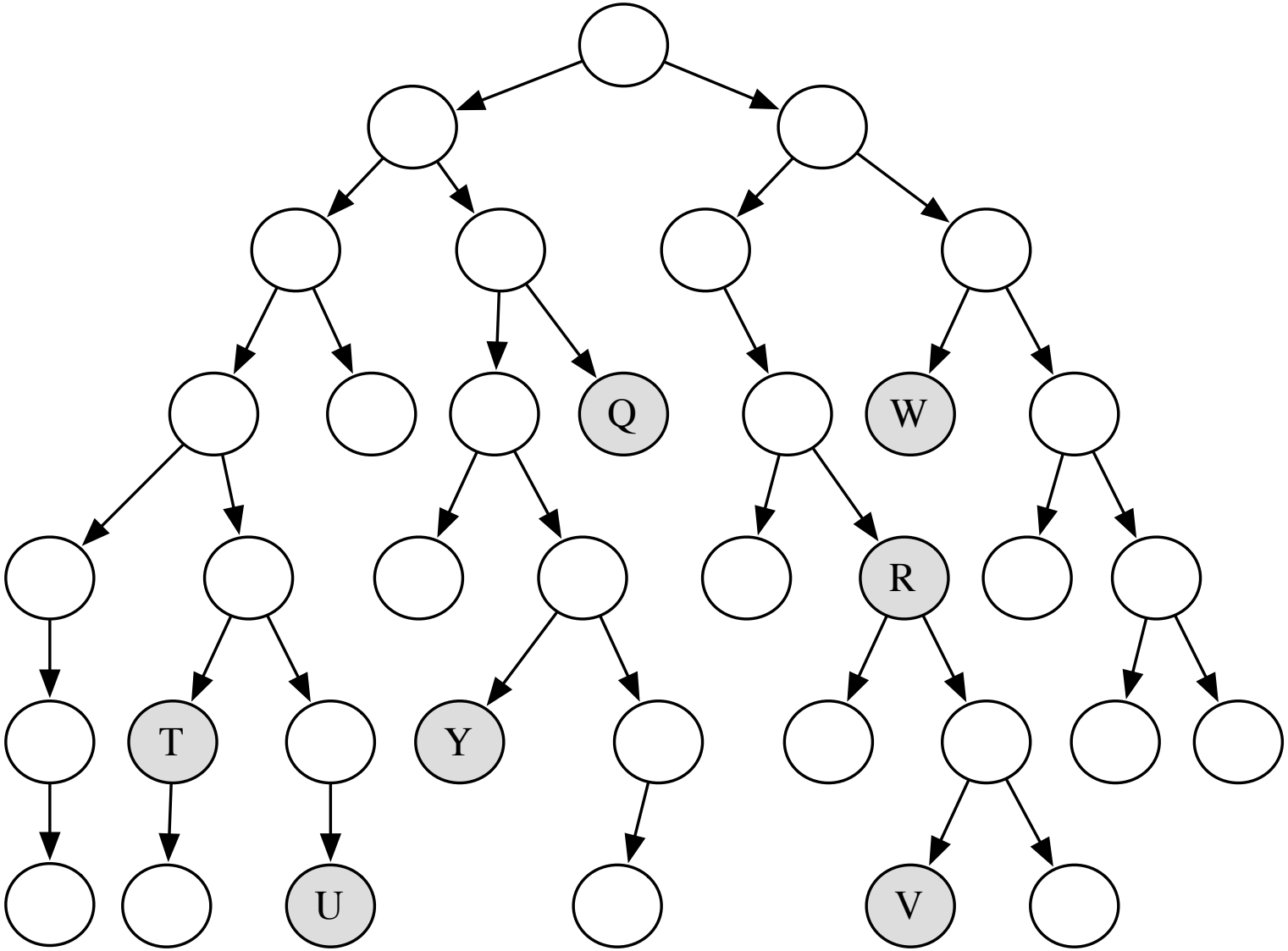
Space — as a function of the length of current path

Assume graph satisfies the assumptions of A^* proof + montonicity

Bounded Depth-first search

- A bounded depth-first search takes a bound (cost or depth) and does not expand paths that exceed the bound.
  - explores part of the search graph
  - uses space linear in the depth of the search.
- How does this relate to other searches?
- How can this be extended to be complete?

Which shaded goal will a depth-bounded search find first?



Iterative-deepening search

- Iterative-deepening search:
  - Start with a bound $b = 0$.
  - Do a bounded depth-first search with bound b
  - If a solution is found return that solution
  - Otherwise increment b and repeat.

Iterative-deepening search

- Iterative-deepening search:
  - Start with a bound $b = 0$.
  - Do a bounded depth-first search with bound b
  - If a solution is found return that solution
  - Otherwise increment b and repeat.
- This will find the same first solution as what other method?

Iterative-deepening search

- Iterative-deepening search:
  - Start with a bound $b = 0$.
  - Do a bounded depth-first search with bound b
  - If a solution is found return that solution
  - Otherwise increment b and repeat.
- This will find the same first solution as what other method?
- How much space is used?

Iterative-deepening search

- Iterative-deepening search:
  - Start with a bound $b = 0$.
  - Do a bounded depth-first search with bound b
  - If a solution is found return that solution
  - Otherwise increment b and repeat.
- This will find the same first solution as what other method?
- How much space is used?
- What happens if there is no path to a goal?

Iterative-deepening search

- Iterative-deepening search:
  - Start with a bound $b = 0$.
  - Do a bounded depth-first search with bound b
  - If a solution is found return that solution
  - Otherwise increment b and repeat.
- This will find the same first solution as what other method?
- How much space is used?
- What happens if there is no path to a goal?
- Surely recomputing paths is wasteful!!!

Iterative Deepening Complexity

Complexity with solution at depth k & branching factor b:

level	breadth-first	iterative deepening	# nodes
1	1	k	b
2	1	$k-1$	b^2
...	...	...	...
$k-1$	1	2	b^{k-1}
k	1	1	b^k
total			

Iterative Deepening Complexity

Complexity with solution at depth k & branching factor b:

level	breadth-first	iterative deepening	# nodes
1	1	k	b
2	1	$k-1$	b^2
...	...	...	...
$k-1$	1	2	b^{k-1}
k	1	1	b^k
total	$\geq b^k$	$\leq b^k \left(\frac{b}{b-1}\right)^2$	

Depth-first Branch-and-Bound

- combines depth-first search with heuristic information.
- finds optimal solution.
- most useful when there are multiple solutions, and we want an optimal one.
- uses the space of depth-first search.

Depth-first Branch-and-Bound

Suppose we want to find a single optimal solution.

- Suppose *bound* is the cost of the lowest-cost path found to a goal so far.
- What if the search encounters a path p such that $cost(p) + h(p) \geq bound$?

Depth-first Branch-and-Bound

Suppose we want to find a single optimal solution.

- Suppose *bound* is the cost of the lowest-cost path found to a goal so far.
- What if the search encounters a path p such that $cost(p) + h(p) \geq bound$?
  p can be pruned.
- What can we do if a non-pruned path to a goal is found?

Depth-first Branch-and-Bound

Suppose we want to find a single optimal solution.

- Suppose *bound* is the cost of the lowest-cost path found to a goal so far.
- What if the search encounters a path p such that $cost(p) + h(p) \geq bound$?
  p can be pruned.
- What can we do if a non-pruned path to a goal is found?
  bound can be set to the cost of p, and p can be remembered as the best solution so far.
- Why should this use a depth-first search?

Depth-first Branch-and-Bound

Suppose we want to find a single optimal solution.

- Suppose *bound* is the cost of the lowest-cost path found to a goal so far.
- What if the search encounters a path p such that $cost(p) + h(p) \geq bound$?
  p can be pruned.
- What can we do if a non-pruned path to a goal is found?
  bound can be set to the cost of p, and p can be remembered as the best solution so far.
- Why should this use a depth-first search?
  Uses linear space.
- What can be guaranteed when the search completes?

Depth-first Branch-and-Bound

Suppose we want to find a single optimal solution.

- Suppose *bound* is the cost of the lowest-cost path found to a goal so far.
- What if the search encounters a path p such that $cost(p) + h(p) \geq bound$?
  p can be pruned.
- What can we do if a non-pruned path to a goal is found?
  bound can be set to the cost of p, and p can be remembered as the best solution so far.
- Why should this use a depth-first search?
  Uses linear space.
- What can be guaranteed when the search completes?
  It has found an optimal solution.

Depth-first Branch-and-Bound

Suppose we want to find a single optimal solution.

- Suppose *bound* is the cost of the lowest-cost path found to a goal so far.
- What if the search encounters a path p such that $cost(p) + h(p) \geq bound$?
  p can be pruned.
- What can we do if a non-pruned path to a goal is found?
  bound can be set to the cost of p, and p can be remembered as the best solution so far.
- Why should this use a depth-first search?
  Uses linear space.
- What can be guaranteed when the search completes?
  It has found an optimal solution.
- How should the bound be initialized?

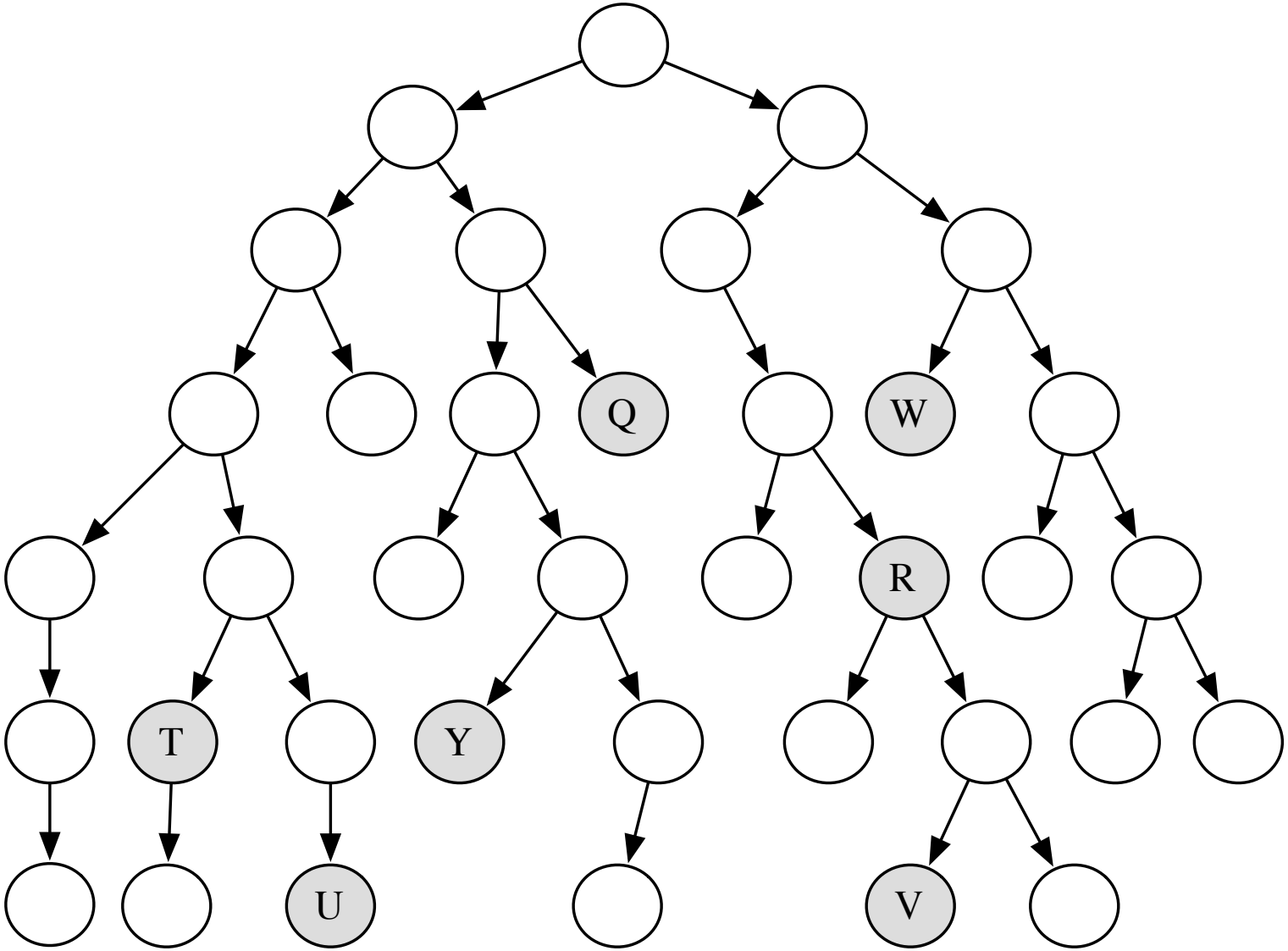
Depth-first Branch-and-Bound: Initializing Bound

- The bound can be initialized to ∞.
- The bound can be set to an estimate of the optimal path cost. After depth-first search terminates either:

Depth-first Branch-and-Bound: Initializing Bound

- The bound can be initialized to ∞.
- The bound can be set to an estimate of the optimal path cost. After depth-first search terminates either:
  - A solution was found.
  - No solution was found, and no path was pruned
  - No solution was found, and a path was pruned.

Which shaded goals will be best solutions so far?



Learning Objectives

At the end of the class you should be able to:

- recognize and represent constraint satisfaction problems
- count how big the search space is

Posing a Constraint Satisfaction Problem

A CSP is characterized by

- A set of variables $V_1, V_2, \ldots, V_n$.
- Each variable V_i has an associated domain $dom(V_i)$ which specifies the set of possible values the variable can take. (We assume domains are finite.)
- A possible world or total assignment is an assignment of a value to each variable.

Posing a Constraint Satisfaction Problem

A CSP is characterized by

- A set of variables $V_1, V_2, \ldots, V_n$.
- Each variable V_i has an associated domain $dom(V_i)$ which specifies the set of possible values the variable can take. (We assume domains are finite.)
- A possible world or total assignment is an assignment of a value to each variable.
- A hard constraint on a subset of variables specifies which combinations of values are legal. The legal assignments are said to satisfy the constraint.

Posing a Constraint Satisfaction Problem

A CSP is characterized by

- A set of variables $V_1, V_2, \ldots, V_n$.
- Each variable V_i has an associated domain $dom(V_i)$ which specifies the set of possible values the variable can take. (We assume domains are finite.)
- A possible world or total assignment is an assignment of a value to each variable.
- A hard constraint on a subset of variables specifies which combinations of values are legal. The legal assignments are said to satisfy the constraint.
- A solution to CSP (a model) is possible world that satisfies all the constraints.

Example: Map colouring

- Assign a colour (red, green, or blue) to each state so neighbouring states have different colours.
- What are the variables?
- What are the domains?

Example: Map colouring

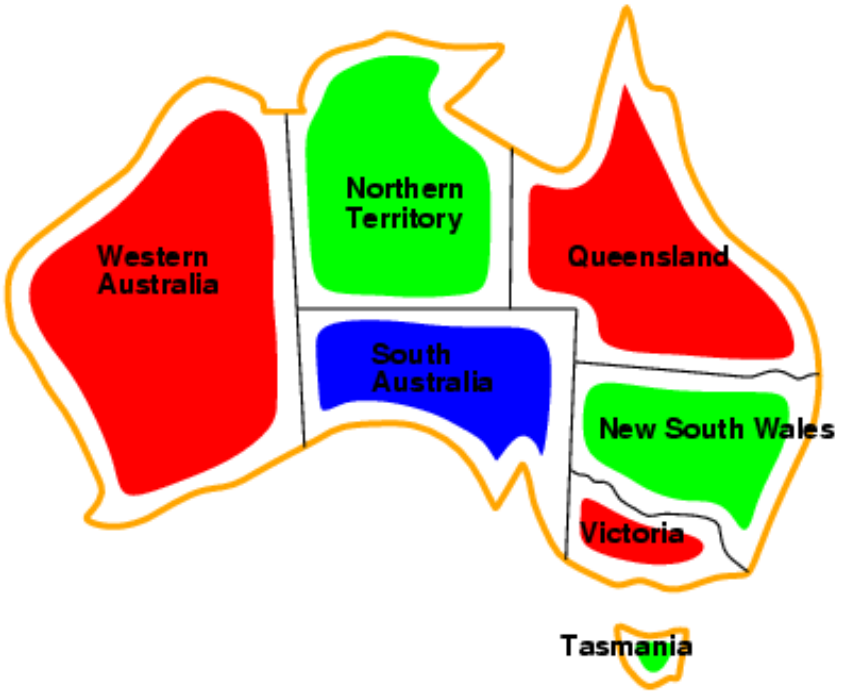
- Assign a colour (red, green, or blue) to each state so neighbouring states have different colours.
- What are the variables?
- What are the domains?
- How many possible worlds are there?

Example: Map colouring

- Assign a colour (red, green, or blue) to each state so neighbouring states have different colours.
- What are the variables?
- What are the domains?
- How many possible worlds are there?
- What are the constraints?

Example: Map colouring

Possible solution.



Simple Examples

Example 1:

- Variables: A, B, C
- Domains: $\{1, 2, 3, 4\}$
- Constraints $A < B$, $B < C$

Simple Examples

Example 1:

- Variables: A, B, C
- Domains: $\{1, 2, 3, 4\}$
- Constraints $A < B$, $B < C$

Example 2:

- Variables: A, B, C, D
- Domains: $\{1, 2, 3, 4\}$
- Constraints $A < B$, $B < C, C < D$

Simple Examples

Example 1:

- Variables: A, B, C
- Domains: $\{1, 2, 3, 4\}$
- Constraints $A < B$, $B < C$

Example 2:

- Variables: A, B, C, D
- Domains: $\{1, 2, 3, 4\}$
- Constraints $A < B$, $B < C, C < D$

Example 3:

- Variables: A, B, C, D, E
- Domains: $\{1, 2, 3, 4\}$
- Constraints $A < B$, $B < C, C < D, D < E$

CSP variants

- determine whether or not a model exists

CSP variants

- determine whether or not a model exists
- find a model

CSP variants

- determine whether or not a model exists
- find a model
- find all models

CSP variants

- determine whether or not a model exists
- find a model
- find all models
- count the number of models

CSP variants

- determine whether or not a model exists
- find a model
- find all models
- count the number of models
- find the best model given some model quality
  - soft constraints specify preferences

CSP variants

- determine whether or not a model exists
- find a model
- find all models
- count the number of models
- find the best model given some model quality
  - soft constraints specify preferences
- determine whether some property holds in all of the models

Example: scheduling activities

- Variables: A, B, C, D, E that represent the starting times of various activities.
- Domains: $dom(A) = \{1, 2, 3, 4\}$, $dom(B) = \{1, 2, 3, 4\}$, $dom(C) = \{1, 2, 3, 4\}$, $dom(D) = \{1, 2, 3, 4\}$, $dom(E) = \{1, 2, 3, 4\}$

Example: scheduling activities

- Variables: A, B, C, D, E that represent the starting times of various activities.
- Domains: $dom(A) = \{1, 2, 3, 4\}$, $dom(B) = \{1, 2, 3, 4\}$, $dom(C) = \{1, 2, 3, 4\}$, $dom(D) = \{1, 2, 3, 4\}$, $dom(E) = \{1, 2, 3, 4\}$
- What are some possible worlds?

Example: scheduling activities

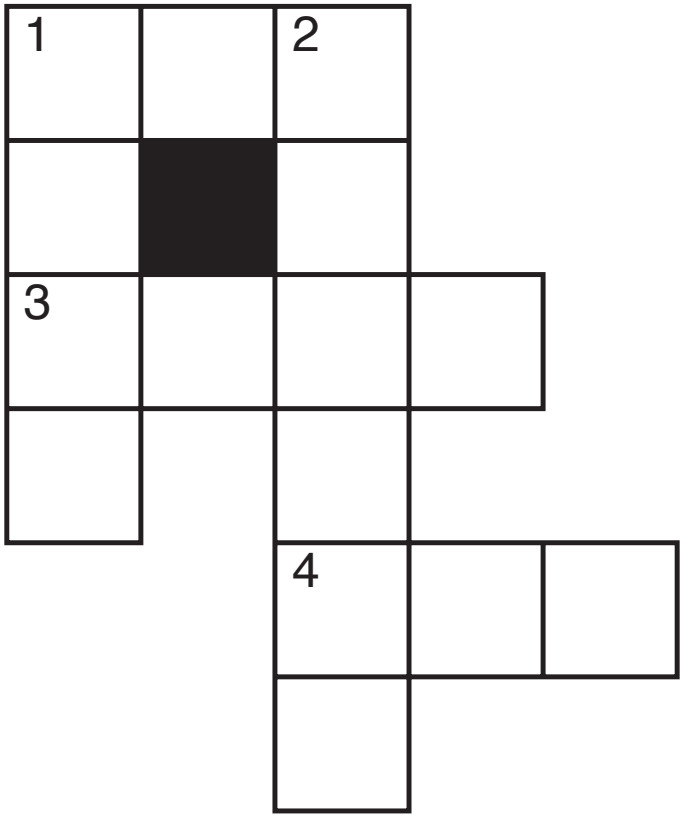
- Variables: A, B, C, D, E that represent the starting times of various activities.
- Domains: $dom(A) = \{1, 2, 3, 4\}$, $dom(B) = \{1, 2, 3, 4\}$, $dom(C) = \{1, 2, 3, 4\}$, $dom(D) = \{1, 2, 3, 4\}$, $dom(E) = \{1, 2, 3, 4\}$
- What are some possible worlds?
- How many possible worlds are there?

Example: scheduling activities

- Variables: A, B, C, D, E that represent the starting times of various activities.
- Domains: $dom(A) = \{1, 2, 3, 4\}$, $dom(B) = \{1, 2, 3, 4\}$, $dom(C) = \{1, 2, 3, 4\}$, $dom(D) = \{1, 2, 3, 4\}$, $dom(E) = \{1, 2, 3, 4\}$
- What are some possible worlds?
- How many possible worlds are there?
- Constraints:

$$
\begin{aligned}
&(B \neq 3) \wedge (C \neq 2) \wedge (A \neq B) \wedge (B \neq C) \wedge \\
&\quad (C < D) \wedge (A = D) \wedge (E < A) \wedge (E < B) \wedge \\
&\quad (E < C) \wedge (E < D) \wedge (B \neq D).
\end{aligned}
$$

Example: Crossword Puzzle

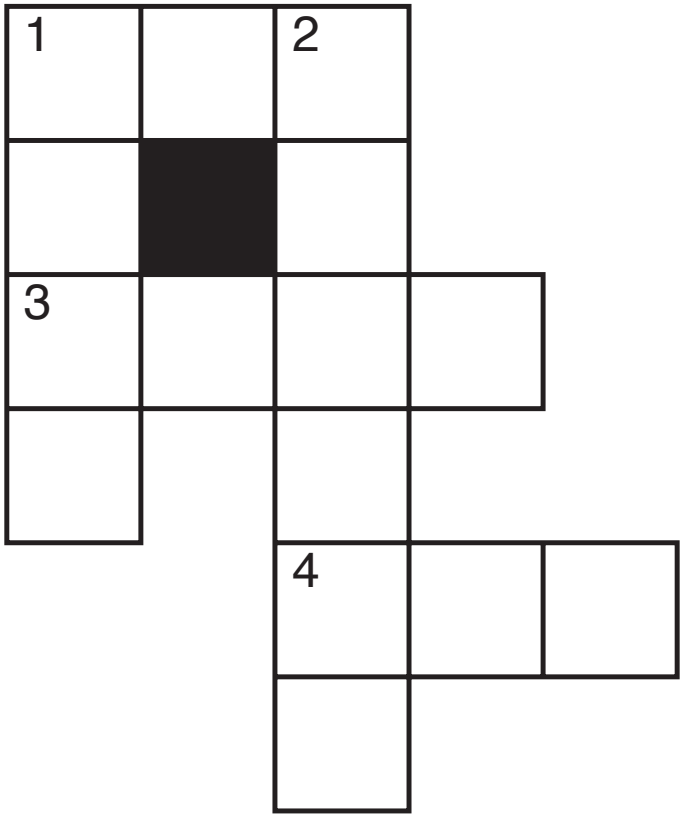


Words:

ant, big, bus, car, has
book, buys, hold,
lane, year
beast, ginger, search,
symbol, syntax

- What are the variables?
- What are their domains?

Example: Crossword Puzzle

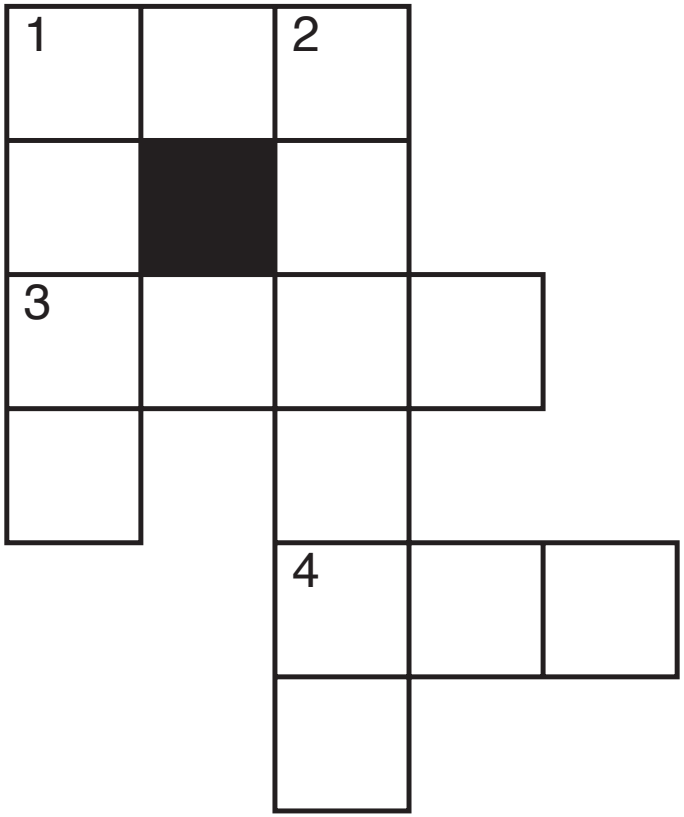


Words:

ant, big, bus, car, has
book, buys, hold,
lane, year
beast, ginger, search,
symbol, syntax

- What are the variables?
- What are their domains?
- How many possible worlds are there?

Example: Crossword Puzzle

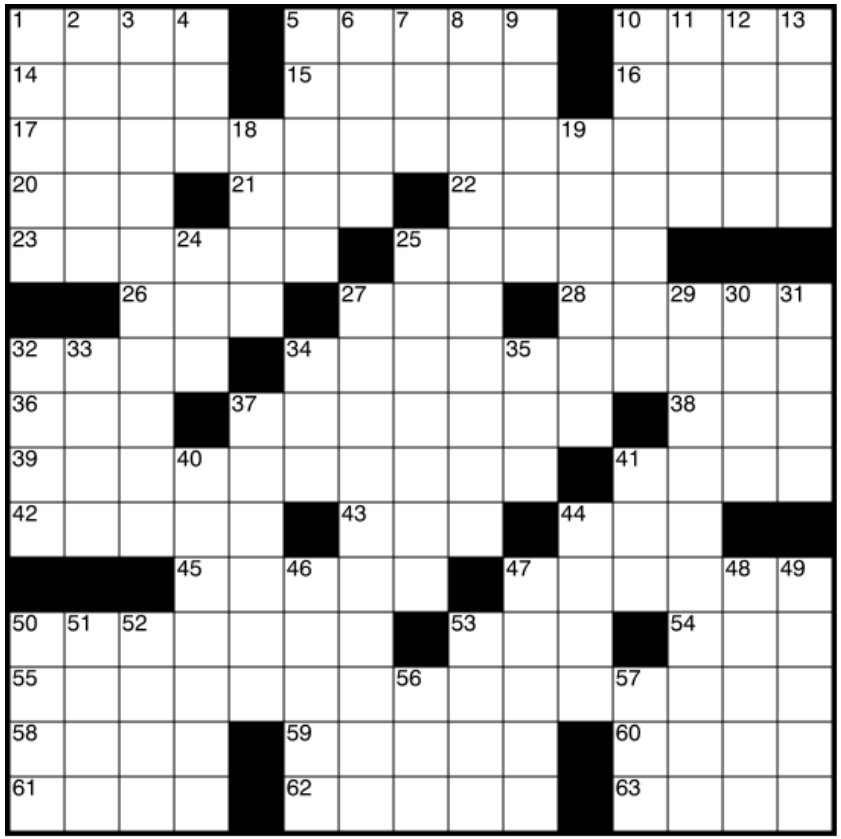


Words:

ant, big, bus, car, has
book, buys, hold,
lane, year
beast, ginger, search,
symbol, syntax

- What are the variables?
- What are their domains?
- How many possible worlds are there?
- What are the constraints?

Example: Crossword Puzzle



Suppose there are 10,000 words of each length (from 2 to 10).

- How many possible worlds are there?

Example: Sodoku

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

- What are the variables?

Example: Sodoku

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

- What are the variables?
- What is their domain?

Example: Sodoku

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

- What are the variables?
- What is their domain?
- How many possible worlds are there?

Example: Sodoku

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

- What are the variables?
- What is their domain?
- How many possible worlds are there?
- What are the constraints?

Hard and Soft Constraints

- Given a set of variables, assign a value to each variable that either
  - satisfies some set of constraints: satisfiability problems — "hard constraints"
  - minimizes some cost function, where each assignment of values to variables has some cost: optimization problems — "soft constraints"
- Many problems are a mix of hard and soft constraints (called constrained optimization problems).

Scheduling final exams

UBC exam scheduling is done by an AI system:

- 13 exam days, 52 timeslots
- 30,000 students take exams
- 1,700 sections with exams
- 105,000 student-exam pairs
- 274 rooms across 38 buildings

Scheduling final exams

UBC exam scheduling is done by an AI system:

- 13 exam days, 52 timeslots
- 30,000 students take exams
- 1,700 sections with exams
- 105,000 student-exam pairs
- 274 rooms across 38 buildings

- What are the variables?

Scheduling final exams

UBC exam scheduling is done by an AI system:

- 13 exam days, 52 timeslots
- 30,000 students take exams
- 1,700 sections with exams
- 105,000 student-exam pairs
- 274 rooms across 38 buildings

- What are the variables?
- What are the domains?

Scheduling final exams

UBC exam scheduling is done by an AI system:

- 13 exam days, 52 timeslots
- 30,000 students take exams
- 1,700 sections with exams
- 105,000 student-exam pairs
- 274 rooms across 38 buildings

- What are the variables?
- What are the domains?
- How many possible worlds are there?

Scheduling final exams

UBC exam scheduling is done by an AI system:

- 13 exam days, 52 timeslots
- 30,000 students take exams
- 1,700 sections with exams
- 105,000 student-exam pairs
- 274 rooms across 38 buildings

- What are the variables?
- What are the domains?
- How many possible worlds are there?
- What are the constraints?

UBC Exam Scheduling Hard Constraints

- There can't be more than 30 conflicts for a section

UBC Exam Scheduling Hard Constraints

- There can't be more than 30 conflicts for a section
- Allowable times for each exam

UBC Exam Scheduling Hard Constraints

- There can't be more than 30 conflicts for a section
- Allowable times for each exam
- Allowable rooms for each exam

UBC Exam Scheduling Hard Constraints

- There can't be more than 30 conflicts for a section
- Allowable times for each exam
- Allowable rooms for each exam
- Requested room features for each exam

UBC Exam Scheduling Hard Constraints

- There can't be more than 30 conflicts for a section
- Allowable times for each exam
- Allowable rooms for each exam
- Requested room features for each exam
- Unrelated exams cannot share a room

UBC Exam Scheduling Hard Constraints

- There can't be more than 30 conflicts for a section
- Allowable times for each exam
- Allowable rooms for each exam
- Requested room features for each exam
- Unrelated exams cannot share a room
- Cross-listed courses must have the same exam time

UBC Exam Scheduling Hard Constraints

- There can't be more than 30 conflicts for a section
- Allowable times for each exam
- Allowable rooms for each exam
- Requested room features for each exam
- Unrelated exams cannot share a room
- Cross-listed courses must have the same exam time
- Evening courses must have evening exams

UBC Exam Scheduling Soft Constraints

Try to minimize:

- Conflicts

UBC Exam Scheduling Soft Constraints

Try to minimize:

- Conflicts
- Students with 2+ exams on the same day

UBC Exam Scheduling Soft Constraints

Try to minimize:

- Conflicts
- Students with 2+ exams on the same day
- Students with 3+ exams in 4 consecutive timeslots

UBC Exam Scheduling Soft Constraints

Try to minimize:

- Conflicts
- Students with 2+ exams on the same day
- Students with 3+ exams in 4 consecutive timeslots
- Students with back-to-back exams

UBC Exam Scheduling Soft Constraints

Try to minimize:

- Conflicts
- Students with 2+ exams on the same day
- Students with 3+ exams in 4 consecutive timeslots
- Students with back-to-back exams
- Students with less than 8 timeslots between exams

UBC Exam Scheduling Soft Constraints

Try to minimize:

- Conflicts
- Students with 2+ exams on the same day
- Students with 3+ exams in 4 consecutive timeslots
- Students with back-to-back exams
- Students with less than 8 timeslots between exams
- Preferred times for each exam

UBC Exam Scheduling Soft Constraints

Try to minimize:

- Conflicts
- Students with 2+ exams on the same day
- Students with 3+ exams in 4 consecutive timeslots
- Students with back-to-back exams
- Students with less than 8 timeslots between exams
- Preferred times for each exam
- Preferred rooms for each exam

UBC Exam Scheduling Soft Constraints

Try to minimize:

- Conflicts
- Students with 2+ exams on the same day
- Students with 3+ exams in 4 consecutive timeslots
- Students with back-to-back exams
- Students with less than 8 timeslots between exams
- Preferred times for each exam
- Preferred rooms for each exam
- Room capacities

UBC Exam Scheduling Soft Constraints

Try to minimize:

- Conflicts
- Students with 2+ exams on the same day
- Students with 3+ exams in 4 consecutive timeslots
- Students with back-to-back exams
- Students with less than 8 timeslots between exams
- Preferred times for each exam
- Preferred rooms for each exam
- Room capacities
- First-year exams on the last two days (Fall exams)
- Fourth-year exams on the last two days (Spring exams)

Learning Objectives

At the end of the class you should be able to:

- show how constraint satisfaction problems can be solved with generate-and-test
- show how constraint satisfaction problems can be solved with search
- explain and trace arc-consistency of a constraint graph
- show how domain splitting can solve constraint problems

Generate-and-Test Algorithm

- Generate the assignment space $\mathrm{D} = dom(V_1) \times dom(V_2) \times \ldots \times dom(V_n)$. Test each assignment with the constraints.
- Example:

$$\begin{aligned}
\mathrm{D} &= dom(A) \times dom(B) \times dom(C) \times dom(D) \times dom(E) \\
&= \{1,2,3,4\} \times \{1,2,3,4\} \times \{1,2,3,4\} \times \{1,2,3,4\} \times \{1,2,3, \\
&= \{\langle 1,1,1,1,1\rangle, \langle 1,1,1,1,2\rangle, ..., \langle 4,4,4,4,4\rangle\}.
\end{aligned}$$

Generate-and-Test Algorithm

- Generate the assignment space $\mathrm{D} = dom(V_1) \times dom(V_2) \times \ldots \times dom(V_n)$. Test each assignment with the constraints.
- Example:

$$\begin{aligned}
\mathrm{D} &= dom(A) \times dom(B) \times dom(C) \times dom(D) \times dom(E) \\
&= \{1,2,3,4\} \times \{1,2,3,4\} \times \{1,2,3,4\} \times \{1,2,3,4\} \times \{1,2,3, \\
&= \{\langle 1,1,1,1,1\rangle, \langle 1,1,1,1,2\rangle, ..., \langle 4,4,4,4,4\rangle\}.
\end{aligned}$$

- Can be implemented with n nested for-loops.

Generate-and-Test Algorithm

- Generate the assignment space $\mathrm{D} = dom(V_1) \times dom(V_2) \times \ldots \times dom(V_n)$. Test each assignment with the constraints.
- Example:

$$\begin{aligned}
\mathrm{D} &= dom(A) \times dom(B) \times dom(C) \times dom(D) \times dom(E) \\
&= \{1,2,3,4\} \times \{1,2,3,4\} \times \{1,2,3,4\} \times \{1,2,3,4\} \times \{1,2,3, \\
&= \{\langle 1,1,1,1,1\rangle, \langle 1,1,1,1,2\rangle, ..., \langle 4,4,4,4,4\rangle\}.
\end{aligned}$$

- Can be implemented with n nested for-loops.

```
for A in dom_A:
   for B in dom_B:
      ...
       if constraints are satisfied: return (A,B,...)
```

Generate-and-Test Algorithm

- Generate the assignment space $\mathrm{D} = dom(V_1) \times dom(V_2) \times \ldots \times dom(V_n)$. Test each assignment with the constraints.
- Example:

$$\begin{aligned}
\mathrm{D} &= dom(A) \times dom(B) \times dom(C) \times dom(D) \times dom(E) \\
&= \{1,2,3,4\} \times \{1,2,3,4\} \times \{1,2,3,4\} \times \{1,2,3,4\} \times \{1,2,3, \\
&= \{\langle 1,1,1,1,1\rangle, \langle 1,1,1,1,2\rangle, ..., \langle 4,4,4,4,4\rangle\}.
\end{aligned}$$

- Can be implemented with n nested for-loops.

```
for A in dom_A:
   for B in dom_B:
      ...
       if constraints are satisfied: return (A,B,...)
```

- How many assignments need to be tested for n variables each with domain size d?

Backtracking Algorithms

- Systematically explore D by instantiating the variables one at a time

Backtracking Algorithms

- Systematically explore D by instantiating the variables one at a time
- evaluate each constraint predicate as soon as all its variables are bound

Backtracking Algorithms

- Systematically explore D by instantiating the variables one at a time
- evaluate each constraint predicate as soon as all its variables are bound
- any partial assignment that doesn't satisfy the constraint can be pruned.

Backtracking Algorithms

- Systematically explore D by instantiating the variables one at a time
- evaluate each constraint predicate as soon as all its variables are bound
- any partial assignment that doesn't satisfy the constraint can be pruned.

Example Variables A, B, C, domains $\{1, 2, 3, 4\}$, constraints $A < B, B < C$.

Backtracking Algorithms

- Systematically explore D by instantiating the variables one at a time
- evaluate each constraint predicate as soon as all its variables are bound
- any partial assignment that doesn't satisfy the constraint can be pruned.

Example Variables A, B, C, domains $\{1, 2, 3, 4\}$, constraints $A < B, B < C$.
Assignment $A = 1 \wedge B = 1$ is inconsistent with constraint $A < B$ regardless of the value of the other variables.

CSP as Graph Searching

A CSP can be solved by graph-searching:

- A node is an assignment values to some of the variables.

CSP as Graph Searching

A CSP can be solved by graph-searching:

- A node is an assignment values to some of the variables.
- Suppose node N is the assignment $X_1 = v_1, \ldots, X_k = v_k$.
  Select a variable Y that isn't assigned in N.
  For each value $y_i \in dom(Y)$
  $X_1 = v_1, \ldots, X_k = v_k, Y = y_i$ is a neighbour if it is consistent with the constraints that can be evaluated.

CSP as Graph Searching

A CSP can be solved by graph-searching:

- A node is an assignment values to some of the variables.
- Suppose node N is the assignment $X_1 = v_1, \ldots, X_k = v_k$.
  Select a variable Y that isn't assigned in N.
  For each value $y_i \in dom(Y)$
  $X_1 = v_1, \ldots, X_k = v_k, Y = y_i$ is a neighbour if it is consistent with the constraints that can be evaluated.
- The start node is

CSP as Graph Searching

A CSP can be solved by graph-searching:

- A node is an assignment values to some of the variables.
- Suppose node N is the assignment $X_1 = v_1, \ldots, X_k = v_k$.
  Select a variable Y that isn't assigned in N.
  For each value $y_i \in dom(Y)$
  $X_1 = v_1, \ldots, X_k = v_k, Y = y_i$ is a neighbour if it is consistent with the constraints that can be evaluated.
- The start node is the empty assignment.

CSP as Graph Searching

A CSP can be solved by graph-searching:

- A node is an assignment values to some of the variables.
- Suppose node N is the assignment $X_1 = v_1, \ldots, X_k = v_k$.
  Select a variable Y that isn't assigned in N.
  For each value $y_i \in dom(Y)$
  $X_1 = v_1, \ldots, X_k = v_k, Y = y_i$ is a neighbour if it is consistent with the constraints that can be evaluated.
- The start node is the empty assignment.
- A goal node is a

CSP as Graph Searching

A CSP can be solved by graph-searching:

- A node is an assignment values to some of the variables.
- Suppose node N is the assignment $X_1 = v_1, \ldots, X_k = v_k$.
  Select a variable Y that isn't assigned in N.
  For each value $y_i \in dom(Y)$
  $X_1 = v_1, \ldots, X_k = v_k, Y = y_i$ is a neighbour if it is consistent with the constraints that can be evaluated.
- The start node is the empty assignment.
- A goal node is a total assignment that satisfies the constraints.

CSP as Graph Searching

A CSP can be solved by graph-searching:

- A node is an assignment values to some of the variables.
- Suppose node N is the assignment $X_1 = v_1, \ldots, X_k = v_k$.
  Select a variable Y that isn't assigned in N.
  For each value $y_i \in dom(Y)$
  $X_1 = v_1, \ldots, X_k = v_k, Y = y_i$ is a neighbour if it is consistent with the constraints that can be evaluated.
- The start node is the empty assignment.
- A goal node is a total assignment that satisfies the constraints.
- The search space depends on which variable is selected to be assigned for each node. There are no cycles or multiple paths to a node.

Simple Example 1

- Variables: A, B, C
- Domains: $\{1, 2, 3, 4\}$
- Constraints $A < B$, $B < C$

Simple Example 2

- Variables: A, B, C, D
- Domains: $\{1, 2, 3, 4\}$
- Constraints $A < B$, $B < C$, $C < D$

Simple Example 3

- Variables: A, B, C, D, E
- Domains: $\{1, 2, 3, 4\}$
- Constraints $A < B$, $B < C, C < D, D < E$

Example: scheduling activities

- Variables: A, B, C, D, E that represent the starting times of various activities.
- Domains: $dom(A) = \{1, 2, 3, 4\}$, $dom(B) = \{1, 2, 3, 4\}$, $dom(C) = \{1, 2, 3, 4\}$, $dom(D) = \{1, 2, 3, 4\}$, $dom(E) = \{1, 2, 3, 4\}$
- Constraints:

$$
\begin{aligned}
&(B \neq 3) \wedge (C \neq 2) \wedge (A \neq B) \wedge (B \neq C) \wedge \\
&\quad (C < D) \wedge (A = D) \wedge (E < A) \wedge (E < B) \wedge \\
&\quad (E < C) \wedge (E < D) \wedge (B \neq D).
\end{aligned}
$$

Consistency Algorithms

- Idea: prune the domains as much as possible before selecting values from them.
- A variable is domain consistent if no value of the domain of the variable is ruled impossible by any of the constraints.
- Example: Is the scheduling example domain consistent?

Consistency Algorithms

- Idea: prune the domains as much as possible before selecting values from them.
- A variable is domain consistent if no value of the domain of the variable is ruled impossible by any of the constraints.
- Example: Is the scheduling example domain consistent? $dom(B) = \{1, 2, 3, 4\}$ isn't domain consistent as $B = 3$ violates the constraint $B \neq 3$.

Constraint Network

- There is a oval-shaped node for each variable.

Constraint Network

- There is a oval-shaped node for each variable.
- There is a rectangular node for each constraint.

Constraint Network

- There is a oval-shaped node for each variable.
- There is a rectangular node for each constraint.
- There is a domain of values associated with each variable node.

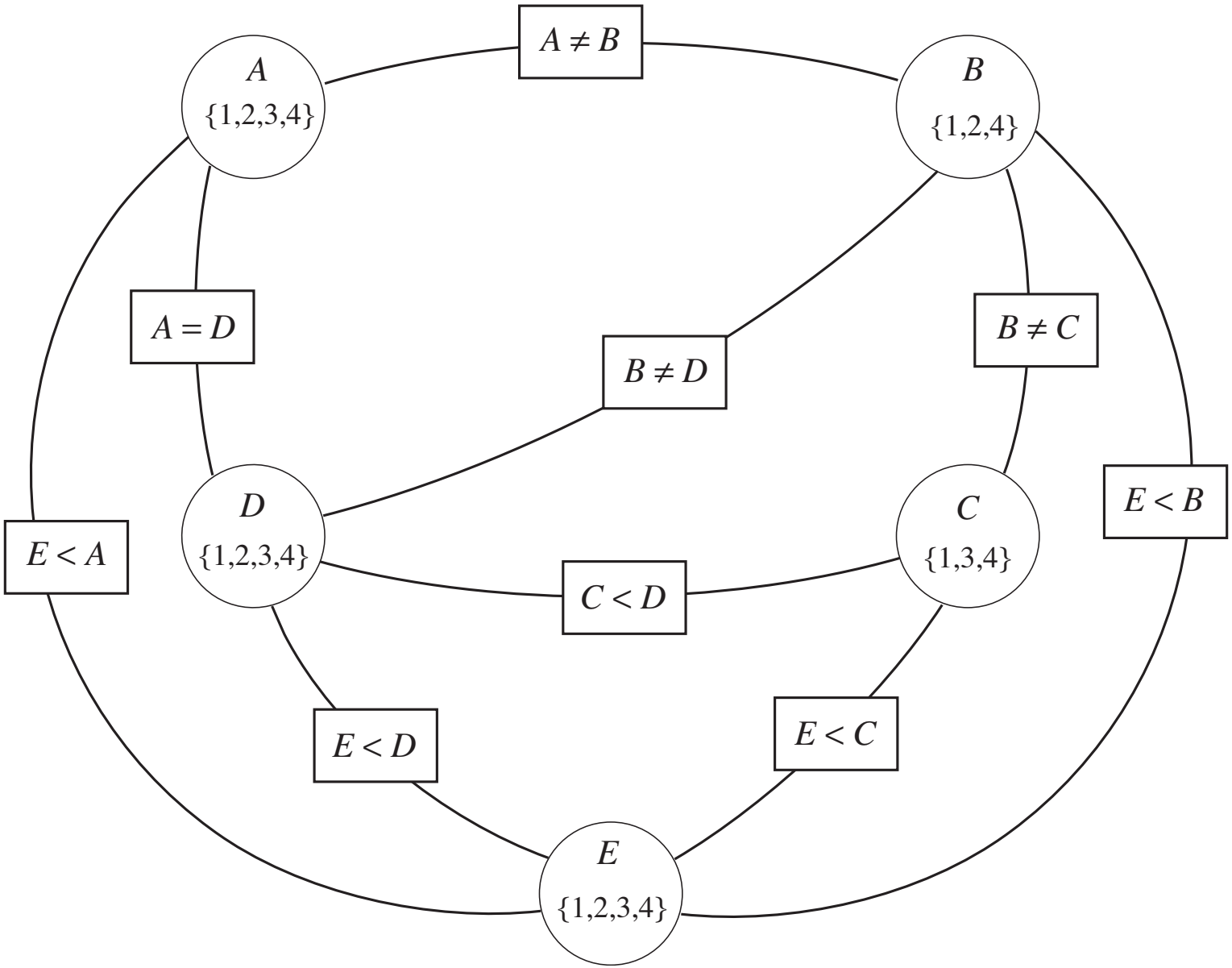
Constraint Network

- There is a oval-shaped node for each variable.
- There is a rectangular node for each constraint.
- There is a domain of values associated with each variable node.
- There is an arc from variable X to each constraint that involves X.

Constraint Network

- There is a oval-shaped node for each variable.
- There is a rectangular node for each constraint.
- There is a domain of values associated with each variable node.
- There is an arc from variable X to each constraint that involves X.
  An arc is written as $\langle X, r(X, \overline{Y}) \rangle$
  E.g., $\langle X, X < Y \rangle$, $\langle Y, X < Y \rangle$
  $\langle X, X + Y = Z \rangle$, $\langle Y, X + Y = Z \rangle$, $\langle Z, X + Y = Z \rangle$

Example Constraint Network



Arc Consistency

- An arc $\langle X, r(X, \overline{Y}) \rangle$ is arc consistent if, for each value $x \in dom(X)$, there is some value $\overline{y} \in dom(\overline{Y})$ such that $r(x, \overline{y})$ is satisfied.

Arc Consistency

- An arc $\langle X, r(X, \overline{Y}) \rangle$ is arc consistent if, for each value $x \in dom(X)$, there is some value $\overline{y} \in dom(\overline{Y})$ such that $r(x, \overline{y})$ is satisfied.
- A network is arc consistent if all its arcs are arc consistent.
- What if arc $\langle X, r(X, \overline{Y}) \rangle$ is *not* arc consistent?

Arc Consistency

- An arc $\langle X, r(X, \overline{Y}) \rangle$ is arc consistent if, for each value $x \in dom(X)$, there is some value $\overline{y} \in dom(\overline{Y})$ such that $r(x, \overline{y})$ is satisfied.
- A network is arc consistent if all its arcs are arc consistent.
- What if arc $\langle X, r(X, \overline{Y}) \rangle$ is *not* arc consistent?
  All values of X in $dom(X)$ for which there is no corresponding value in $dom(\overline{Y})$ can be deleted from $dom(X)$ to make the arc $\langle X, r(X, \overline{Y}) \rangle$ consistent.

Arc Consistency Algorithm

- The arcs can be considered in turn making each arc consistent.
- When an arc has been made arc consistent, does it ever need to be checked again?

Arc Consistency Algorithm

- The arcs can be considered in turn making each arc consistent.
- When an arc has been made arc consistent, does it ever need to be checked again?
  An arc $\langle X, r(X, \overline{Y}) \rangle$ needs to be revisited if the domain of one of the Y's is reduced.

Generalized Arc Consistency

for each variable X:
    $D_X := dom(X)$
$TDA := \{\langle X, c\rangle \mid c \in C \text{ and } X \in scope(c)\}$
while TDA is not empty:
    select and **remove** path $\langle X, c\rangle$ from TDA
    suppose scope of c is $\{X, Y_1, \ldots, Y_k\}$
    $ND_X := \{x \mid x \in D_X$ and
        exists $y_1 \in D_{Y_1}, \ldots, y_k \in D_{Y_k}$
        s.th. $c(X = x, Y_1 = y_1, \ldots, Y_k = y_k) = true\ \}$
    if $ND_X \neq D_X$:
        $TDA := TDA \cup \{\langle Z, c'\rangle \mid X \in scope(c'),$
            c' is not c, $Z \in scope(c') \setminus \{X\}\}$
        $D_X := ND_X$
return $\{D_X \mid X \text{ is a variable}\}$

Arc Consistency Algorithm

Three possible outcomes when all arcs are made arc consistent:

Arc Consistency Algorithm

Three possible outcomes when all arcs are made arc consistent:

- One domain is empty $\Longrightarrow$
- Each domain has a single value $\Longrightarrow$
- Some domains have more than one value $\Longrightarrow$

Arc Consistency Algorithm

Three possible outcomes when all arcs are made arc consistent:

- One domain is empty $\Longrightarrow$ no solution
- Each domain has a single value $\Longrightarrow$
- Some domains have more than one value $\Longrightarrow$

Arc Consistency Algorithm

Three possible outcomes when all arcs are made arc consistent:

- One domain is empty $\Longrightarrow$ no solution
- Each domain has a single value $\Longrightarrow$ unique solution
- Some domains have more than one value $\Longrightarrow$

Arc Consistency Algorithm

Three possible outcomes when all arcs are made arc consistent:

- One domain is empty $\Longrightarrow$ no solution
- Each domain has a single value $\Longrightarrow$ unique solution
- Some domains have more than one value $\Longrightarrow$ there may or may not be a solution

Complexity of Arc Consistency

- Consider binary constraints
- Each variable domain is of size d
- There are e arcs.
- Checking an arc takes time

Complexity of Arc Consistency

- Consider binary constraints
- Each variable domain is of size d
- There are e arcs.
- Checking an arc takes time $O(d^2)$
  $\langle X, c(X, Y)\rangle$ for each value for X, check each value for Y
- Each constraint needs to be checked at most

Complexity of Arc Consistency

- Consider binary constraints
- Each variable domain is of size d
- There are e arcs.
- Checking an arc takes time $O(d^2)$
  $\langle X, c(X, Y)\rangle$ for each value for X, check each value for Y
- Each constraint needs to be checked at most d times.
  $\langle X, c(X, Y)\rangle$ rechecked when a value for Y is removed.
- Thus the algorithm GAC takes time

Complexity of Arc Consistency

- Consider binary constraints
- Each variable domain is of size d
- There are e arcs.
- Checking an arc takes time $O(d^2)$
  $\langle X, c(X, Y)\rangle$ for each value for X, check each value for Y
- Each constraint needs to be checked at most d times.
  $\langle X, c(X, Y)\rangle$ rechecked when a value for Y is removed.
- Thus the algorithm GAC takes time $O(ed^3)$.

Complexity of Arc Consistency

- Consider binary constraints
- Each variable domain is of size d
- There are e arcs.
- Checking an arc takes time $O(d^2)$
  $\langle X, c(X, Y)\rangle$ for each value for X, check each value for Y
- Each constraint needs to be checked at most d times.
  $\langle X, c(X, Y)\rangle$ rechecked when a value for Y is removed.
- Thus the algorithm GAC takes time $O(ed^3)$.

Solving a CSP is an NP-complete problem where n the number of variables

- Give a solution it can be checked in polynomial time
- But it can be made arc consistent in polynomial time. How?

Complexity of Arc Consistency

- Consider binary constraints
- Each variable domain is of size d
- There are e arcs.
- Checking an arc takes time $O(d^2)$
  $\langle X, c(X, Y)\rangle$ for each value for X, check each value for Y
- Each constraint needs to be checked at most d times.
  $\langle X, c(X, Y)\rangle$ rechecked when a value for Y is removed.
- Thus the algorithm GAC takes time $O(ed^3)$.

Solving a CSP is an NP-complete problem where n the number of variables

- Give a solution it can be checked in polynomial time
- But it can be made arc consistent in polynomial time. How? Making the network arc consistent does not solve the problem. We need to search for a solution.

Finding solutions with AC and domain splitting

To solve a CSP:

- Simplify with arc-consistency
- If a domain is empty, return no solution
- If all domains have size 1, return solution found
- Else split a domain, and recursively solve each half.

Finding one solutions with AC and domain splitting

$Solve_one(CSP, domains)$:

simplify CSP with arc-consistency

if one domain is empty:

return False

else if all domains have one element:

return solution of that element for each variable

else:

select variable X with domain D and $|D| > 1$

partition D into D_1 and D_2

return $Solve_one(CSP, domains$ with $dom(X) = D_1)$ or
$Solve_one(CSP, domains$ with $dom(X) = D_2)$

Finding set of all solutions with AC and domain splitting

Solve_all(*CSP*, *domains*) :
    simplify *CSP* with arc-consistency
    if one domain is empty:
        return
    else if all domains have one element:
        return
    else:
        select variable X with domain D and $|D| > 1$
        partition D into D_1 and D_2
        return

Finding set of all solutions with AC and domain splitting

$Solve_all(CSP, domains)$:

simplify CSP with arc-consistency

if one domain is empty:

return $\{\}$

else if all domains have one element:

return

else:

select variable X with domain D and $|D| > 1$

partition D into D_1 and D_2

return

Finding set of all solutions with AC and domain splitting

Solve_all(*CSP*, *domains*) :

simplify *CSP* with arc-consistency

if one domain is empty:

return $\{\}$

else if all domains have one element:

return {solution of that element for each variable}

else:

select variable X with domain D and $|D| > 1$

partition D into D_1 and D_2

return

Finding set of all solutions with AC and domain splitting

$Solve_all(CSP, domains)$:

simplify CSP with arc-consistency

if one domain is empty:

return $\{\}$

else if all domains have one element:

return $\{$solution of that element for each variable$\}$

else:

select variable X with domain D and $|D| > 1$

partition D into D_1 and D_2

return $Solve_all(CSP, domains \text{ with } dom(X) = D_1) \cup$
$Solve_all(CSP, domains \text{ with } dom(X) = D_2)$

AC and domain splitting as search

Domain splitting leads to search space

- Nodes:
- Neighbors

- Goal:
- Start node:

AC and domain splitting as search

Domain splitting leads to search space

- Nodes: CSP with arc-consistent domains
- Neighbors

- Goal:
- Start node:

AC and domain splitting as search

Domain splitting leads to search space

- Nodes: CSP with arc-consistent domains
- Neighbors of CSP:
  if all domains are non-empty:
  select variable X with domain D and $|D| > 1$
  partition D into D_1 and D_2
  neighbors are
  - $make_AC(CSP \mid dom(X) = D_1)$
  - $make_AC(CSP \mid dom(X) = D_2)$
- Goal:
- Start node:

AC and domain splitting as search

Domain splitting leads to search space

- Nodes: CSP with arc-consistent domains
- Neighbors of CSP:
  if all domains are non-empty:
  select variable X with domain D and $|D| > 1$
  partition D into D_1 and D_2
  neighbors are
  - $make_AC(CSP \mid dom(X) = D_1)$
  - $make_AC(CSP \mid dom(X) = D_2)$
- Goal: all domains have size 1
- Start node:

AC and domain splitting as search

Domain splitting leads to search space

- Nodes: CSP with arc-consistent domains
- Neighbors of CSP:
  if all domains are non-empty:
  select variable X with domain D and $|D| > 1$
  partition D into D_1 and D_2
  neighbors are
  - $make_AC(CSP \mid dom(X) = D_1)$
  - $make_AC(CSP \mid dom(X) = D_2)$
- Goal: all domains have size 1
- Start node: $make_AC(CSP)$

Local Search

Local Search:

- Maintain a complete assignment of a value to each variable.
- Start with random assignment or a best guess.

Local Search

Local Search:

- Maintain a complete assignment of a value to each variable.
- Start with random assignment or a best guess.
- Repeat:
  - Select a variable to change
  - Select a new value for that variable
- Until a satisfying assignment is found

Local Search for CSPs

- Aim: find an assignment with zero unsatisfied constraints.
- Given an assignment of a value to each variable, a conflict is an unsatisfied constraint.

Local Search for CSPs

- Aim: find an assignment with zero unsatisfied constraints.
- Given an assignment of a value to each variable, a conflict is an unsatisfied constraint.
- The goal is an assignment with zero conflicts.

Local Search for CSPs

- Aim: find an assignment with zero unsatisfied constraints.
- Given an assignment of a value to each variable, a conflict is an unsatisfied constraint.
- The goal is an assignment with zero conflicts.
- Function to be minimized: the number of conflicts.

Iterative Best Improvement (2 stage) "greedy descent"

- Start with random assignment (for each variable, select a value for that variable at random)
- Repeat:
  - Select a variable that participates in the most conflicts
  - Select a different value for that variable
- Until a satisfying assignment is found

All selections are random and uniform.

Any Conflict

- Start with random assignment (for each variable, select a value for that variable at random)
- Repeat:
  - Select a variable at random that participates in any conflict
  - Select a different value for that variable
- Until a satisfying assignment is found

All selections are random and uniform.

Comparing Stochastic Algorithms

Which of the preceding algorithms work better?

Comparing Stochastic Algorithms

Which of the preceding algorithms work better?
How would we tell if one is better than the other?

Comparing Stochastic Algorithms

Which of the preceding algorithms work better?
How would we tell if one is better than the other?

- How can you compare three algorithms when
  - one solves the problem 30% of the time very quickly but doesn't halt for the other 70% of the cases
  - one solves 60% of the cases reasonably quickly but doesn't solve the rest
  - one solves the problem in 100% of the cases, but slowly?

Comparing Stochastic Algorithms

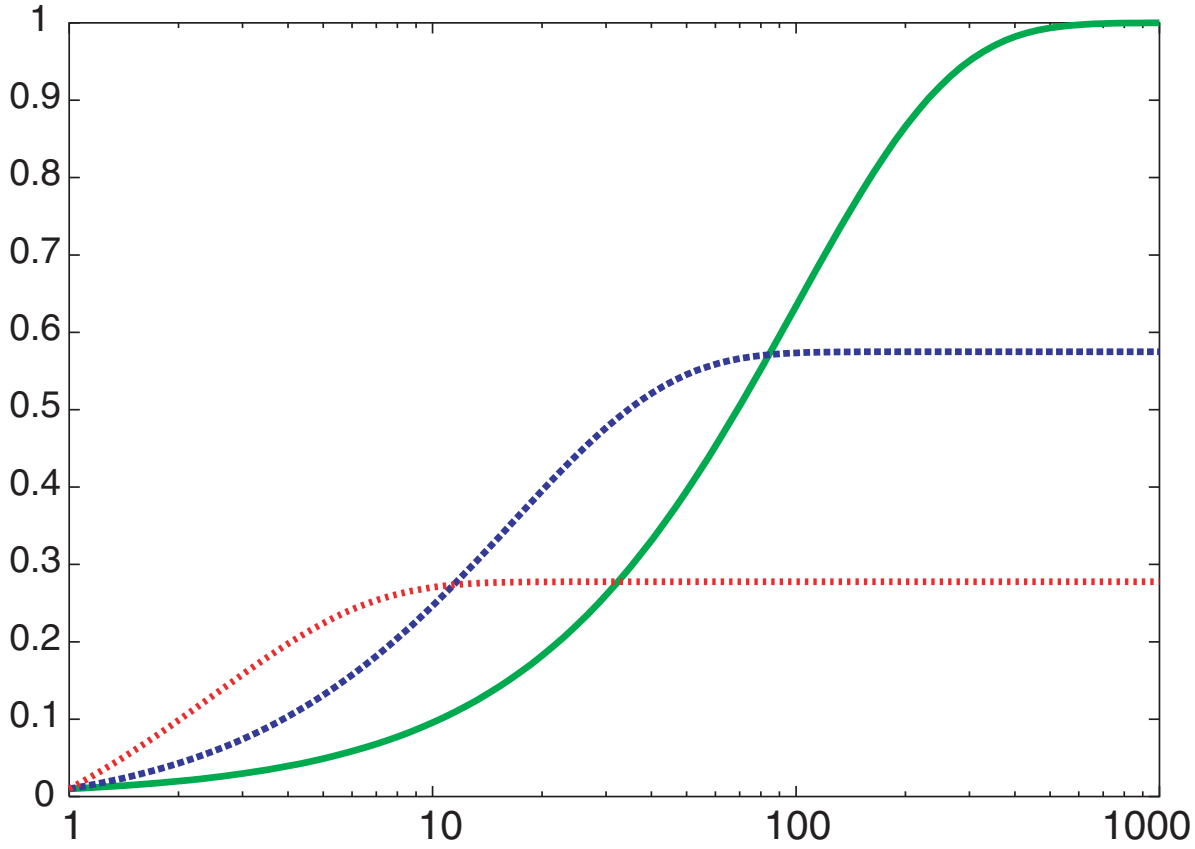
Which of the preceding algorithms work better?
How would we tell if one is better than the other?

- How can you compare three algorithms when
  - one solves the problem 30% of the time very quickly but doesn't halt for the other 70% of the cases
  - one solves 60% of the cases reasonably quickly but doesn't solve the rest
  - one solves the problem in 100% of the cases, but slowly?
- Summary statistics, such as mean run time, median run time, and mode run time don't make much sense.

Runtime Distribution

x-axis runtime (or number of steps)

y-axis the proportion (or number) of runs that are solved within that runtime



Runtime Distribution

- Run the same algorithm on the same instance for a number of trials (e.g., 100 or 1000)

Runtime Distribution

- Run the same algorithm on the same instance for a number of trials (e.g., 100 or 1000)
- Sort the trials according to the run time.

Runtime Distribution

- Run the same algorithm on the same instance for a number of trials (e.g., 100 or 1000)
- Sort the trials according to the run time.
- Plot:

  x-axis run time of the trial
  y-axis index of the trial

  This produces a cumulative distribution

Runtime Distribution

- Run the same algorithm on the same instance for a number of trials (e.g., 100 or 1000)
- Sort the trials according to the run time.
- Plot:

  x-axis run time of the trial
  y-axis index of the trial

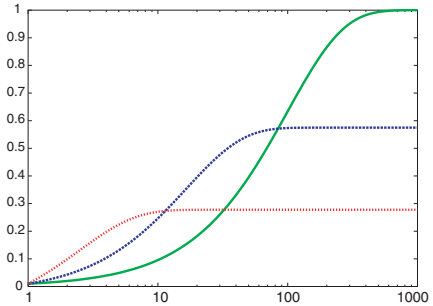
  This produces a cumulative distribution
- Do this this a few times to gauge the variability (take a statistics course!)

Runtime Distribution

- Run the same algorithm on the same instance for a number of trials (e.g., 100 or 1000)
- Sort the trials according to the run time.
- Plot:

  x-axis run time of the trial
  y-axis index of the trial

  This produces a cumulative distribution



- Do this this a few times to gauge the variability (take a statistics course!)
- Sometimes use number of steps instead of run time (because computers measure small run times inaccurately) . . . not good measure to compare algorithms if steps take different times

Randomized Algorithms

- A probabilistic mix of *greedy* and *any-conflict* — e.g., 70% of time pick best variable, otherwise pick any variable in a conflict – works better than either alone.

Stochastic Local Search

Stochastic local search is a mix of:

- Greedy descent: pick the best variable and/or value
- Random walk: picking variables and values at random
- Random restart: reassigning values to all variables

Some of these might be more complex than the others.
A probabilistic mix might work better.

Greedy Descent Variants

To select a variable to change and a new value for it:

- Most Improving Step: Find a variable-value pair that minimizes the number of conflicts.

Greedy Descent Variants

To select a variable to change and a new value for it:

- Most Improving Step: Find a variable-value pair that minimizes the number of conflicts.
  What data structures are required?

Greedy Descent Variants

To select a variable to change and a new value for it:

- Most Improving Step: Find a variable-value pair that minimizes the number of conflicts.
  What data structures are required?
- Two Stage Choice: Select a variable that participates in the most conflicts.
  Select a value that minimizes the number of conflicts.

Greedy Descent Variants

To select a variable to change and a new value for it:

- Most Improving Step: Find a variable-value pair that minimizes the number of conflicts.
  What data structures are required?
- Two Stage Choice: Select a variable that participates in the most conflicts.
  Select a value that minimizes the number of conflicts.
  What data structures are required?

Greedy Descent Variants

To select a variable to change and a new value for it:

- Most Improving Step: Find a variable-value pair that minimizes the number of conflicts.
  What data structures are required?
- Two Stage Choice: Select a variable that participates in the most conflicts.
  Select a value that minimizes the number of conflicts.
  What data structures are required?
- Any Conflict: Select a variable that appears in any conflict.
  Select a value at random.

Greedy Descent Variants

To select a variable to change and a new value for it:

- Most Improving Step: Find a variable-value pair that minimizes the number of conflicts.
  What data structures are required?
- Two Stage Choice: Select a variable that participates in the most conflicts.
  Select a value that minimizes the number of conflicts.
  What data structures are required?
- Any Conflict: Select a variable that appears in any conflict.
  Select a value at random.
  What data structures are required?

Greedy Descent Variants

To select a variable to change and a new value for it:

- Most Improving Step: Find a variable-value pair that minimizes the number of conflicts.
  What data structures are required?
- Two Stage Choice: Select a variable that participates in the most conflicts.
  Select a value that minimizes the number of conflicts.
  What data structures are required?
- Any Conflict: Select a variable that appears in any conflict.
  Select a value at random.
  What data structures are required?
- Select a variable at random.
  Select a value that minimizes the number of conflicts.

Greedy Descent Variants

To select a variable to change and a new value for it:

- Most Improving Step: Find a variable-value pair that minimizes the number of conflicts.
  What data structures are required?
- Two Stage Choice: Select a variable that participates in the most conflicts.
  Select a value that minimizes the number of conflicts.
  What data structures are required?
- Any Conflict: Select a variable that appears in any conflict.
  Select a value at random.
  What data structures are required?
- Select a variable at random.
  Select a value that minimizes the number of conflicts.
  What data structures are required?

Greedy Descent Variants

To select a variable to change and a new value for it:

- Most Improving Step: Find a variable-value pair that minimizes the number of conflicts.
  What data structures are required?
- Two Stage Choice: Select a variable that participates in the most conflicts.
  Select a value that minimizes the number of conflicts.
  What data structures are required?
- Any Conflict: Select a variable that appears in any conflict.
  Select a value at random.
  What data structures are required?
- Select a variable at random.
  Select a value that minimizes the number of conflicts.
  What data structures are required?
- Select a variable and value at random; accept this change if it doesn't increase the number of conflicts.

Greedy Descent Variants

To select a variable to change and a new value for it:

- Most Improving Step: Find a variable-value pair that minimizes the number of conflicts.
  What data structures are required?
- Two Stage Choice: Select a variable that participates in the most conflicts.
  Select a value that minimizes the number of conflicts.
  What data structures are required?
- Any Conflict: Select a variable that appears in any conflict.
  Select a value at random.
  What data structures are required?
- Select a variable at random.
  Select a value that minimizes the number of conflicts.
  What data structures are required?
- Select a variable and value at random; accept this change if it doesn't increase the number of conflicts.
  What data structures are required?

Steps

- One measure of an assignment is number of conflicts
- It is possible to weight some conflicts higher than others.
- Why would we?

Steps

- One measure of an assignment is number of conflicts
- It is possible to weight some conflicts higher than others.
- Why would we?
  Because some are easier to solve than other. E.g., in scheduling exams....

Steps

- One measure of an assignment is number of conflicts
- It is possible to weight some conflicts higher than others.
- Why would we?
  Because some are easier to solve than other. E.g., in scheduling exams....
- If A is a total assignment, define $h(A)$ to be a measure of the difficulty of solving problem from A.
- $h(A) = 0$ then A a solution; lower h is better

Variant: Simulated Annealing

- Pick a variable at random and a new value at random.
- If it isn't worse, accept it.
- If it is worse,

Variant: Simulated Annealing

- Pick a variable at random and a new value at random.
- If it isn't worse, accept it.
- If it is worse, accept it probabilistically depending on a temperature parameter, T:
  - With current assignment A and proposed assignment A' accept A' with probability $e^{(h(A)-h(A'))/T}$

Note: $h(A) - h(A')$ is negative if A' is worse.

Variant: Simulated Annealing

- Pick a variable at random and a new value at random.
- If it isn't worse, accept it.
- If it is worse, accept it probabilistically depending on a temperature parameter, T:
  - With current assignment A and proposed assignment A' accept A' with probability $e^{(h(A)-h(A'))/T}$

  Note: $h(A) - h(A')$ is negative if A' is worse.
- Probability of accepting a change:

Temperature	1-worse	2-worse	3-worse
10	0.91	0.81	0.74
1	0.37	0.14	0.05
0.25	0.02	0.0003	0.000006
0.1	0.00005	2×10^{-9}	9×10^{-14}

Variant: Simulated Annealing

- Pick a variable at random and a new value at random.
- If it isn't worse, accept it.
- If it is worse, accept it probabilistically depending on a temperature parameter, T:
  - With current assignment A and proposed assignment A' accept A' with probability $e^{(h(A)-h(A'))/T}$

  Note: $h(A) - h(A')$ is negative if A' is worse.
- Probability of accepting a change:

Temperature	1-worse	2-worse	3-worse
10	0.91	0.81	0.74
1	0.37	0.14	0.05
0.25	0.02	0.0003	0.000006
0.1	0.00005	2×10^{-9}	9×10^{-14}

- Temperature can be reduced.

Random Restart

- A random restart involves reassigning all variables to values at random.
- allows for exploration of a different part of the search space.

Random Restart

- A random restart involves reassigning all variables to values at random.
- allows for exploration of a different part of the search space.
- Each run is independent of the others, so probabilities can be derived analytically.
  Suppose each run has a probability of p of finding a solution.
  We do n runs or until a solution is found.

Random Restart

- A random restart involves reassigning all variables to values at random.
- allows for exploration of a different part of the search space.
- Each run is independent of the others, so probabilities can be derived analytically.
  Suppose each run has a probability of p of finding a solution.
  We do n runs or until a solution is found.
  The probability of n runs failing to find a solution is

Random Restart

- A random restart involves reassigning all variables to values at random.
- allows for exploration of a different part of the search space.
- Each run is independent of the others, so probabilities can be derived analytically.
  Suppose each run has a probability of p of finding a solution.
  We do n runs or until a solution is found.
  The probability of n runs failing to find a solution is $(1-p)^n$
  The probability of finding a solution in n-runs is

Random Restart

- A random restart involves reassigning all variables to values at random.
- allows for exploration of a different part of the search space.
- Each run is independent of the others, so probabilities can be derived analytically.
  Suppose each run has a probability of p of finding a solution.
  We do n runs or until a solution is found.
  The probability of n runs failing to find a solution is $(1-p)^n$
  The probability of finding a solution in n-runs is $1-(1-p)^n$

Random Restart

- A random restart involves reassigning all variables to values at random.
- allows for exploration of a different part of the search space.
- Each run is independent of the others, so probabilities can be derived analytically.
  Suppose each run has a probability of p of finding a solution.
  We do n runs or until a solution is found.
  The probability of n runs failing to find a solution is $(1-p)^n$
  The probability of finding a solution in n-runs is $1-(1-p)^n$

n	$p = 0.1$	$p = 0.3$	$p = 0.5$	$p = 0.8$
5	0.410	0.832	0.969	0.9997
10	0.65	0.971	0.9990	0.9999998
20	0.878	0.9992	0.9999991	0.99999999999
50	0.995	0.99999998	0.9999999999999991	1.0

Tabu lists

- To prevent cycling we can maintain a tabu list of the k last assignments.
- Don't allow an assignment that is already on the tabu list.

Tabu lists

- To prevent cycling we can maintain a tabu list of the k last assignments.
- Don't allow an assignment that is already on the tabu list.
- If $k = 1$, we don't allow an assignment of to the same value to the variable chosen.

Tabu lists

- To prevent cycling we can maintain a tabu list of the k last assignments.
- Don't allow an assignment that is already on the tabu list.
- If $k = 1$, we don't allow an assignment of to the same value to the variable chosen.
- We can implement it more efficiently than as a list of complete assignments.

Tabu lists

- To prevent cycling we can maintain a tabu list of the k last assignments.
- Don't allow an assignment that is already on the tabu list.
- If $k = 1$, we don't allow an assignment of to the same value to the variable chosen.
- We can implement it more efficiently than as a list of complete assignments.
- It can be expensive if k is large.

Complex Domains

- When the domains are small or unordered, the neighbors of an assignment can correspond to choosing another value for one of the variables.

Complex Domains

- When the domains are small or unordered, the neighbors of an assignment can correspond to choosing another value for one of the variables.
- When the domains are large and ordered, the neighbors of an assignment are the adjacent values for one of the variables.

Complex Domains

- When the domains are small or unordered, the neighbors of an assignment can correspond to choosing another value for one of the variables.
- When the domains are large and ordered, the neighbors of an assignment are the adjacent values for one of the variables.
- If the domains are continuous, Gradient descent changes each variable proportionally to the gradient of the heuristic function in that direction.
  The value of variable X_i goes from v_i to

Complex Domains

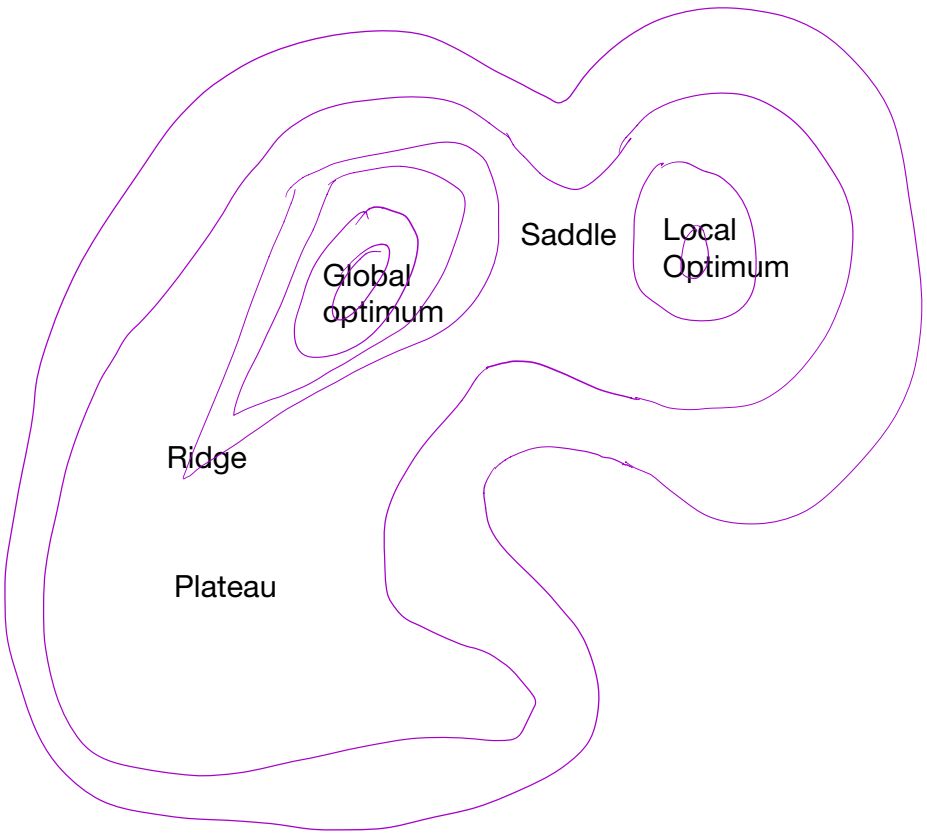
- When the domains are small or unordered, the neighbors of an assignment can correspond to choosing another value for one of the variables.
- When the domains are large and ordered, the neighbors of an assignment are the adjacent values for one of the variables.
- If the domains are continuous, Gradient descent changes each variable proportionally to the gradient of the heuristic function in that direction.
  The value of variable X_i goes from v_i to $v_i - \eta \frac{\partial h}{\partial X_i}$.
  η is the step size.

Complex Domains

- When the domains are small or unordered, the neighbors of an assignment can correspond to choosing another value for one of the variables.
- When the domains are large and ordered, the neighbors of an assignment are the adjacent values for one of the variables.
- If the domains are continuous, Gradient descent changes each variable proportionally to the gradient of the heuristic function in that direction.
  The value of variable X_i goes from v_i to $v_i - \eta \frac{\partial h}{\partial X_i}$.
  η is the step size.
- Neural networds do gradient descent with thousands or millions or billions of dimensions to minimize error on a dataset. (See CPSC 340).

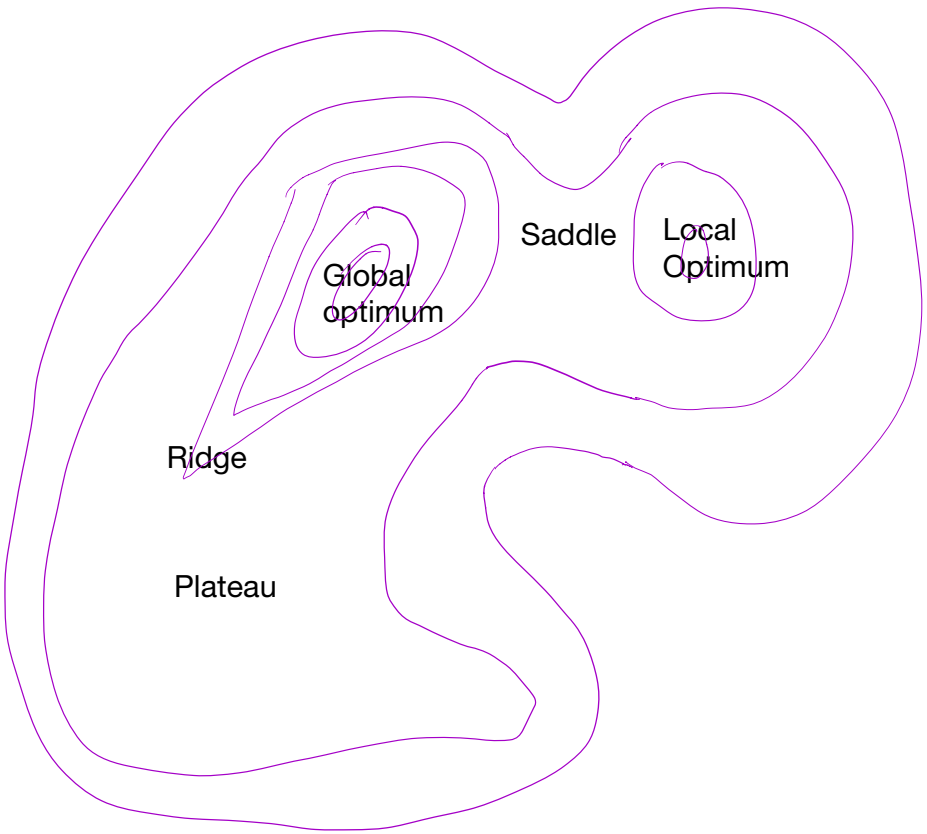
Problems with Greedy Descent

- a local optimum that is not a global optimum



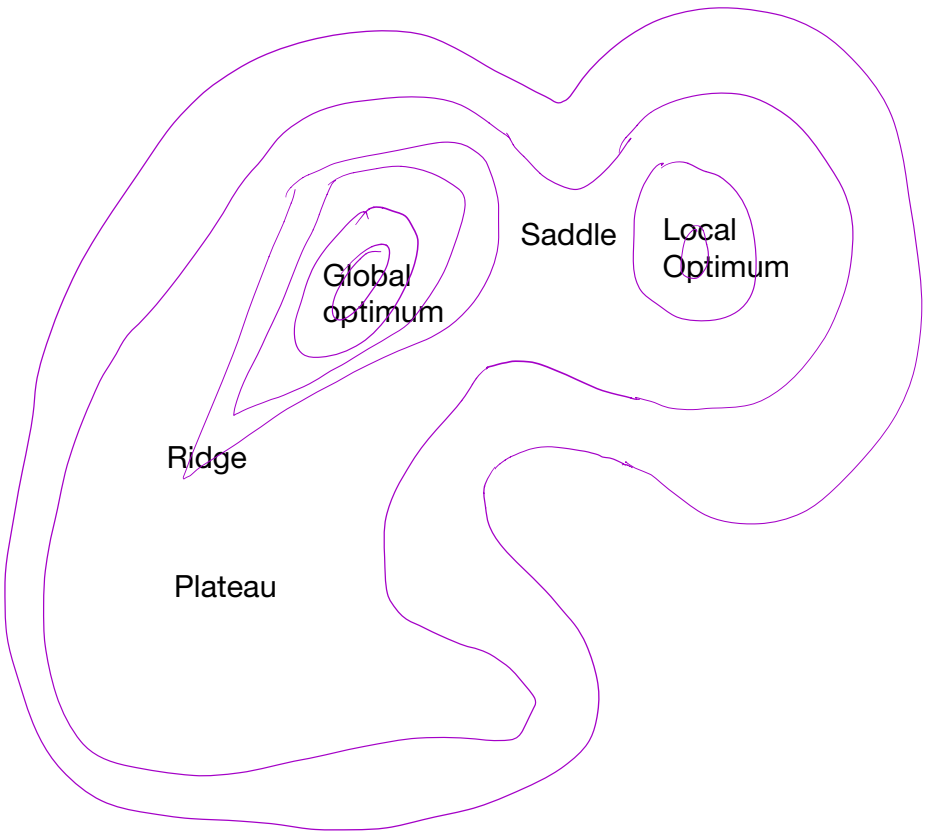
Problems with Greedy Descent

- a local optimum that is not a global optimum
- a plateau where the heuristic values are uninformative



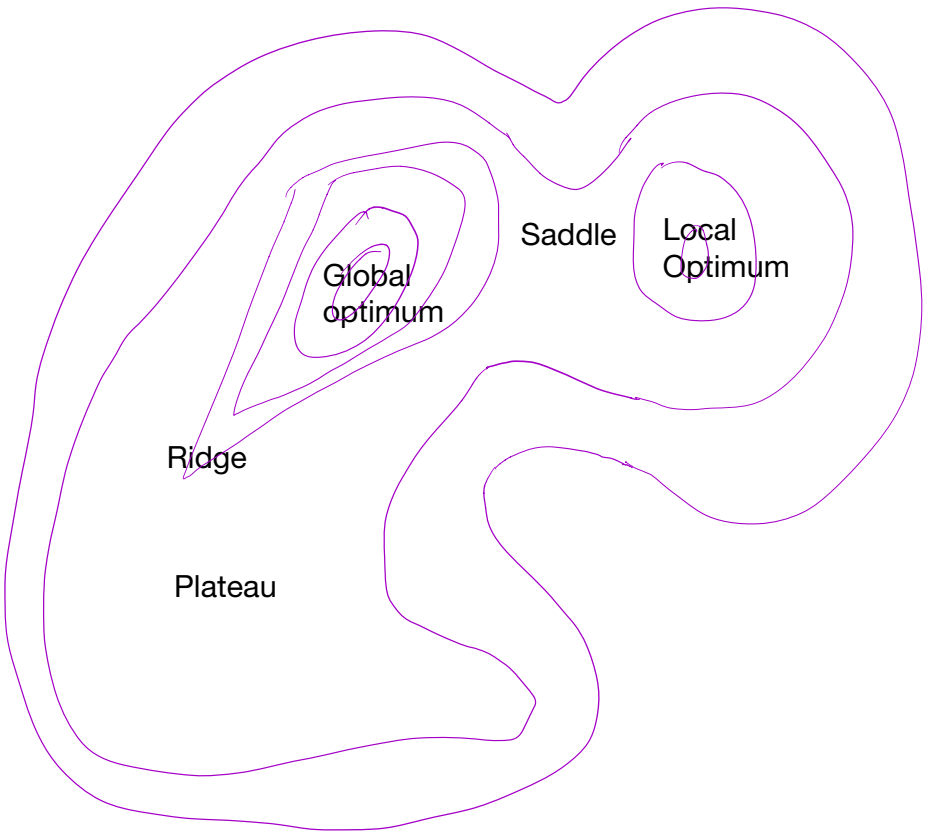
Problems with Greedy Descent

- a local optimum that is not a global optimum
- a plateau where the heuristic values are uninformative
- a ridge is a local minimum where n-step look-ahead might help



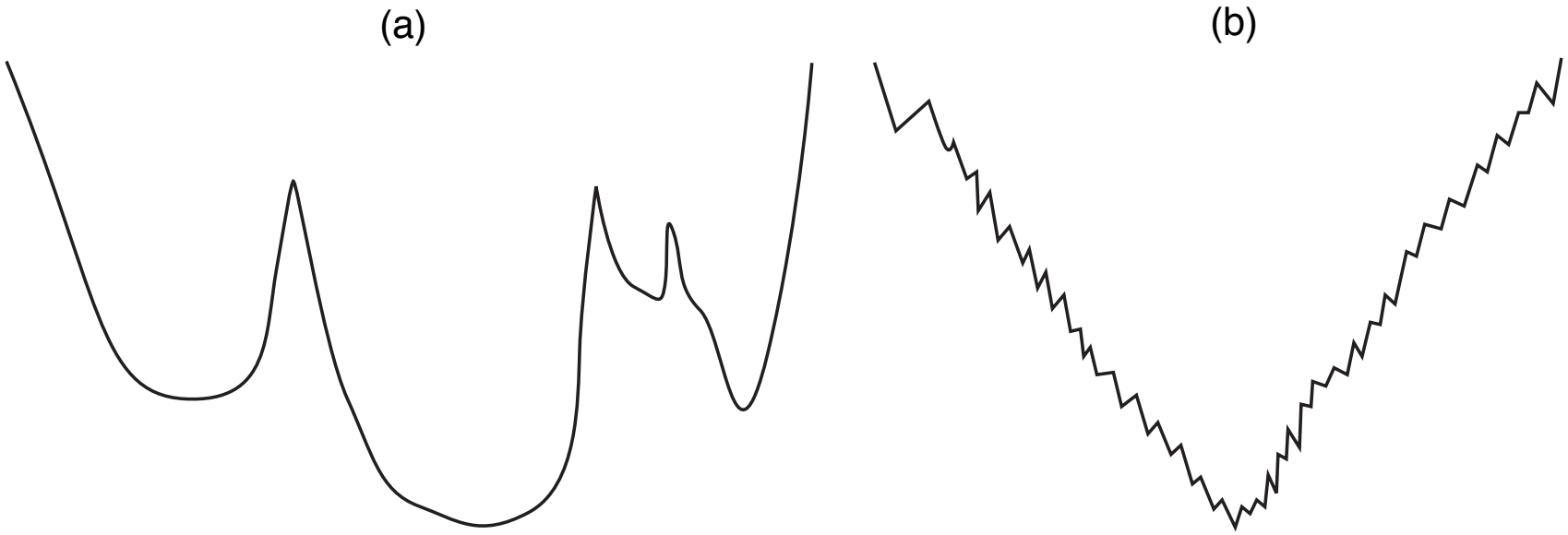
Problems with Greedy Descent

- a local optimum that is not a global optimum
- a plateau where the heuristic values are uninformative
- a ridge is a local minimum where n-step look-ahead might help
- a saddle is a flat area where steps need to change direction



1-Dimensional Ordered Examples

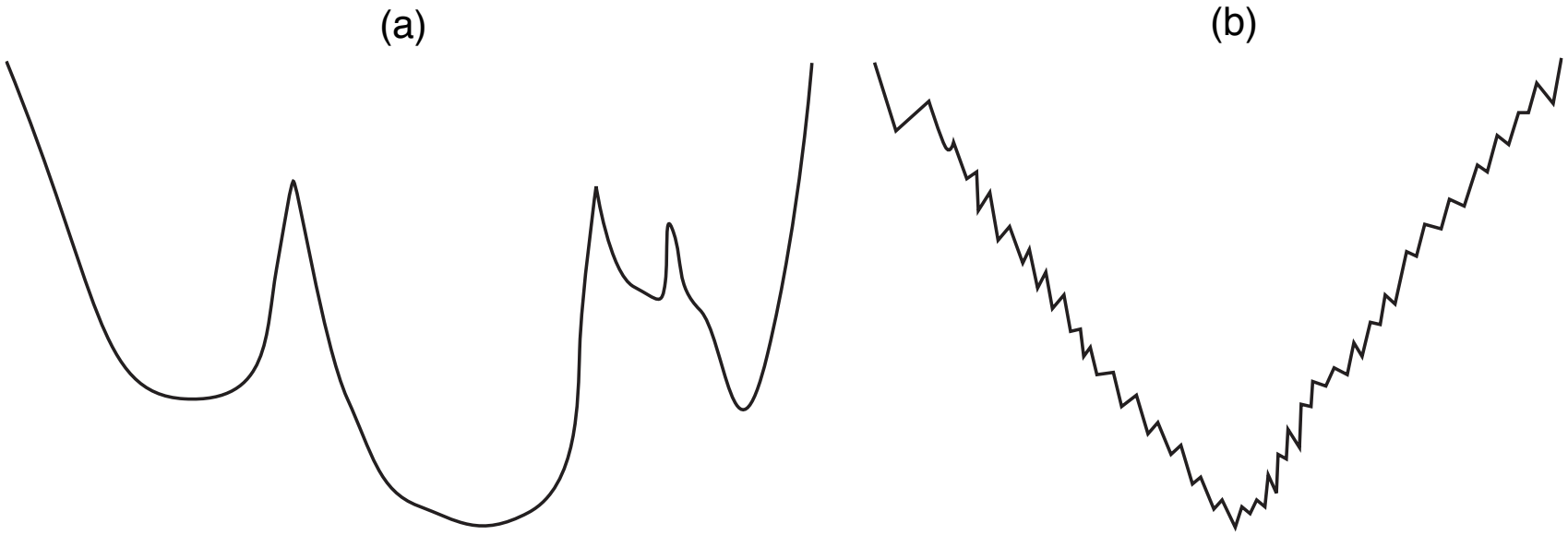
Two 1-dimensional search spaces; small step right or left:



- Which method would most easily find the global minimum?

1-Dimensional Ordered Examples

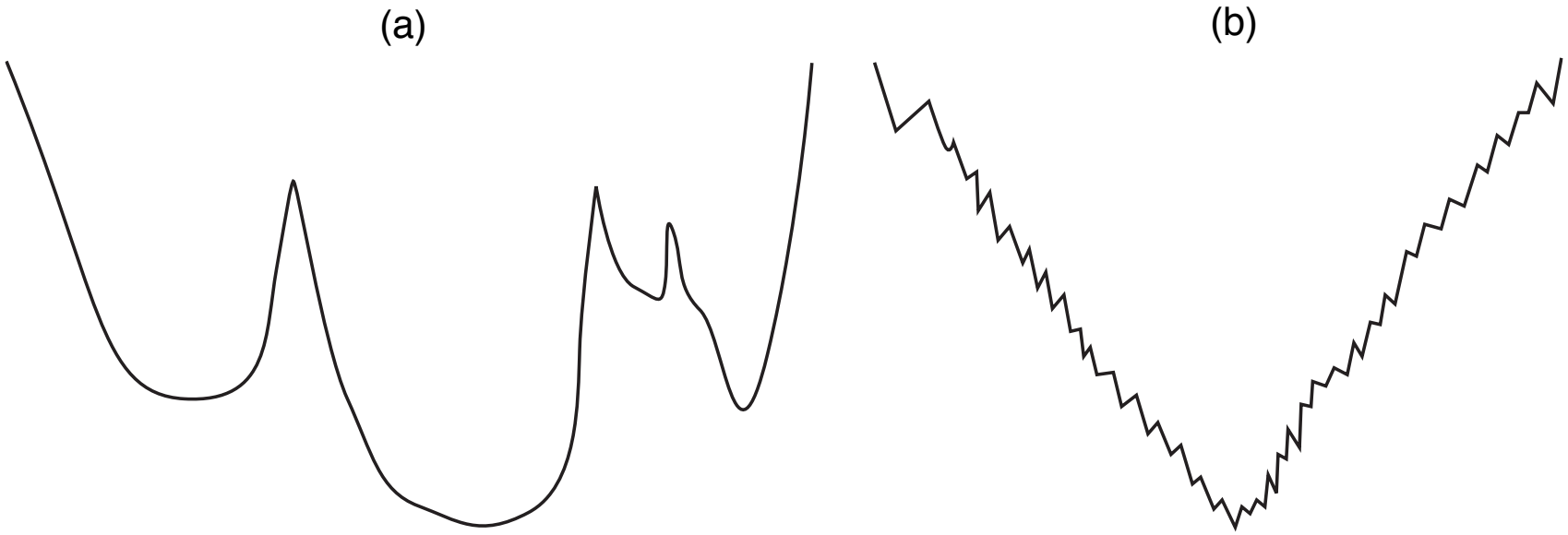
Two 1-dimensional search spaces; small step right or left:



- Which method would most easily find the global minimum?
- What happens in hundreds or thousands of dimensions?

1-Dimensional Ordered Examples

Two 1-dimensional search spaces; small step right or left:



- Which method would most easily find the global minimum?
- What happens in hundreds or thousands of dimensions?
- What if different parts of the search space have different structure?

Parallel Search

A total assignment is called an individual.

- Idea: maintain a population of k individuals instead of one.
- At every stage, update each individual in the population.

Parallel Search

A total assignment is called an individual.

- Idea: maintain a population of k individuals instead of one.
- At every stage, update each individual in the population.
- Whenever an individual is a solution, it can be reported.

Parallel Search

A total assignment is called an individual.

- Idea: maintain a population of k individuals instead of one.
- At every stage, update each individual in the population.
- Whenever an individual is a solution, it can be reported.
- Like k restarts, but uses k times the *minimum* number of steps.

Beam Search

- Like parallel search, with k individuals, but choose the k best out of all of the neighbors.

Beam Search

- Like parallel search, with k individuals, but choose the k best out of all of the neighbors.
- When $k = 1$, it is greedy descent.

Beam Search

- Like parallel search, with k individuals, but choose the k best out of all of the neighbors.
- When $k = 1$, it is greedy descent.
- The value of k lets us limit space and parallelism.

Stochastic Beam Search

- Like beam search, but it probabilistically chooses the k individuals at the next generation.

Stochastic Beam Search

- Like beam search, but it probabilistically chooses the k individuals at the next generation.
- The probability that a neighbor is chosen is proportional to its heuristic value.

Stochastic Beam Search

- Like beam search, but it probabilistically chooses the k individuals at the next generation.
- The probability that a neighbor is chosen is proportional to its heuristic value.
- This maintains diversity amongst the individuals.
- The heuristic value reflects the fitness of the individual.

Stochastic Beam Search

- Like beam search, but it probabilistically chooses the k individuals at the next generation.
- The probability that a neighbor is chosen is proportional to its heuristic value.
- This maintains diversity amongst the individuals.
- The heuristic value reflects the fitness of the individual.
- Like asexual reproduction: each individual mutates and the fittest ones survive.

Genetic Algorithms

- Like stochastic beam search, but pairs of individuals are combined to create the offspring.
- For each generation:
  - Randomly choose pairs of individuals where the fittest individuals are more likely to be chosen.
  - For each pair, perform a crossover: form two offspring each taking different parts of their parents.

Genetic Algorithms

- Like stochastic beam search, but pairs of individuals are combined to create the offspring.
- For each generation:
  - Randomly choose pairs of individuals where the fittest individuals are more likely to be chosen.
  - For each pair, perform a crossover: form two offspring each taking different parts of their parents.
  - Mutate some values.
- Stop when a solution is found.

Crossover

- Given two individuals:

$$X_1 = a_1, X_2 = a_2, \ldots, X_m = a_m$$

$$X_1 = b_1, X_2 = b_2, \ldots, X_m = b_m$$

- Select i at random.
- Form two offspring:

$$X_1 = a_1, \ldots, X_i = a_i, X_{i+1} = b_{i+1}, \ldots, X_m = b_m$$

$$X_1 = b_1, \ldots, X_i = b_i, X_{i+1} = a_{i+1}, \ldots, X_m = a_m$$

- The effectiveness depends on the ordering of the variables.

Crossover

- Given two individuals:

$$X_1 = a_1, X_2 = a_2, \ldots, X_m = a_m$$

$$X_1 = b_1, X_2 = b_2, \ldots, X_m = b_m$$

- Select i at random.
- Form two offspring:

$$X_1 = a_1, \ldots, X_i = a_i, X_{i+1} = b_{i+1}, \ldots, X_m = b_m$$

$$X_1 = b_1, \ldots, X_i = b_i, X_{i+1} = a_{i+1}, \ldots, X_m = a_m$$

- The effectiveness depends on the ordering of the variables.
- Many variations are possible.

Optimization

An optimization problem is given

- a set of variables, each with an associated domain
- an objective function that maps total assignments to real numbers, and
- an optimality criterion, which is typically to find a total assignment that minimizes (or maximizes) the objective function.

Constraint optimization problem

- In a constraint optimization problem the objective function is factored into a sum of soft constraints
- A soft constraint is a function from scope of constraint into non-negative reals (the cost)

Constraint optimization problem

- In a constraint optimization problem the objective function is factored into a sum of soft constraints
- A soft constraint is a function from scope of constraint into non-negative reals (the cost)
- The aim is to find a total assignment that minimizes the sum of the values of the soft constraints.

Constraint optimization problem

- In a constraint optimization problem the objective function is factored into a sum of soft constraints
- A soft constraint is a function from scope of constraint into non-negative reals (the cost)
- The aim is to find a total assignment that minimizes the sum of the values of the soft constraints.
- Can use systematic search (e.g., A^* or branch-and-bound search)

Constraint optimization problem

- In a constraint optimization problem the objective function is factored into a sum of soft constraints
- A soft constraint is a function from scope of constraint into non-negative reals (the cost)
- The aim is to find a total assignment that minimizes the sum of the values of the soft constraints.
- Can use systematic search (e.g., A^* or branch-and-bound search)
- Arc consistency can be used to prune dominated values

Constraint optimization problem

- In a constraint optimization problem the objective function is factored into a sum of soft constraints
- A soft constraint is a function from scope of constraint into non-negative reals (the cost)
- The aim is to find a total assignment that minimizes the sum of the values of the soft constraints.
- Can use systematic search (e.g., A^* or branch-and-bound search)
- Arc consistency can be used to prune dominated values
- Can use local search

Constraint optimization problem

- In a constraint optimization problem the objective function is factored into a sum of soft constraints
- A soft constraint is a function from scope of constraint into non-negative reals (the cost)
- The aim is to find a total assignment that minimizes the sum of the values of the soft constraints.
- Can use systematic search (e.g., A^* or branch-and-bound search)
- Arc consistency can be used to prune dominated values
- Can use local search
- Problem: we can't tell if a value is a global minimum unless we do systematic search

CS 188 Introduction to Artificial Intelligence
Fall 2018 Note 1

These lecture notes are heavily based on notes originally written by Nikhil Sharma.

Agents

In artificial intelligence, the central problem at hand is that of the creation of a rational **agent**, an entity that has goals or preferences and tries to perform a series of **actions** that yield the best/optimal expected outcome given these goals. Rational agents exist in an **environment**, which is specific to the given instantiation of the agent. As a very simple example, the environment for a checkers agent is the virtual checkers board on which it plays against opponents, where piece moves are actions. Together, an environment and the agents that reside within it create a **world**.

A **reflex agent** is one that doesn't think about the consequences of its actions, but rather selects an action based solely on the current state of the world. These agents are typically outperformed by **planning agents**, which maintain a model of the world and use this model to simulate performing various actions. Then, the agent can determine hypothesized consequences of the actions and can select the best one. This is simulated "intelligence" in the sense that it's exactly what humans do when trying to determine the best possible move in any situation - thinking ahead.

State Spaces and Search Problems

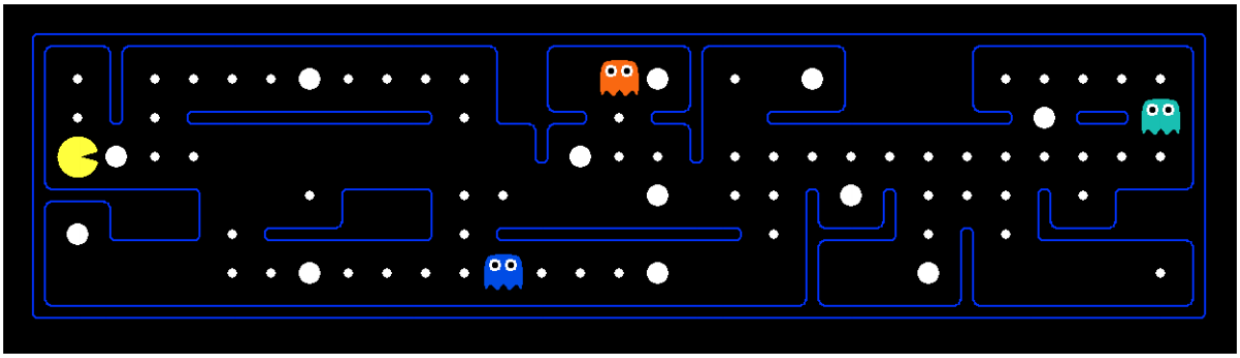
In order to create a rational planning agent, we need a way to mathematically express the given environment in which the agent will exist. To do this, we must formally express a **search problem** - given our agent's current **state** (its configuration within its environment), how can we arrive at a new state that satisfies its goals in the best possible way? Formulating such a problem requires four things:

- A **state space** - The set of all possible states that are possible in your given world
- A **successor function** - A function that takes in a state and an action and computes the cost of performing that action as well as the **successor state**, the state the world would be in if the given agent performed that action
- A **start state** - The state in which an agent exists initially
- A **goal test** - A function that takes a state as input, and determines whether it is a goal state

Fundamentally, a search problem is solved by first considering the start state, then exploring the state space using the successor function, iteratively computing successors of various states until we arrive at a goal state, at which point we will have determined a path from the start state to the goal state (typically called a **plan**). The order in which states are considered is determined using a predetermined **strategy**. We'll cover types of strategies and their usefulness shortly.

Before we continue with how to solve search problems, it's important to note the difference between a **world state**, and a **search state**. A world state contains all information about a given state, whereas a search state

contains only the information about the world that's necessary for planning (primarily for space effiency reasons). To illustrate these concepts, we'll introduce the hallmark motivating example of this course - Pacman. The game of Pacman is simple: Pacman must navigate a maze and eat all the (small) food pellets in the maze without being eaten by the malicious patrolling ghosts. If Pacman eats one of the (large) power pellets, he becomes ghost-immune for a set period of time and gains the ability to eat ghosts for points.



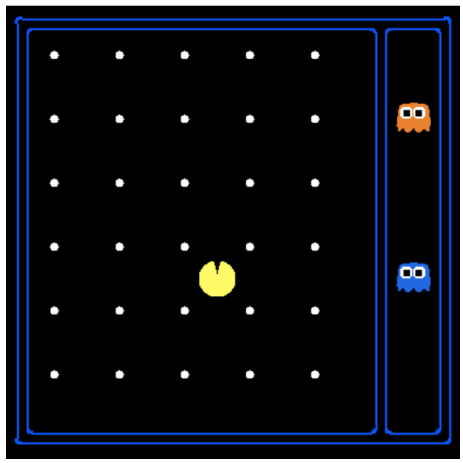
Let's consider a variation of the game in which the maze contains only Pacman and food pellets. We can pose two distinct search problems in this scenario: pathing and eat-all-dots. Pathing attempts to solve the problem of getting from position (x_1, y_1) to position (x_2, y_2) in the maze optimally, while eat all dots attempts to solve the problem of consuming all food pellets in the maze in the shortest time possible. Below, the states, actions, successor function, and goal test for both problems are listed:

- **Pathing**
  - States: (x,y) locations
  - Actions: North, South, East, West
  - Successor: Update location only
  - Goal test: Is (x,y)=END?
- **Eat-all-dots**
  - States: (x,y) location, dot booleans
  - Actions: North, South, East, West
  - Successor: Update location and booleans
  - Goal test: Are all dot booleans false?

Note that for pathing, states contain less information than states for eat-all-dots, because for eat-all-dots we must maintain an array of booleans corresponding to each food pellet and whether or not it's been eaten in the given state. A world state may contain more information still, potentially encoding information about things like total distance traveled by Pacman or all positions visited by Pacman on top of its current (x,y) location and dot booleans.

State Space Size

An important question that often comes up while estimating the computational runtime of solving a search problem is the size of the state space. This is done almost exclusively with the **fundamental counting principle**, which states that if there are n variable objects in a given world which can take on x_1, x_2, ..., x_n different values respectively, then the total number of states is $x_1 \cdot x_2 \cdot \ldots \cdot x_n$. Let's use Pacman to show this concept by example:



Let's say that the variable objects and their corresponding number of possiblilites are as follows:

- *Pacman positions* - Pacman can be in 120 distinct (x, y) positions, and there is only one Pacman
- *Pacman Direction* - this can be North, South, East, or West, for a total of 4 possibilities
- *Ghost positions* - There are two ghosts, each of which can be in 12 distinct (x, y) positions
- *Food pellet configurations* - There are 30 food pellets, each of which can be eaten or not eaten

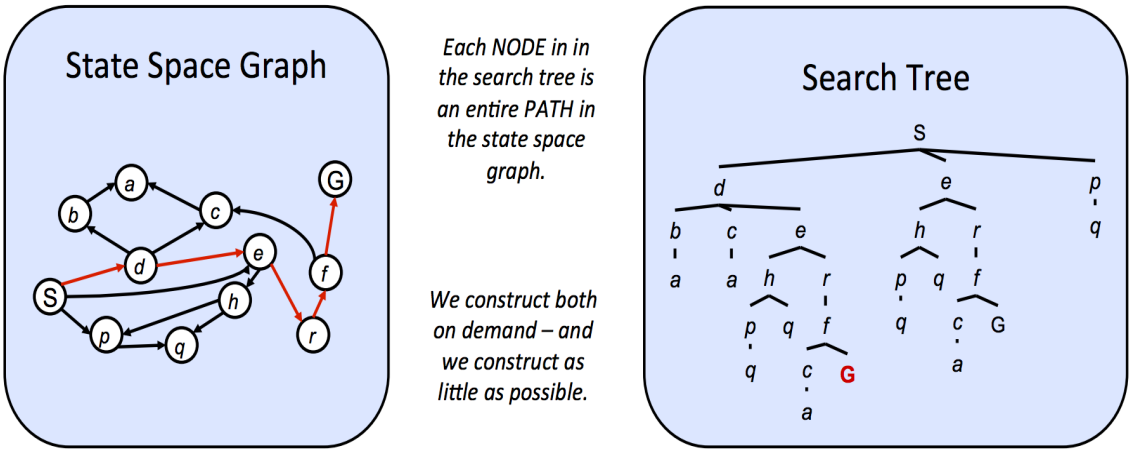
Using the fundamental counting principle, we have 120 positions for Pacman, 4 directions Pacman can be facing, $12 \cdot 12$ ghost configurations (12 for each ghost), and $2 \cdot 2 \cdot \ldots \cdot 2 = 2^{30}$ food pellet configurations (each of 30 food pellets has two possible values - eaten or not eaten). This gives us a total state space size of $\boxed{120 \cdot 4 \cdot 12^2 \cdot 2^{30}}$.

State Space Graphs and Search Trees

Now that we've established the idea of a state space and the four components necessary to completely define one, we're almost ready to begin solving search problems. The final piece of the puzzle is that of state space graphs and search trees.

Recall that a graph is defined by a set of nodes and a set of edges connecting various pairs of nodes. These edges may also have weights associated with them. A **state space graph** is constructed with states representing nodes, with directed edges existing from a state to its successors. These edges represent actions, and any associated weights represent the cost of performing the corresponding action. Typically, state space graphs are much too large to store in memory (even our simple Pacman example from above has $\approx 10^{13}$ possible states, yikes!), but they're good to keep in mind conceptually while solving problems. It's also important to note that in a state space graph, each state is represented exactly once - there's simply no need to represent a state multiple times, and knowing this helps quite a bit when trying to reason about search problems.

Unlike state space graphs, our next structure of interest, **search trees**, have no such restriction on the number of times a state can appear. This is because though search trees are also a class of graph with states as nodes and actions as edges between states, each state/node encodes not just the state itself, but the entire path (or **plan**) from the start state to the given state in the state space graph. Observe the state space graph and corresponding search tree below:



The highlighted path (S $\rightarrow$ d $\rightarrow$ e $\rightarrow$ r $\rightarrow$ f $\rightarrow$ G) in the given state space graph is represented in the corresponding search tree by following the path in the tree from the start state S to the highlighted goal state G. Similarly, each and every path from the start node to any other node is represented in the search tree by a path from the root S to some descendant of the root corresponding to the other node. Since there often exist multiple ways to get from one state to another, states tend to show up multiple times in search trees. As a result, search trees are greater than or equal to their corresponding state space graph in size.

We've already determined that state space graphs themselves can be enormous in size even for simple problems, and so the question arises - how can we perform useful computation on these structures if they're too big to represent in memory? The answer lies in successor functions - we only store states we're immediately working with, and compute new ones on-demand using the corresponding successor function. Typically, search problems are solved using search trees, where we very carefully store a select few nodes to observe at a time, iteratively replacing nodes with their successors until we arrive at a goal state. There exist various methods by which to decide the order in which to conduct this iterative replacement of search tree nodes, and we'll present these methods now.

Uninformed Search

The standard protocol for finding a plan to get from the start state to a goal state is to maintain an outer **fringe** of partial plans derived from the search tree. We continually **expand** our fringe by removing a node (which is selected using our given **strategy**) corresponding to a partial plan from the fringe, and replacing it on the fringe with all its children. Removing and replacing an element on the fringe with its children corresponds to discarding a single length n plan and bringing all length $(n+1)$ plans that stem from it into consideration. We continue this until eventually removing a goal state off the fringe, at which point we conclude the partial plan corresponding to the removed goal state is in fact a path to get from the start state to the goal state. Practically, most implementations of such algorithms will encode information about the parent node, distance to node, and the state inside the node object. This procedure we have just outlined is known as **tree search**, and the pseudocode for it is presented below:

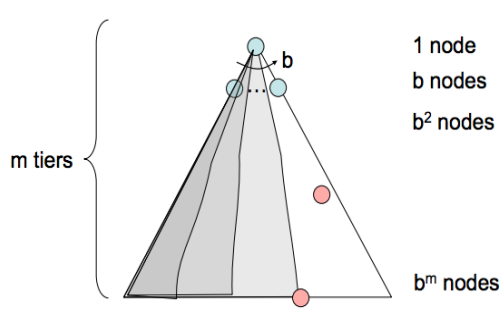
```
function TREE-SEARCH(problem, fringe) return a solution, or failure
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST(problem, STATE[node]) then return node
        for child-node in EXPAND(STATE[node], problem) do
            fringe ← INSERT(child-node, fringe)
        end
    end
```

When we have no knowledge of the location of goal states in our search tree, we are forced to select our strategy for tree search from one of the techniques that falls under the umbrella of **uninformed search**. We'll now cover three such strategies in succession: **depth-first search**, **breadth-first search**, and **uniform cost search**. Along with each strategy, some rudimentary properties of the strategy are presented as well, in terms of the following:

- The **completeness** of each search strategy - if there exists a solution to the search problem, is the strategy guaranteed to find it given infinite computational resources?
- The **optimality** of each search strategy - is the strategy guaranteed to find the lowest cost path to a goal state?
- The **branching factor** b - The increase in the number of nodes on the fringe each time a fringe node is dequeued and replaced with its children is $O(b)$. At depth k in the search tree, there exists $O(b^k)$ nodes.
- The maximum depth m.
- The depth of the shallowest solution s.

Depth-First Search

- *Description* - Depth-first search (DFS) is a strategy for exploration that always selects the *deepest* fringe node from the start node for expansion.
- *Fringe representation* - Removing the deepest node and replacing it on the fringe with its children necessarily means the children are now the new deepest nodes - their depth is one greater than the depth of the previous deepest node. This implies that to implement DFS, we require a structure that always gives the most recently added objects highest priority. A last-in, first-out (LIFO) stack does exactly this, and is what is traditionally used to represent the fringe when implementing DFS.



- *Completeness* - Depth-first search is not complete. If there exist cycles in the state space graph, this inevitably means that the corresponding search tree will be infinite in depth. Hence, there exists the possibility that DFS will faithfully yet tragically get "stuck" searching for the deepest node in an infinite-sized search tree, doomed to never find a solution.

- *Optimality* - Depth-first search simply finds the "leftmost" solution in the search tree without regard for path costs, and so is not optimal.

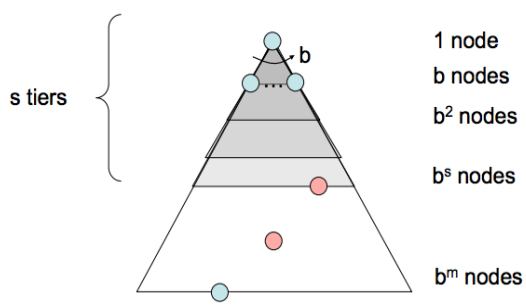
- *Time Complexity* - In the worst case, depth first search may end up exploring the entire search tree. Hence, given a tree with maximum depth m, the runtime of DFS is $O(b^m)$.

- *Space Complexity* - In the worst case, DFS maintains b nodes at each of m depth levels on the fringe. This is a simple consequence of the fact that once b children of some parent are enqueued, the nature of DFS allows only one of the subtrees of any of these children to be explored at any given point in time. Hence, the space complexity of BFS is $O(bm)$.

Breadth-First Search

- *Description* - Breadth-first search is a strategy for exploration that always selects the *shallowest* fringe node from the start node for expansion.

- *Fringe representation* - If we want to visit shallower nodes before deeper nodes, we must visit nodes in their order of insertion. Hence, we desire a structure that outputs the oldest enqueued object to represent our fringe. For this, BFS uses a first-in, first-out (FIFO) queue, which does exactly this.



- *Completeness* - If a solution exists, then the depth of the shallowest node s must be finite, so BFS must eventually search this depth. Hence, it's complete.

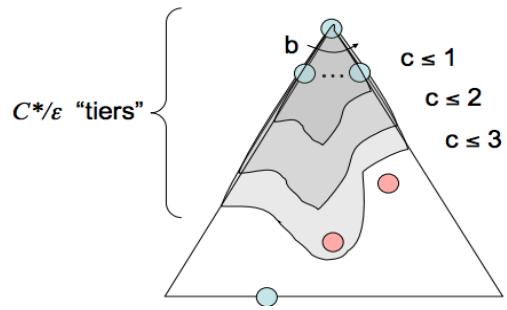
- *Optimality* - BFS is generally not optimal because it simply does not take costs into consideration when determining which node to replace on the fringe. The special case where BFS is guaranteed to be optimal is if all edge costs are equivalent, because this reduces BFS to a special case of uniform cost search, which is discussed below.

- *Time Complexity* - We must search $1+b+b^2+...+b^s$ nodes in the worst case, since we go through all nodes at every depth from 1 to s. Hence, the time complexity is $O(b^s)$.

- *Space Complexity* - The fringe, in the worst case, contains all the nodes in the level corresponding to the shallowest solution. Since the shallowest solution is located at depth s, there are $O(b^s)$ nodes at this depth.

Uniform Cost Search

- *Description* - Uniform cost search (UCS), our last strategy, is a strategy for exploration that always selects the *lowest cost* fringe node from the start node for expansion.
- *Fringe representation* - To represent the fringe for UCS, the choice is usually a heap-based priority queue, where the weight for a given enqueued node v is the path cost from the start node to v, or the *backward cost* of v. Intuitively, a priority queue constructed in this manner simply reshuffles itself to maintain the desired ordering by path cost as we remove the current minimum cost path and replace it with its children.



- *Completeness* - Uniform cost search is complete. If a goal state exists, it must have some finite length shortest path; hence, UCS must eventually find this shortest length path.
- *Optimality* - UCS is also optimal if we assume all edge costs are nonnegative. By construction, since we explore nodes in order of increasing path cost, we're guaranteed to find the lowest-cost path to a goal state. The strategy employed in Uniform Cost Search is identical to that of Dijkstra's algorithm, and the chief difference is that UCS terminates upon finding a solution state instead of finding the shortest path to all states. Note that having negative edge costs in our graph can make nodes on a path have decreasing length, ruining our guarantee of optimality. (See Bellman-Ford algorithm for a slower algorithm that handles this possibility)
- *Time Complexity* - Let us define the optimal path cost as C^* and the minimal cost between two nodes in the state space graph as ε. Then, we must roughly explore all nodes at depths ranging from 1 to C^*/ε, leading to an runtime of $O(b^{C^*/\varepsilon})$.
- *Space Complexity* - Roughly, the fringe will contain all nodes at the level of the cheapest solution, so the space complexity of UCS is estimated as $O(b^{C^*/\varepsilon})$.

As a parting note about uninformed search, it's critical to note that the three strategies outlined above are fundamentally the same - differing only in expansion strategy, with their similarities being captured by the tree search pseudocode presented above.

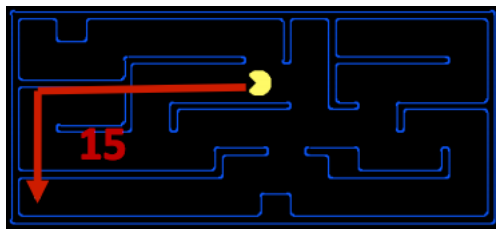
Informed Search

Uniform cost search is good because it's both complete and optimal, but it can be fairly slow because it expands in every direction from the start state while searching for a goal. If we have some notion of the direction in which we should focus our search, we can significantly improve performance and "hone in" on a goal much more quickly. This is exactly the focus of **informed search**.

Heuristics

Heuristics are the driving force that allow estimation of distance to goal states - they're functions that take in a state as input and output a corresponding estimate. The computation performed by such a function is specific to the search problem being solved. For reasons that we'll see in A* search, below, we usually want heuristic functions to be a lower bound on this remaining distance to the goal, and so heuristics are typically solutions to **relaxed problems** (where some of the constraints of the original problem have been removed). Turning to our Pacman example, let's consider the pathing problem described earlier. A common heuristic that's used to solve this problem is the **Manhattan distance**, which for two points (x_1, y_1) and (x_2, y_2) is defined as follows:

$$Manhattan(x_1, y_1, x_2, y_2) = |x_1 - x_2| + |y_1 - y_2|$$



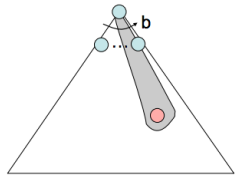
The above visualization shows the relaxed problem that the Manhattan distance helps solve - assuming Pacman desires to get to the bottom left corner of the maze, it computes the distance from Pacman's current location to Pacman's desired location *assuming a lack of walls in the maze*. This distance is the *exact* goal distance in the relaxed search problem, and correspondingly is the *estimated* goal distance in the actual search problem. With heuristics, it becomes very easy to implement logic in our agent that enables them to "prefer" expanding states that are estimated to be closer to goal states when deciding which action to perform. This concept of preference is very powerful, and is utilized by the following two search algorithms that implement heuristic functions: greedy search and A*.

Greedy Search

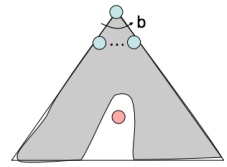
- *Description* - Greedy search is a strategy for exploration that always selects the fringe node with the *lowest heuristic value* for expansion, which corresponds to the state it believes is nearest to a goal.
- *Fringe representation* - Greedy search operates identically to UCS, with a priority queue fringe representation. The difference is that instead of using *computed backward cost* (the sum of edge weights in the path to the state) to assign priority, greedy search uses *estimated forward cost* in the form of heuristic values.
- *Completeness and Optimality* - Greedy search is not guaranteed to find a goal state if one exists, nor is it optimal, particularly in cases where a very bad heuristic function is selected. It generally acts fairly unpredictably from scenario to scenario, and can range from going straight to a goal state to acting like a badly-guided DFS and exploring all the wrong areas.

A* Search

- *Description* - A* search is a strategy for exploration that always selects the fringe node with the *lowest estimated total cost* for expansion, where total cost is the entire cost from the start node to the goal node.



(a) Greedy search on a good day :)



(b) Greedy search on a bad day :(

- *Fringe representation* - Just like greedy search and UCS, A* search also uses a priority queue to represent its fringe. Again, the only difference is the method of priority selection. A* combines the total backward cost (sum of edge weights in the path to the state) used by UCS with the estimated forward cost (heuristic value) used by greedy search by adding these two values, effectively yielding an *estimated total cost* from start to goal. Given that we want to minimize the total cost from start to goal, this is an excellent choice.

- *Completeness and Optimality* - A* search is both complete and optimal, given an appropriate heuristic (which we'll cover in a minute). It's a combination of the good from all the other search strategies we've covered so far, incorporating the generally high speed of greedy search with the optimality and completeness of UCS!

Admissibility and Consistency

Now that we've discussed heuristics and how they are applied in both greedy and A* search, let's spend some time discussing what constitutes a good heuristic. To do so, let's first reformulate the methods used for determining priority queue ordering in UCS, greedy search, and A* with the following definitions:

- $g(n)$ - The function representing total backwards cost computed by UCS.
- $h(n)$ - The *heuristic value* function, or estimated forward cost, used by greedy search.
- $f(n)$ - The function representing estimated total cost, used by A* search. $f(n) = g(n) + h(n)$.

Before attacking the question of what constitutes a "good" heuristic, we must first answer the question of whether A* maintains its properties of completeness and optimality regardless of the heuristic function we use. Indeed, it's very easy to find heuristics that break these two coveted properties. As an example, consider the heuristic function $h(n) = 1 - g(n)$. Regardless of the search problem, using this heuristic yields

$$\begin{aligned} f(n) &= g(n) + h(n) \\ &= g(n) + (1 - g(n)) \\ &= 1 \end{aligned}$$

Hence, such a heuristic reduces A* search to BFS, where all edge costs are equivalent. As we've already shown, BFS is not guaranteed to be optimal in the general case where edge weights are not constant.

The condition required for optimality when using A* tree search is known as **admissibility**. The admissibility constraint states that the value estimated by an admissible heuristic is neither negative nor an overestimate. Defining $h^*(n)$ as the true optimal forward cost to reach a goal state from a given node n, we can formulate the admissibility constraint mathematically as follows:

$$\forall n,\ 0 \leq h(n) \leq h^*(n)$$

Theorem. For a given search problem, if the admissibility constraint is satisfied by a heuristic function h, using A* tree search with h on that search problem will yield an optimal solution.

Proof. Assume two reachable goal states are located in the search tree for a given search problem, an optimal goal A and a suboptimal goal B. Some ancestor n of A (including perhaps A itself) must currently be on the fringe, since A is reachable from the start state. We claim n will be selected for expansion before B, using the following three statements:

1. $g(A) < g(B)$. Because A is given to be optimal and B is given to be suboptimal, we can conclude that A has a lower backwards cost to the start state than B.

2. $h(A) = h(B) = 0$, because we are given that our heuristic satisfies the admissibility constraint. Since both A and B are both goal states, the true optimal cost to a goal state from A or B is simply $h^*(n) = 0$; hence $0 \leq h(n) \leq 0$.

3. $f(n) \leq f(A)$, because, through admissibility of h, $f(n) = g(n) + h(n) \leq g(n) + h^*(n) = g(A) = f(A)$. The total cost through node n is at most the true backward cost of A, which is also the total cost of A.

We can combine statements 1. and 2. to conclude that $f(A) < f(B)$ as follows:

$$f(A) = g(A) + h(A) = g(A) < g(B) = g(B) + h(B) = f(B)$$

A simple consequence of combining the above derived inequality with statement 3. is the following:

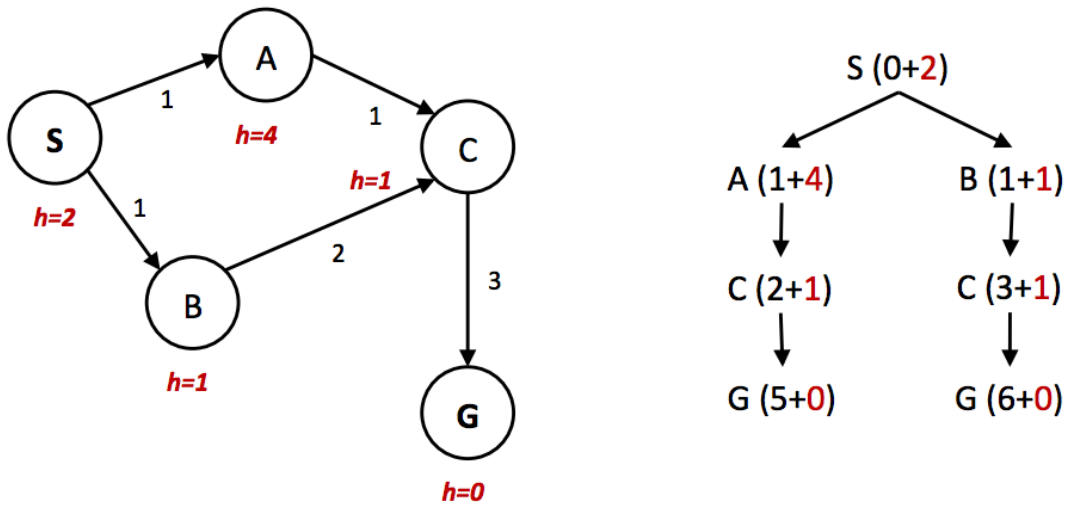
$$f(n) \leq f(A) \wedge f(A) < f(B) \Longrightarrow f(n) < f(B)$$

Hence, we can conclude that n is expanded before B. Because we have proven this for arbitrary n, we can conclude that *all* ancestors of A (including A itself) expand before B. □

One problem we found above with tree search was that in some cases it could fail to ever find a solution, getting stuck searching the same cycle in the state space graph infinitely. Even in situations where our search technique doesn't involve such an infinite loop, it's often the case that we revisit the same node multiple times because there's multiple ways to get to that same node. This leads to exponentially more work, and the natural solution is to simply keep track of which states you've already expanded, and never expand them again. More explicitly, maintain a "closed" set of expanded nodes while utilizing your search method of choice. Then, ensure that each node isn't already in the set before expansion and add it to the set after expansion if it's not. Tree search with this added optimization is known as **graph search**, and the pseudocode for it is presented below:

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

Note that in implementation, it's critically important to store the closed set as a disjoint set and not a list. Storing it as a list requires costs $O(n)$ operations to check for membership, which eliminates the performance improvement graph search is intended to provide. An additional caveat of graph search is that it tends to ruin the optimality of A*, even under admissible heuristics. Consider the following simple state space graph and corresponding search tree, annotated with weights and heuristic values:



In the above example, it's clear that the optimal route is to follow $S \to A \to C \to G$, yielding a total path cost of $1+1+3=5$. The only other path to the goal, $S \to B \to C \to G$ has a path cost of $1+2+3=6$. However, because the heuristic value of node A is so much larger than the heuristic value of node B, node C is first expanded along the second, suboptimal path as a child of node B. It's then placed into the "closed" set, and so A* graph search fails to reexpand it when it visits it as a child of A, so it never finds the optimal solution. Hence, to maintain completeness and optimality under A* graph search, we need an even stronger property than admissibility, **consistency**. The central idea of consistency is that we enforce not only that a heuristic underestimates the *total* distance to a goal from any given node, but also the cost/weight of each edge in the graph. The cost of an edge as measured by the heuristic function is simply the difference in heuristic values for two connected nodes. Mathematically, the consistency constraint can be expressed as follows:

$$\forall A, C \quad h(A) - h(C) \leq cost(A, C)$$

Theorem. For a given search problem, if the consistency constraint is satisfied by a heuristic function h, using A* graph search with h on that search problem will yield an optimal solution.

Proof. In order to prove the above theorem, we first prove that when running A* graph search with a consistent heuristic, whenever we remove a node for expansion, we've found the optimal path to that node.

Using the consistency constraint, we can show that the values of $f(n)$ for nodes along any plan are nondecreasing. Define two nodes, n and n', where n' is a successor of n. Then:

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

If for every parent-child pair (n, n') along a path, $f(n') \geq f(n)$, then it must be the case that the values of $f(n)$ are nondecreasing along that path. We can check that the above graph violates this rule between $f(A)$

and $f(C)$. With this information, we can now show that whenever a node n is removed for expansion, its optimal path has been found. Assume towards a contradiction that this is false - that when n is removed from the fringe, the path found to n is suboptimal. This means that there must be some ancestor of n, n'', on the fringe that was never expanded but is on the optimal path to n. Contradiction! We've already shown that values of f along a path are nondecreasing, and so n'' would have been removed for expansion before n.

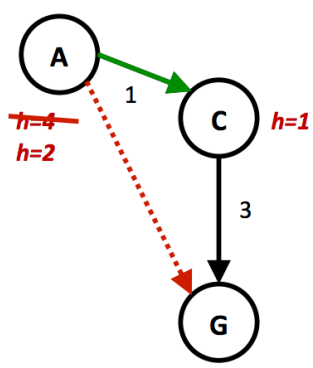
All we have left to show to complete our proof is that an optimal goal A will always be removed for expansion and returned before any suboptimal goal B. This is trivial, since $h(A) = h(B) = 0$, so

$$f(A) = g(A) < g(B) = f(B)$$

just as in our proof of optimality of A* tree search under the admissibility constraint. Hence, we can conclude that A* graph search is optimal under a consistent heuristic. □

A couple of important highlights from the discussion above before we proceed: for heuristics that are either admissible/consistent to be valid, it must by definition be the case that $h(G) = 0$ for any goal state G. Additionally, consistency is not just a stronger constraint than admissibility, consistency *implies* admissibility. This stems simply from the fact that if no edge costs are overestimates (as guaranteed by consistency), the total estimated cost from any node to a goal will also fail to be an overestimate.

Consider the following three-node network for an example of an admissible but inconsistent heuristic:



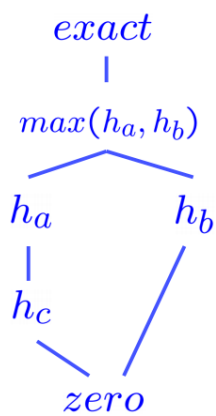
The red dotted line corresponds to the total estimated goal distance. If $h(A) = 4$, then the heuristic is admissible, as the distance from A to the goal is $4 \geq h(A)$, and same for $h(C) = 1 \leq 3$. However, the heuristic cost from A to C is $h(A) - h(C) = 4 - 1 = 3$. Our heuristic estimates the cost of the edge between A and C to be 3 while the true value is $cost(A,C) = 1$, a smaller value. Since $h(A) - h(C) \nleq cost(A,C)$, this heuristic is not consistent. Running the same computation for $h(A) = 2$, however, yields $h(A) - h(C) = 2 - 1 = 1 \leq cost(A,C)$. Thus, using $h(A) = 2$ makes our heuristic consistent.

Dominance

Now that we've established the properties of admissibility and consistency and their roles in maintaining the optimality of A* search, we can return to our original problem of creating "good" heuristics, and how to tell if one heuristic is better than another. The standard metric for this is that of **dominance**. If heuristic a is dominant over heuristic b, then the estimated goal distance for a is greater than the estimated goal distance for b for every node in the state space graph. Mathematically,

$$\forall n : h_a(n) \geq h_b(n)$$

Dominance very intuitively captures the idea of one heuristic being better than another - if one admissible/consistent heuristic is dominant over another, it must be better because it will always more closely estimate the distance to a goal from any given state. Additionally, the **trivial heuristic** is defined as $h(n) = 0$, and using it reduces A* search to UCS. All admissible heuristics dominate the trivial heuristic. The trivial heuristic is often incorporated at the base of a **semi-lattice** for a search problem, a dominance hierarchy of which it is located at the bottom. Below is an example of a semi-lattice that incorporates various heuristics h_a, h_b, and h_c ranging from the trivial heuristic at the bottom to the exact goal distance at the top:



As a general rule, the max function applied to multiple admissible heuristics will also always be admissible. This is simply a consequence of all values output by the heuristics for any given state being constrained by the admissibility condition, $0 \leq h(n) \leq h^*(n)$. The maximum of numbers in this range must also fall in the same range. The same can be shown easily for multiple consistent heuristics as well. It's common practice to generate multiple admissible/consistent heuristics for any given search problem and compute the max over the values output by them to generate a heuristic that dominates (and hence is better than) all of them individually.

Summary

In this note, we discussed *search problems*, which we characterize formally with four components: a *state space*, a *successor function*, a *start state*, and a *goal state*. Search problems can be solved using a variety of search techniques, including but not limited to the five we study in CS 188:

- *Breadth-first Search*
- *Depth-first Search*
- *Uniform Cost Search*
- *Greedy Search*
- *A* Search*

The first three search techniques listed above are examples of *uninformed search*, while the latter two are examples of *informed search* which use *heuristics* to estimate goal distance and optimize performance.

CS 188 Introduction to Artificial Intelligence
Fall 2018 Note 2

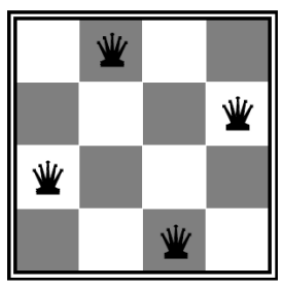
These lecture notes are heavily based on notes originally written by Nikhil Sharma.

Constraint Satisfaction Problems

In the previous note, we learned how to find optimal solutions to search problems, a type of **planning problem**. Now, we'll learn about solving a related class of problems, **constraint satisfaction problems** (CSPs). Unlike search problems, CSPs are a type of **identification problem**, problems in which we must simply identify whether a state is a goal state or not, with no regard to how we arrive at that goal. CSPs are defined by three factors:

1. *Variables* - CSPs possess a set of N variables $X_1, ..., X_N$ that can each take on a single value from some defined set of values.
2. *Domain* - A set $\{x_1, ..., x_d\}$ representing all possible values that a CSP variable can take on.
3. *Constraints* - Constraints define restrictions on the values of variables, potentially with regard to other variables.

Consider the N-queens identification problem: given an $N \times N$ chessboard, can we find a configuration in which to place N queens on the board such that no two queens attack each another?



We can formulate this problem as a CSP as follows:

1. *Variables* - X_{ij}, with $0 \leq i, j < N$. Each X_{ij} represents a grid position on our $N \times N$ chessboard, with i and j specifying the row and column number respectively.
2. *Domain* - $\{0, 1\}$. Each X_{ij} can take on either the value 0 or 1, a boolean value representing the existence of a queen at position (i, j) on the board.
3. *Constraints* -
   - $\forall i, j, k \; (X_{ij}, X_{ik}) \in \{(0,0), (0,1), (1,0)\}$. This constraint states that if two variables have the same value for i, only one of them can take on a value of 1. This effectively encapsulates the condition that no two queens can be in the same row.

- $\forall i, j, k\ (X_{ij}, X_{kj}) \in \{(0,0),(0,1),(1,0)\}$. Almost identically to the previous constraint, this constraint states that if two variables have the same value for j, only one of them can take on a value of 1, encapsulating the condition that no two queens can be in the same column.
- $\forall i, j, k\ (X_{ij}, X_{i+k,j+k}) \in \{(0,0),(0,1),(1,0)\}$. With similar reasoning as above, we can see that this constraint and the next represent the conditions that no two queens can be in the same major or minor diagonals, respectively.
- $\forall i, j, k\ (X_{ij}, X_{i+k,j-k}) \in \{(0,0),(0,1),(1,0)\}$.
- $\sum_{i,j} X_{ij} = N$. This constraint states that we must have exactly N grid positions marked with a 1, and all others marked with a 0, capturing the requirement that there are exactly N queens on the board.

Constraint satisfaction problems are **NP-hard**, which loosely means that there exists no known algorithm for finding solutions to them in polynomial time. Given a problem with N variables with domain of size $O(d)$ for each variable, there are $O(d^N)$ possible assignments, exponential in the number of variables. We can often get around this caveat by formulating CSPs as search problems, defining states as **partial assignments** (variable assignments to CSPs where some variables have been assigned values while others have not). Correspondingly, the successor function for a CSP state outputs all states with one new variable assigned, and the goal test verifies all variables are assigned and all constraints are satisfied in the state it's testing. Constraint satisfaction problems tend to have significantly more structure than traditional search problems, and we can exploit this structure by combining the above formulation with appropriate heuristics to hone in on solutions in a feasible amount of time.

Constraint Graphs

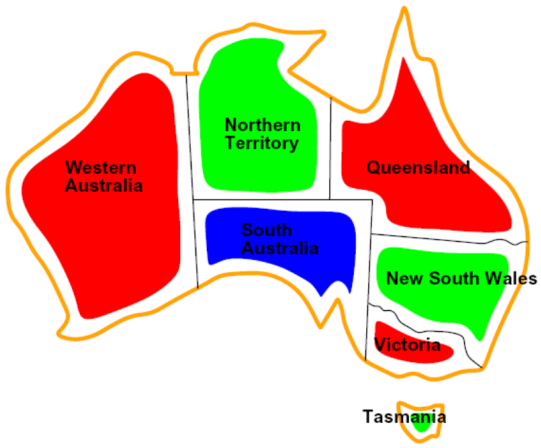
Let's introduce a second CSP example: map coloring. Map coloring solves the problem where we're given a set of colors and must color a map such that no two adjacent states or regions have the same color.

Constraint satisfaction problems are often represented as constraint graphs, where nodes represent variables and edges represent constraints between them. There are many different types of constraints, and each is handled slightly differently:

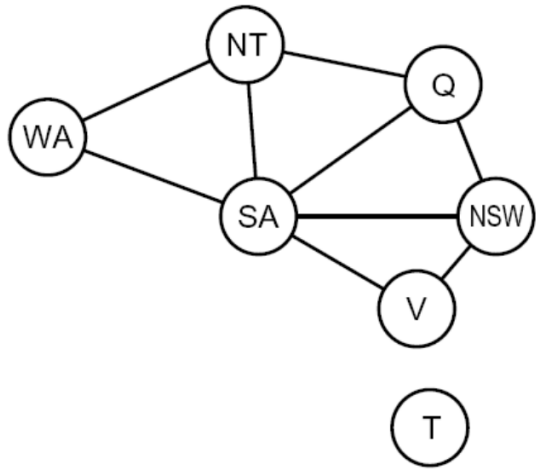
- *Unary Constraints* - Unary constraints involve a single variable in the CSP. They are not represented in constraint graphs, instead simply being used to prune the domain of the variable they constrain when necessary.

- *Binary Constraints* - Binary constraints involve two variables. They're represented in constraint graphs as traditional graph edges.
- *Higher-order Constraints* - Constraints involving three or more variables can also be represented with edges in a CSP graph, they just look slightly unconventional.

Consider map coloring the map of Australia:



The constraints in this problem are simply that no two adjacent states can be the same color. As a result, by drawing an edge between every pair of states that are adjacent to one another, we can generate the constraint graph for the map coloring of Australia as follows:



The value of constraint graphs is that we can use them to extract valuable information about the structure of the CSPs we are solving. By analyzing the graph of a CSP, we can determine things about it like whether it's sparsely or densely connected/constrained and whether or not it's tree-structured. We'll cover this more in depth as we discuss solving constraint satisfaction problems in more detail.

Solving Constraint Satisfaction Problems

Constraint satisfaction problems are traditionally solved using a search algorithm known as **backtracking search**. Backtracking search is an optimization on depth first search used specifically for the problem of constraint satisfaction, with improvements coming from two main principles:

1. Fix an ordering for variables, and select values for variables in this order. Because assignments are commutative (e.g. assigning $WA = Red$, $NT = Green$ is identical to $NT = Green$, $WA = Red$), this is valid.

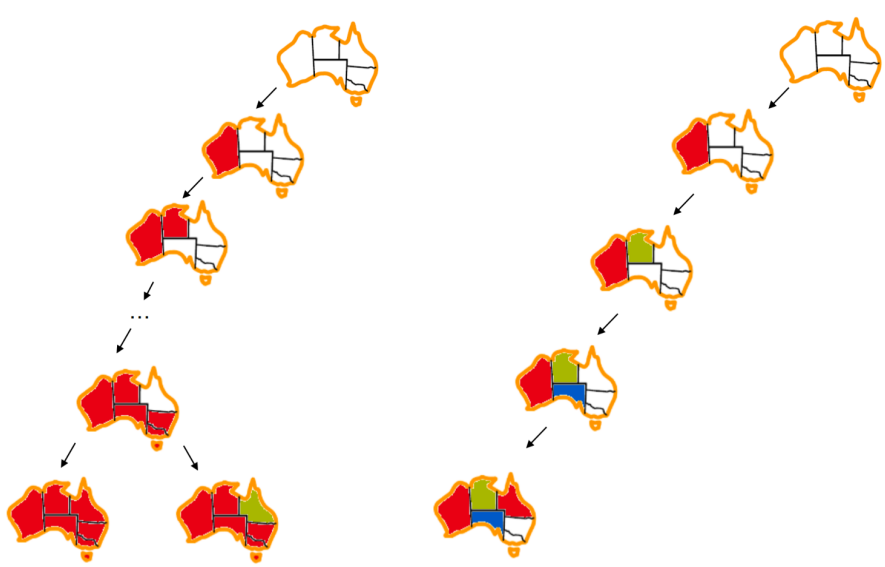
2. When selecting values for a variable, only select values that don't conflict with any previously assigned values. If no such values exist, backtrack and return to the previous variable, changing its value.

The pseudocode for how recursive backtracking works is presented below:

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

function RECURSIVE-BACKTRACKING(assignment, csp) returns soln/failure
   if assignment is complete then return assignment
   var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
   for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
      if value is consistent with assignment given CONSTRAINTS[csp] then
         add {var = value} to assignment
         result ← RECURSIVE-BACKTRACKING(assignment, csp)
         if result ≠ failure then return result
         remove {var = value} from assignment
   return failure
```

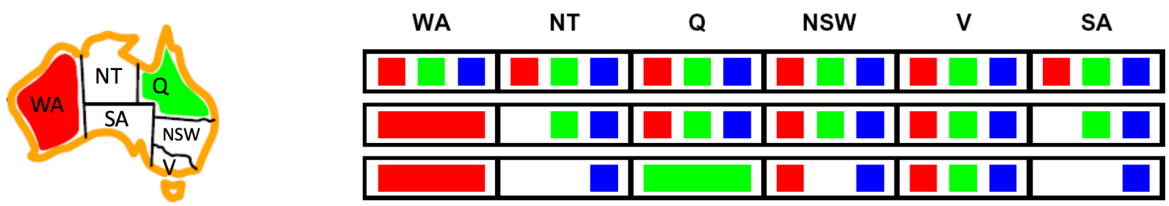
For a visualization of how this process works, consider the partial search trees for both depth first search and backtracking search in map coloring:



Note how DFS regretfully colors everything red before ever realizing the need for change, and even then doesn't move too far in the right direction towards a solution. On the other hand, backtracking search only assigns a value to a variable if that value violates no constraints, leading to a significantly less backtracking. Though backtracking search is a vast improvement over the brute-forcing of depth first search, we can get more gains in speed still with further improvements through filtering, variable/value ordering, and structural explotation.

Filtering

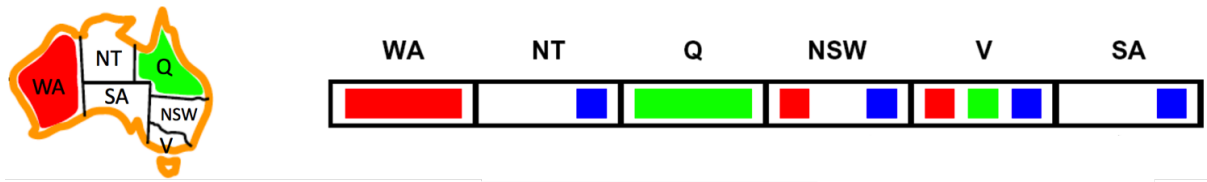
The first improvement to CSP performance we'll consider is **filtering**, which checks if we can prune the domains of unassigned variables ahead of time by removing values we know will result in backtracking. A naïve method for filtering is **forward checking**, which whenever a value is assigned to a variable X_i, prunes the domains of unassigned variables that share a constraint with X_i that would violate the constraint if assigned. Whenever a new variable is assigned, we can run forward checking and prune the domains of unassigned variables adjacent to the newly assigned variable in the constraint graph. Consider our map coloring example, with unassigned variables and their potential values:



Note how as we assign $WA = red$ and then $Q = green$, the size of the domains for NT, NSW, and SA (states adjacent to WA, Q, or both) decrease in size as values are eliminated. The idea of forward checking can be generalized into the principle of **arc consistency**. For arc consistency, we interpret each undirected edge of the constraint graph for a CSP as two directed edges pointing in opposite directions. Each of these directed edges is called an **arc**. The arc consistency algorithm works as follows:

- Begin by storing all arcs in the constraint graph for the CSP in a queue Q.
- Iteratively remove arcs from Q and enforce the condition that in each removed arc $X_i \longrightarrow X_j$, for every remaining value v for the tail variable X_i, there is at least one remaining value w for the head variable X_j such that $X_i = v, X_j = w$ does not violate any constraints. If some value v for X_i would not work with any of the remaining values for X_j, we remove v from the set of possible values for X_i.
- If at least one value is removed for X_i when enforcing arc consistency for an arc $X_i \longrightarrow X_j$, add arcs of the form $X_k \longrightarrow X_i$ to Q, for all unassigned variables X_k. If an arc $X_k \longrightarrow X_i$ is already in Q during this step, it doesn't need to be added again.
- Continue until Q is empty, or the domain of some variable is empty and triggers a backtrack.

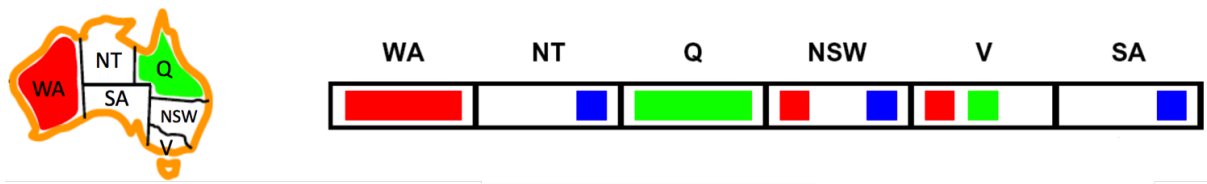
The arc consistency algorithm is typically not the most intuitive, so let's walk through a quick example with map coloring:



We begin by adding all arcs between unassigned variables sharing a constraint to a queue Q, which gives us

$$Q = [SA \to V, V \to SA, SA \to NSW, NSW \to SA, SA \to NT, NT \to SA, V \to NSW, NSW \to V]$$

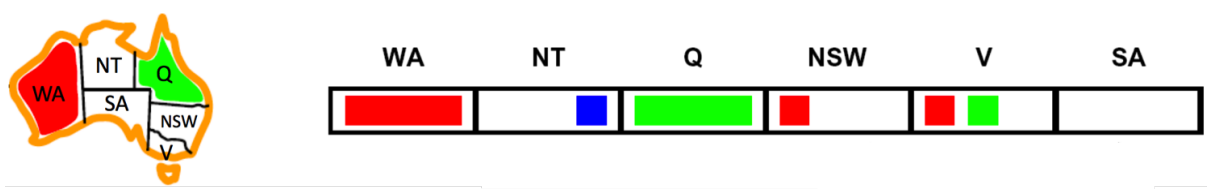
For our first arc, $SA \to V$, we see that for every value in the domain of SA, $\{blue\}$, there is *at least* one value in the domain of V, $\{red, green, blue\}$, that violates no constraints, and so no values need to be pruned from SA's domain. However, for our next arc $V \to SA$, if we set $V = blue$ we see that SA will have no remaining values that violate no constraints, and so we prune $blue$ from V's domain.



Because we pruned a value from the domain of V, we need to enqueue all arcs with V at the head - $SA \to V$, $NSW \to V$. Since $NSW \to V$ is already in Q, we only need to add $SA \to V$, leaving us with our updated queue

$$Q = [SA \to NSW, NSW \to SA, SA \to NT, NT \to SA, V \to NSW, NSW \to V, SA \to V]$$

We can continue this process until we eventually remove the arc $SA \to NT$ from Q. Enforcing arc consistency on this arc removes $blue$ from SA's domain, leaving it empty and triggering a backtrack. Note that the arc $NSW \to SA$ appears before $SA \to NT$ in Q and that enforcing consistency on this arc removes $blue$ from the domain of NSW.



Arc consistency is typically implemented with the AC-3 algorithm (Arc Consistency Algorithm #3), for which the pseudocode is as follows:

```
function AC-3( csp) returns the CSP, possibly with reduced domains
   inputs: csp, a binary CSP with variables {X1, X2, ..., Xn}
   local variables: queue, a queue of arcs, initially all the arcs in csp

   while queue is not empty do                    tail      head
      (Xi, Xj) ← REMOVE-FIRST(queue)
      if REMOVE-INCONSISTENT-VALUES(Xi, Xj) then
         for each Xk in NEIGHBORS[Xi] do
            add (Xk, Xi) to queue
```

```
function REMOVE-INCONSISTENT-VALUES( Xi, Xj) returns true iff succeeds
   removed ← false
   for each x in DOMAIN[Xi] do
      if no value y in DOMAIN[Xj] allows (x,y) to satisfy the constraint Xi ↔ Xj
         then delete x from DOMAIN[Xi]; removed ← true
   return removed
```

The AC-3 algorithm has a worst case time complexity of $O(ed^3)$, where e is the number of arcs (directed edges) and d is the size of the largest domain. Overall, arc consistency is more holistic of a domain pruning technique than forward checking and leads to fewer backtracks, but requires running significantly more computation in order to enforce. Accordingly, it's important to take into account this tradeoff when deciding which filtering technique to implement for the CSP you're attempting to solve.

As an interesting parting note about consistency, arc consistency is a subset of a more generalized notion of consistency known as **k-consistency**, which when enforced guarantees that for any set of k nodes in the CSP, a consistent assignment to any subset of $k-1$ nodes guarantees that the kth node will have at least one consistent value. This idea can be further extended through the idea of **strong k-consistency**. A graph that is strong k-consistent possesses the property that any subset of k nodes is not only k-consistent but also $k-1, k-2, \ldots, 1$ consistent as well. Not surprisingly, imposing a higher degree of consistency on a CSP is more expensive to compute. Under this generalized definition for consistency, we can see that arc consistency is equivalent to 2-consistency.

Ordering

We've delineated that when solving a CSP, we fix some ordering for both the variables and values involved. In practice, it's often much more effective to compute the next variable and corresponding value "on the fly" with two broad principles, **minimum remaining values** and **least constraining value**:

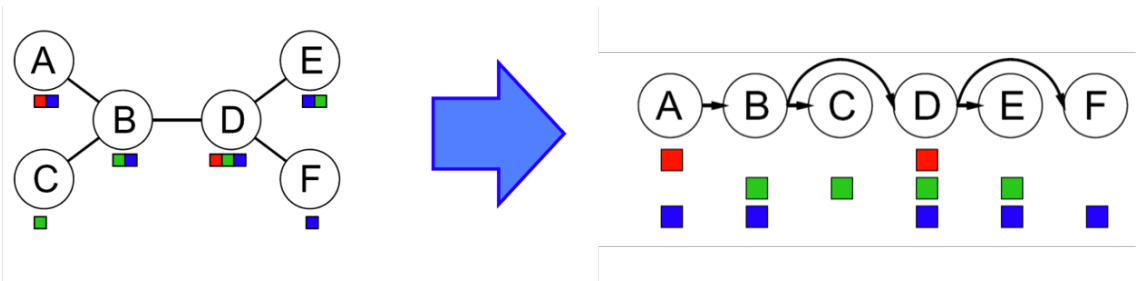
- *Minimum Remaining Values (MRV)* - When selecting which variable to assign next, using an MRV policy chooses whichever unassigned variable has the fewest valid remaining values (the *most constrained variable*). This is intuitive in the sense that the most constrained variable is most likely to run out of possible values and result in backtracking if left unassigned, and so it's best to assign a value to it sooner than later.

- *Least Constraining Value (LCV)* - Similarly, when selecting which value to assign next, a good policy to implement is to select the value that prunes the fewest values from the domains of the remaining unassigned values. Notably, this requires additional computation (e.g. rerunning arc consistency/forward checking or other filtering methods for each value to find the LCV), but can still yield speed gains depending on usage.

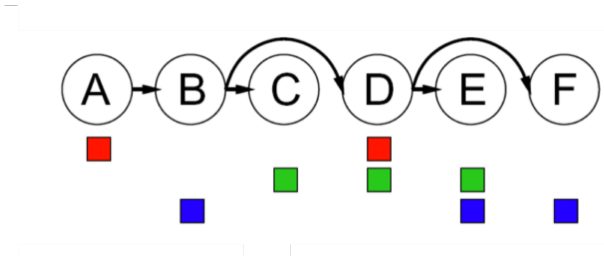
Structure

A final class of improvements to solving constraint satisfaction problems are those that exploit their structure. In particular, if we're trying to solve a **tree-structured CSP** (one that has no loops in its constraint graph), we can reduce the runtime for finding a solution from $O(d^N)$ all the way to $O(nd^2)$, linear in the number of variables. This can be done with the tree-structured CSP algorithm, outlined below:

- First, pick an arbitrary node in the constraint graph for the CSP to serve as the root of the tree (it doesn't matter which one because basic graph theory tells us any node of a tree can serve as a root).
- Convert all undirected edges in the tree to directed edges that point *away* from the root. Then **linearize** (or **topologically sort**) the resulting directed acyclic graph. In simple terms, this just means order the nodes of the graph such that all edges point rightwards. Noting that we select node A to be our root and direct all edges to point away from A, this process results in the following conversion for the CSP presented below:

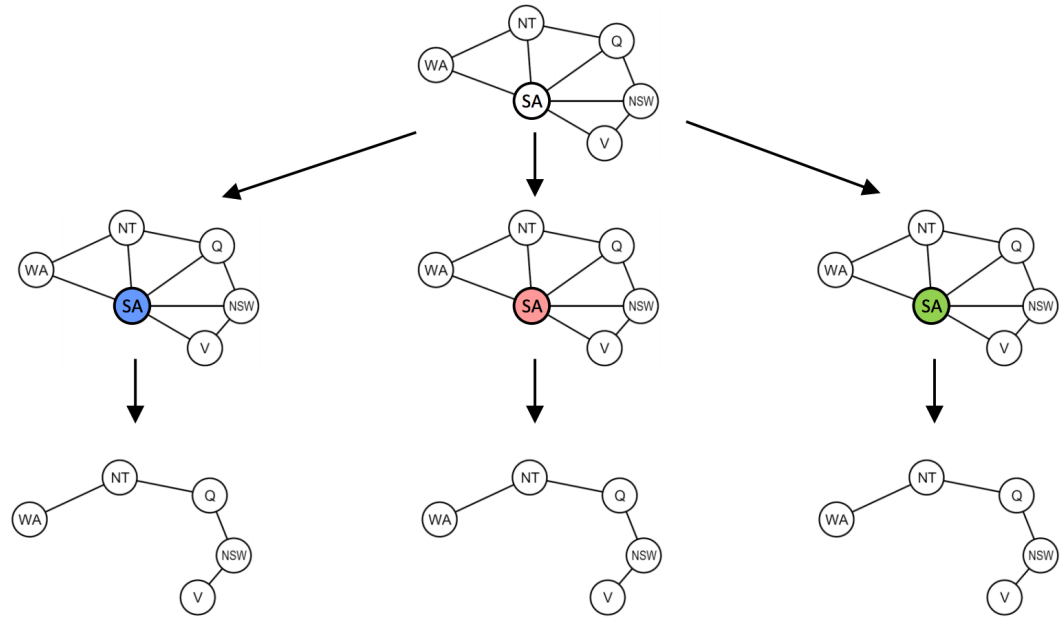


- Perform a **backwards pass** of arc consistency. Iterating from $i = n$ down to $i = 2$, enforce arc consistency for all arcs $Parent(X_i) \longrightarrow X_i$. For the linearized CSP from above, this domain pruning will eliminate a few values, leaving us with the following:



- Finally, perform a **forward assignment**. Starting from X_1 and going to X_n, assign each X_i a value consistent with that of its parent. Because we've enforced arc consistency on all of these arcs, no matter what value we select for any node, we know that its children will each all have at least one consistent value. Hence, this iterative assignment guarantees a correct solution, a fact which can be proven inductively without difficulty.

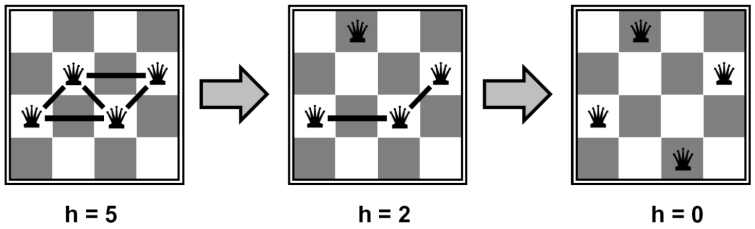
The tree structured algorithm can be extended to CSPs that are reasonably close to being tree-structured with **cutset conditioning**. Cutset conditioning involves first finding the smallest subset of variables in a constraint graph such that their removal results in a tree (such a subset is known as a **cutset** for the graph). For example, in our map coloring example, South Australia (SA) is the smallest possible cutset:



Once the smallest cutset is found, we assign all variables in it and prune the domains of all neighboring nodes. What's left is a tree-structured CSP, upon which we can solve with the tree-structured CSP algorithm from above! The initial assignment to a cutset of size c may leave the resulting tree-structured CSP(s) with no valid solution after pruning, so we may still need to backrack up to d^c times. Since removal of the cutset leaves us with a tree-structured CSP with $(n-c)$ variables, we know this can be solved (or determined that no solution exists) in $O((n-c)d^2)$. Hence, the runtime of cutset conditioning on a general CSP is $O(d^c(n-c)d^2)$, very good for small c.

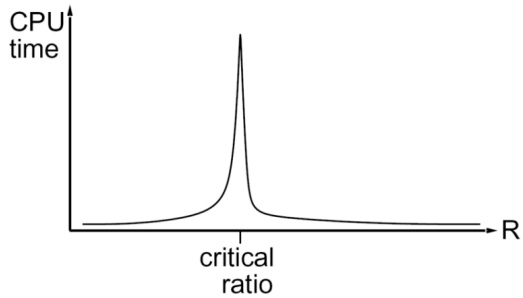
Local Search

As a final topic of interest, backtracking search is not the only algorithm that exists for solving constraint satisfaction problems. Another widely used algorithm is **local search**, for which the idea is childishly simple but remarkably useful. Local search works by iterative improvement - starting with some random assignment to values then repeatedly selecting the variable that violates the most constraints and resetting it to the value that violates the fewest constraints (a policy known as the **min-conflicts heuristic**). Under such a policy, constraint satisfaction problems like N-queens becomes both very time efficient and space efficient to solve. For example, in following example with 4 queens, we arrive at a solution after only 2 iterations:



In fact, local search appears to run in almost constant time and have a high probability of success not only for N-queens with arbitrarily large N, but also for any randomly generated CSP! However, despite these advantages, local search is both incomplete and suboptimal and so won't necessarily converge to an optimal solution. Additionally, there is a critical ratio around which using local search becomes extremely expensive:

$$R = \frac{\text{number of constraints}}{\text{number of variables}}$$



Summary

It's important to remember that constraint satisfaction problems in general do not have an efficient algorithm which solves them in polynomial time with respect to the number of variables involved. However, by using various heuristics, we can often find solutions in an acceptable amount of time:

- *Filtering* - Filtering handles pruning the domains of unassigned variables ahead of time to prevent unnecessary backtracking. The two important filtering techniques we've covered are *forward checking* and *arc consistency*.
- *Ordering* - Ordering handles selection of which variable or value to assign next to make backtracking as unlikely as possible. For variable selection, we learned about a *MRV policy* and for value selection we learned about a *LCV policy*.
- *Structure* - If a CSP is tree-structured or close to tree-structured, we can run the tree-structured CSP algorithm on it to derive a solution in linear time. Similarly, if a CSP is close to tree structured, we can use *cutset conditioning* to transform the CSP into one or more independent tree-structured CSPs and solve each of these separately.