Solving by searching

UFC/DC
CK0031/CK0248
2017.2

Problem solving

Solving agents

Well-definedtness

Problem formulation

Solution search

Search algorithms

Performance metrics

Uninformed search

Breadth-first search

Uniform-cost search

Depth-first search

Depth-limited search

Iterative deepening depth-first search

Bidirectional search

Informed searches

Greedy best-first

A^* search

Memory-bounds

Learning to search

Heuristics

Performance

Admissibility

Learning heuristics

Solving by searching
(CK0031/CK0248)

Francesco Corona

Department of Computer Science
Federal University of Ceará, Fortaleza

Solving by searching

UFC/DC CK0031/CK0248 2017.2

Problem solving

Solving agents
Well-definedtness
Problem formulation

Solution search
Search algorithms
Performance metrics

Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search

Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search

Heuristics
Performance
Admissibility
Learning heuristics

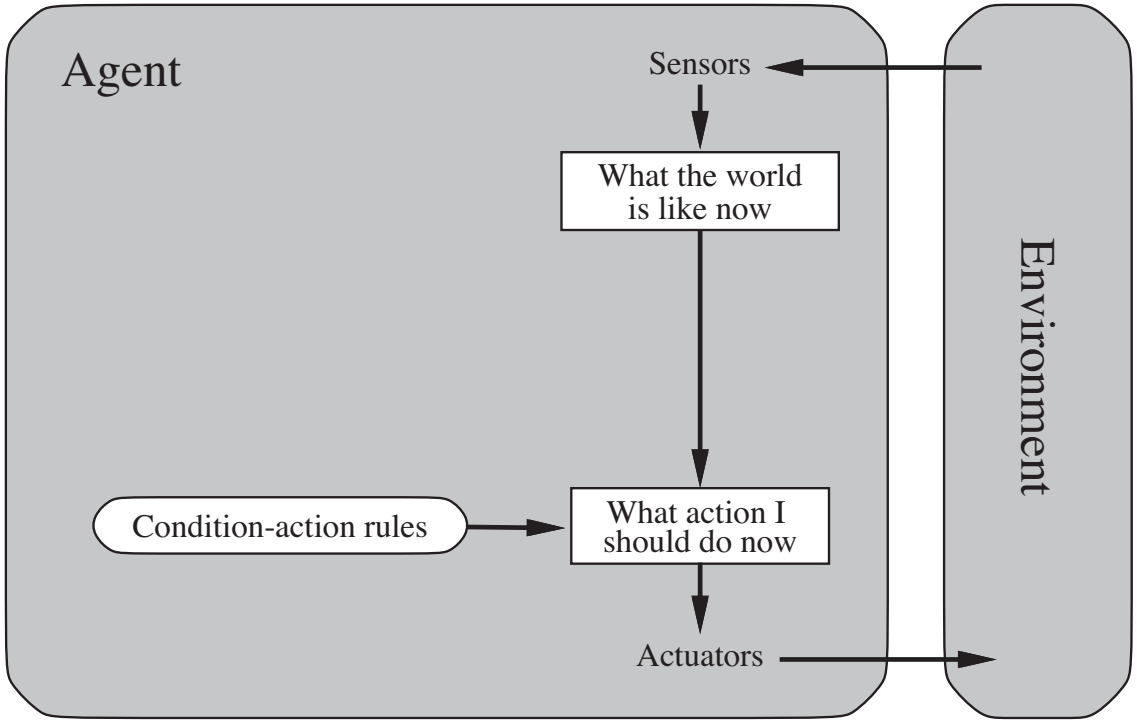
Problem solving

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Problem solving

The simplest agents we discussed thus far are the reflex agents

- Actions based on a direct mapping from states to actions



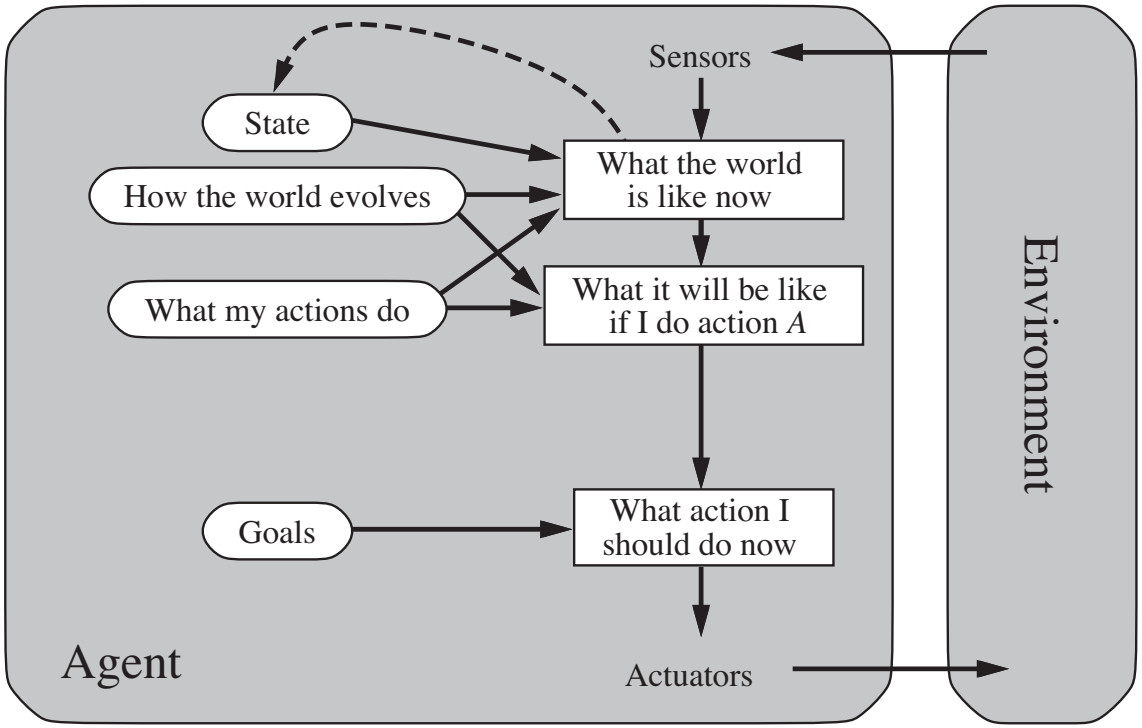
Cannot operate well every environments

1. This mapping may be too large to store
2. This mapping may be too long to learn

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Problem solving (cont.)

Goal-based agents use future actions and desirability of outcomes



Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Problem solving (cont.)

We study one kind of goal-based agent

- **Problem-solving agent**
- They use atomic representations
- (states as wholes, no internal structure visible to the algos)

Goal-based agents that use factored or structured representations

$\rightsquigarrow$ **Planning agents**

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Problem solving (cont.)

We begin with some definitions of problems and their solutions

- Several examples to illustrate these definitions

We describe several general-purpose search algorithms

- They can be used to solve these problems

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Problem solving (cont.)

We discuss several **uninformed search** algorithms

⇝ No info about the problem, other than its definition

Some of these algorithms can solve any solvable problem

- None of them can do so efficiently

We also discuss **Informed search** algorithms

⇝ They can do well, given guidance on where to look for solutions

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Problem solving (cont.)

We consider only the simplest kind of task environment

- The solution is always a fixed sequence of actions

General case (future actions depend on future percepts)

- Not discussed

We shall use the concepts of asymptotic complexity

- $\mathcal{O}$ notation and NP-completeness

Solving by searching

UFC/DC CK0031/CK0248 2017.2

Problem solving

Solving agents

Well-definedtness

Problem formulation

Solution search

Search algorithms

Performance metrics

Uninformed search

Breadth-first search

Uniform-cost search

Depth-first search

Depth-limited search

Iterative deepening depth-first search

Bidirectional search

Informed searches

Greedy best-first

A^* search

Memory-bounds

Learning to search

Heuristics

Performance

Admissibility

Learning heuristics

Problem-solving agents

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Problem-solving agents

Agents are expected to maximize their performance measure

- Achieving this can be sometimes simplified

⇝ The agent needs to adopt a goal

⇝ And it must aim at satisfying it

We look at why and how an agent might want to do this

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Problem-solving agents (cont.)

Example

Imagine an agent in Arad (a city of Romania), enjoying a touring trip

The agent's performance measure contains many factors

- Improve suntan
- Improve Romanian
- Take in the sights
- Enjoy nightlife
- Avoid hangovers
- ...

The decision problem is complex

- Many objectives
- Many trade-offs

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Problem-solving agents (cont.)

Suppose the agent has a nonrefundable ticket

- The following day it must fly out of Bucharest

It makes sense for the agent to adopt an explicit **goal**

$\rightsquigarrow$ Get to Bucharest, by tomorrow

Courses of action that do not reach Bucharest on time can be rejected

- They need no further consideration

The agent's decision problem is greatly simplified

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Problem-solving agents (cont.)

Goals help organise behaviour

The objectives the agent is trying to achieve are limited by goals

$\rightsquigarrow$ The actions the agent needs to consider are also limited

Goal formulation

- This is the first step in problem solving

Based on current situation and agent's performance measure

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Problem-solving agents (cont.)

Definition

We will consider a ***goal*** *to be a set of world states*

- *Those states in which the goal is satisfied*

The agent's task is to find out how to act so that a goal state is reached

$\rightsquigarrow$ *Actions, current and future*

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Problem-solving agents (cont.)

The agent needs to decide what sorts of actions and states to consider

The level of detail of the actions is critical

- 'Move left foot forward an inch' or 'turn steering wheel one degree left'
- The agent would probably never find its way out of the parking lot

At that level of detail there is too much uncertainty in the world

- There would be too many steps in a solution

Problem-solving agents (cont.)

Definition

Problem formulation

It is the process of deciding what actions and states to consider

- *Given a goal*

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Problem-solving agents (cont.)

Example

Assume actions at the level of driving from one major town to another

- ⇝ Each state corresponds to being in a some town

The agent has adopted the goal of driving **to Bucharest**

- It is considering where to go, **from Arad**

Three roads lead out of Arad: to Sibiu, to Timisoar and to Zerind

- None of these (is) achieves the goal

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Problem-solving agents (cont.)

Unless the agent knows Romania, it will not know which road to follow

- The agent does not know which of its possible actions is best

It does not know about the state that results from taking actions

- With no extra information, the **environment** is **unknown**

$\rightsquigarrow$ No choice but to try one action at random

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Problem-solving agents (cont.)

Example

Suppose the agent has a map of Romania

The map provides the agent with information

- The states it might get itself into
- The actions it can take

The agent can use this info to consider subsequent stages

⇝ A hypothetical journey via each of the three towns

⇝ Find a journey that eventually gets to Bucharest

Once the agent has found a path on the map from Arad to Bucharest

⇝ It must achieve the goal

⇝ Done by carrying out the actions

⇝ Drive the individual legs

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Problem-solving agents (cont.)

An agent with immediate options of unknown value can decide what to do

- By *examining future actions*
- Action that eventually lead to states of known value

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Problem-solving agents (cont.)

Assumption

Environment is observable

$\rightsquigarrow$ Agent always knows the current state

- True, for each city on the map has a sign indicating its presence

Environment is discrete

$\rightsquigarrow$ At any given state, only finitely many actions to choose from

- True, for each city is connected to a *small* number of other cities

Environment is known

$\rightsquigarrow$ Agent knows which states are reached by each action

- True, for an accurate map meets this condition for navigation

Environment is deterministic

$\rightsquigarrow$ Each action has one outcome

- Ideally true, for choosing to drive from Arad to Sibiu leads to Sibiu

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Problem-solving agents (cont.)

Under these assumptions, the solution is a fixed sequence of actions

- In general, solution could be a branching strategy

$\rightsquigarrow$ Different future actions depending on percepts

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Problem-solving agents (cont.)

Example

Consider sub-ideal conditions

The agent may plan to drive from Arad to Sibiu and then to Rimnicu Vilcea

- It gets by accident to Zerind instead of Sibiu

A contingency plan is needed, in case this can happen

The agent knows initial state, the environment is known and deterministic

⇝ The agent knows exactly where it will be after the first action

- (and it know what it will perceive)

Only one percept is possible after the first action, ...

- The solution can specify only one possible second action

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Problem-solving agents (cont.)

Definition

Search

The process of looking for a sequence of actions that reaches a goal

- *The search algorithm takes a problem as input and returns a* ***solution***
- *The solution is in the form of an action sequence*

Execution

Once a solution is found, the recommended actions can be carried out

Simple design for the agent

- Formulate $\Longrightarrow$ Search $\Longrightarrow$ Execute

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Problem-solving agents (cont.)

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

  state ← UPDATE-STATE(*state*, *percept*)
  if *seq* is empty **then**
    goal ← FORMULATE-GOAL(*state*)
    problem ← FORMULATE-PROBLEM(*state*, *goal*)
    seq ← SEARCH(*problem*)
    if *seq* = *failure* **then return** a null action
  action ← FIRST(*seq*)
  seq ← REST(*seq*)
  return *action*

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Problem-solving agents (cont.)

While the agent is executing the solution sequence it ignores its percepts

- For choosing an action they are irrelevant
- The agent knows in advance

Such an agent must be certain of what is going on

Control theorists call this an **open-loop** system

- Ignoring the percepts breaks the loop between agent and environment

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Problem-solving agents (cont.)

After formulating a goal and a problem to solve

- ⇝ The agent calls a search procedure to solve it

Then, it uses the solution to guide its actions

- ⇝ The agent does what the solution recommends as next thing to do
  - Typically, first action of the sequence
  - Then, that step is removed from sequence

Once the solution has been executed

- ⇝ The agent will formulate a new goal

Solving by searching

UFC/DC
CK0031/CK0248
2017.2

Problem solving

Solving agents

Well-definedtness

Problem formulation

Solution search

Search algorithms

Performance metrics

Uninformed search

Breadth-first search

Uniform-cost search

Depth-first search

Depth-limited search

Iterative deepening depth-first search

Bidirectional search

Informed searches

Greedy best-first

A^* search

Memory-bounds

Learning to search

Heuristics

Performance

Admissibility

Learning heuristics

Well-definedtness
Problem solving agents

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Well-definedtness

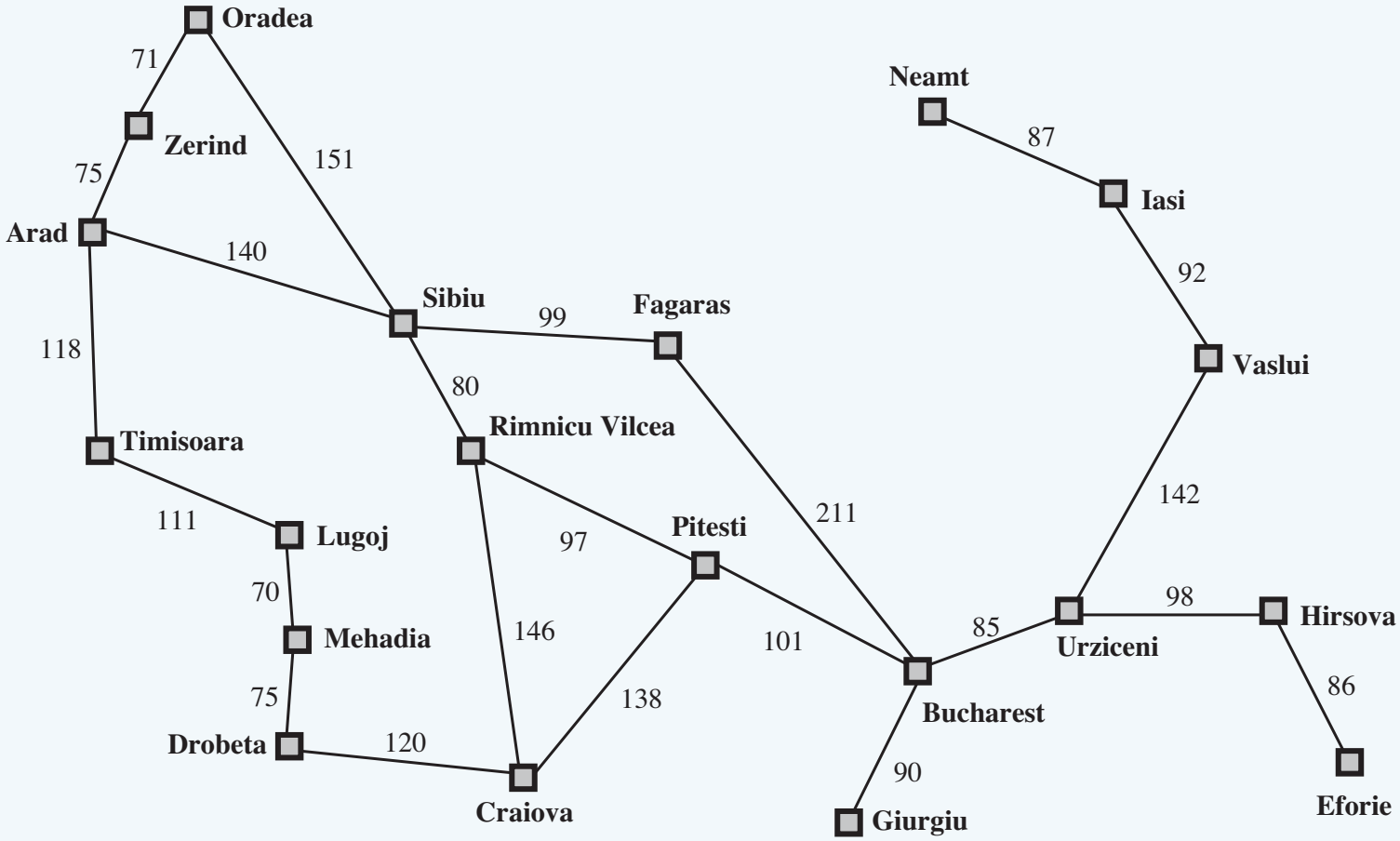
A **problem** can be defined formally by five components

1. **Initial state**
2. **Actions**
3. **Transition model**
4. **Goal test**
5. **Path cost**

The **initial state** is the initial state that the agent starts in

Example

Initial state for agent in Romania can be described as `In(Arad)`



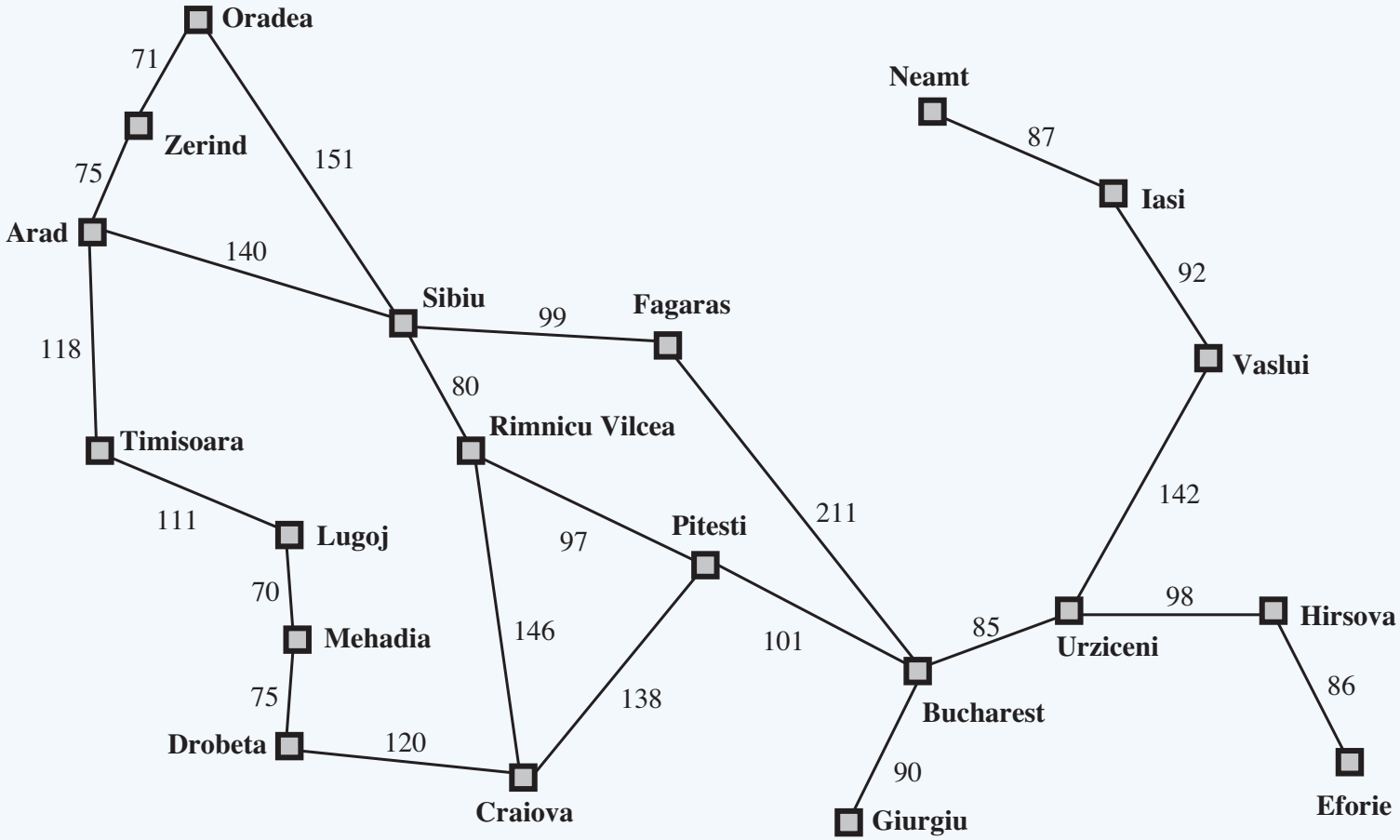
Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

The **actions** are the possible actions available to the agent

Function `ACTIONS(s)` returns the set of actions that can be executed in `s`

- Given a particular state `s`

Example

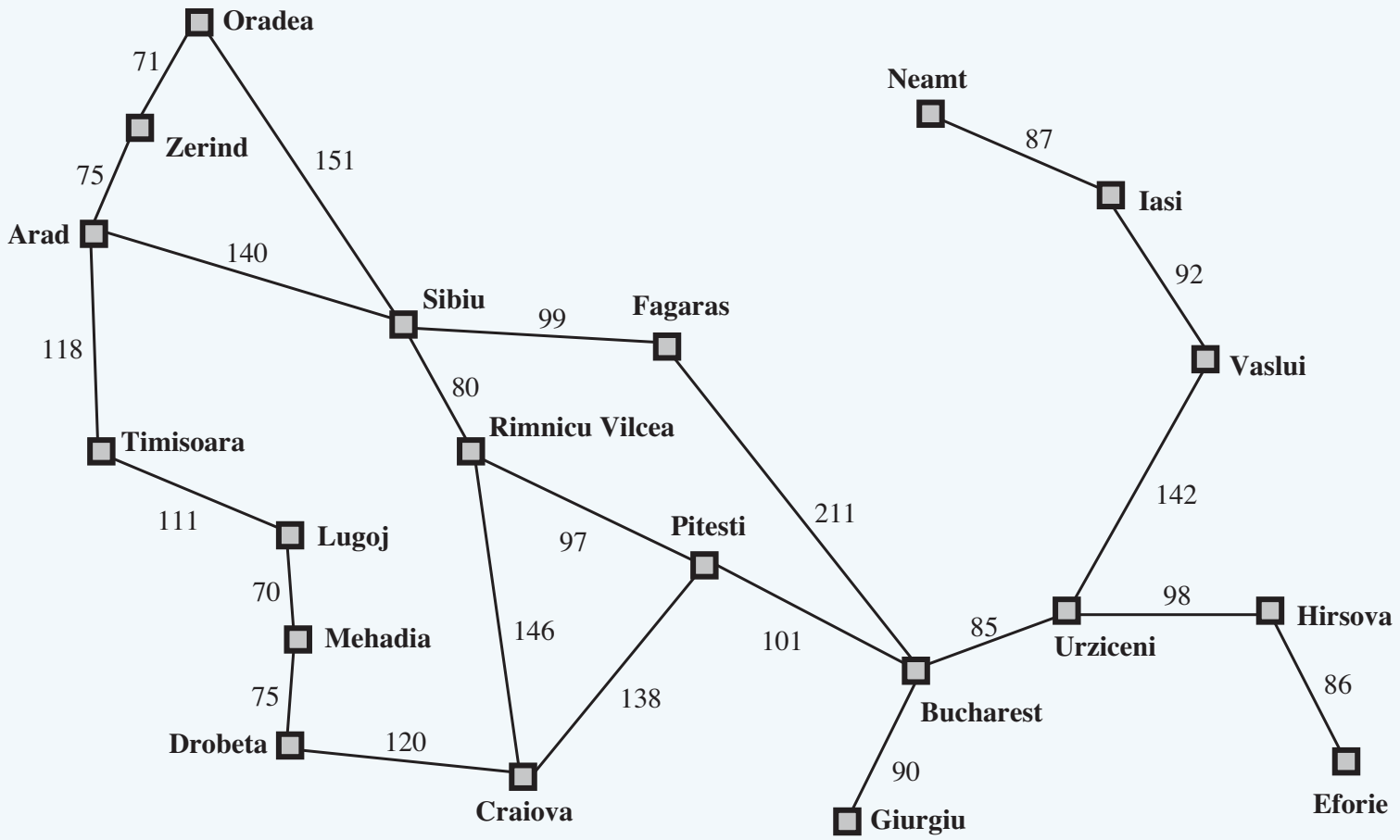


From state `In(Arad)`, actions {`Go(Sibiu)`, `Go(Timisoara)`, `Go(Zerind)`}

A **transition model** is formal description of what each action does

Function `RESULT(s, a)` returns the state from doing action `a` in state `s`

Example



`RESULT(In(Arad), Go(Zerind))` = `In(Zerind)`

Successor state: Any state reachable from a given state, by a single action

Well-definedtness (cont.)

Definition

Together, initial state, actions, and transition model implicitly define the ***state space*** *of the problem*

The set of all states reachable from the initial state by any sequence of actions

The state space forms a **directed network** or **graph**

- The nodes are states
- The links between nodes are actions

The map of Romania can be interpreted as a state-space graph

- Each road stands for two driving actions
- (one in each direction)

Well-definedtness (cont.)

Definition

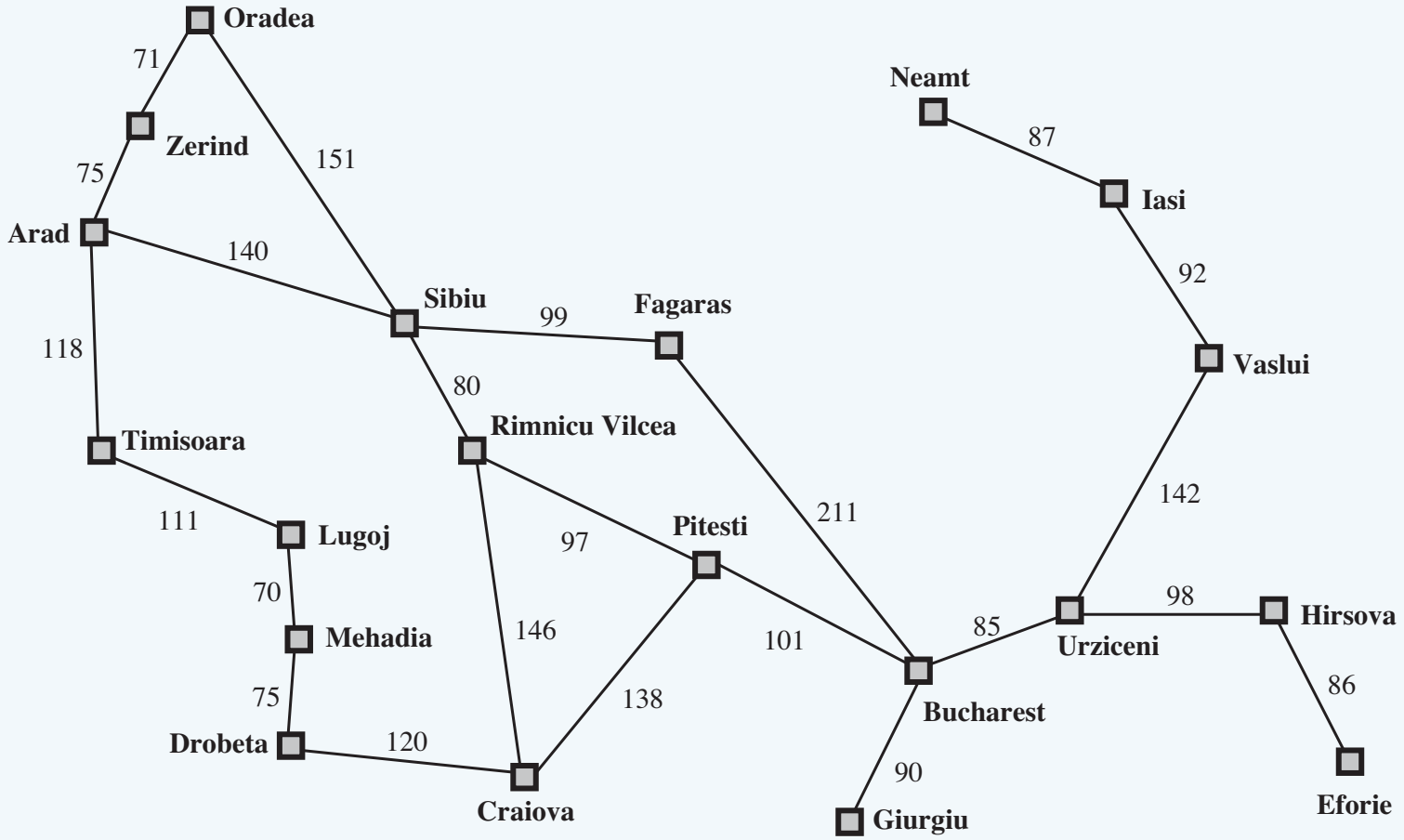
*A **path** in state space*

This is a sequence of states connected by a sequence of actions

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

The **goal test** determines (checks) whether a given state is a goal state

Example



The agent's goal in Romania is the singleton set {`In(Bucharest)`}

Well-definedtness (cont.)

Sometimes the goal is specified by an abstract property

- Rather than an explicitly enumerated set of states

In the game of chess, the goal is to reach a state called ‘checkmate’

- The opponent’s king is under attack and can’t escape

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Well-definedtness (cont.)

A **path cost** function assigns a numeric cost to each path

- For problem-solving agents, the cost function reflects the agent's performance measure

The path cost is the sum of costs of individual actions along path

Example

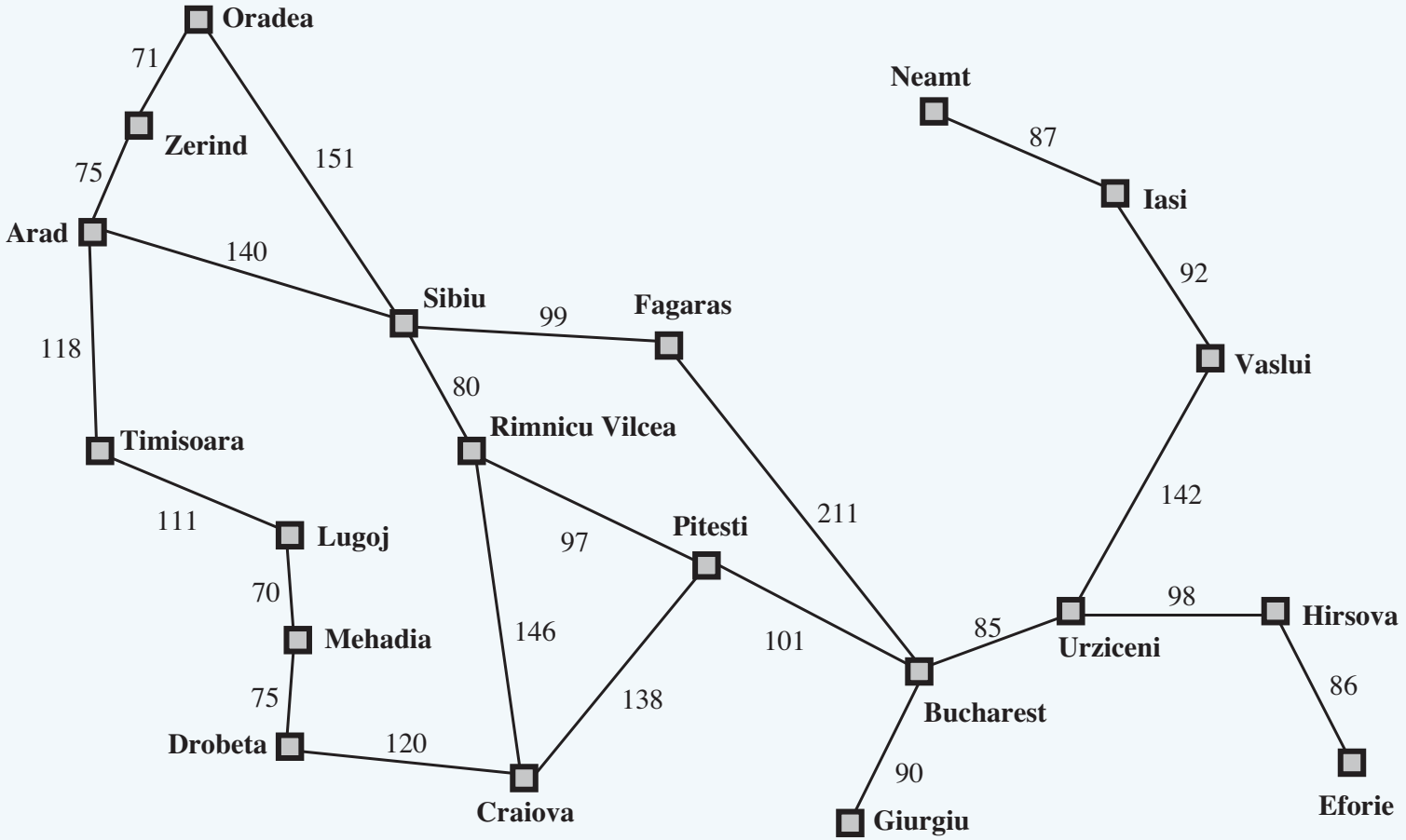
For the agent trying to get to Bucharest, time is essential

- The cost of a path might be its length in kilometres

Well-definedtness (cont.)

Step cost $c(\texttt{s}, \texttt{a}, \texttt{s'})$ of action `a` in state `s` to reach state `s'` is non-negative

Example



Step costs for Romania can be defined as route distances

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Well-definedtness (cont.)

These elements define a problem

- They can be gathered into a single data structure

⇝ Passable as input to a problem-solving algorithm

Definition

*A **solution** to a problem is an action sequence*

⇝ *From the initial state to a goal state*

***Solution quality** is measured by the path cost function*

*An **optimal solution** has the lowest path cost among all solutions*

Solving by searching

UFC/DC CK0031/CK0248 2017.2

Problem solving

Solving agents

Well-definedtness

Problem formulation

Solution search

Search algorithms

Performance metrics

Uninformed search

Breadth-first search

Uniform-cost search

Depth-first search

Depth-limited search

Iterative deepening depth-first search

Bidirectional search

Informed searches

Greedy best-first

A^* search

Memory-bounds

Learning to search

Heuristics

Performance

Admissibility

Learning heuristics

Problem formulation

Problem solving agents

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Problem formulation

Example

The proposed formulation of the problem of getting to Bucharest

- Initial state, actions, transition model, goal test, and path cost

This formulation seems reasonable

- It is still a model (an abstract mathematical description)
- It is not the real thing

Compare an actual trip to the chosen state description

- `In(Arad)`

The state of the world is the real state

- Traveling companions, current radio program
- Proximity of law enforcement officers
- Condition of the road, weather
- Distance to the next rest stop
- Scenery out of the window, ...

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Problem formulation (cont.)

All these considerations are left out of state descriptions

- They are irrelevant to the problem of finding a route to Bucharest

Abstraction: Process of removing detail from a representation

In addition to abstracting state description, we abstract actions

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Problem formulation (cont.)

Example

Driving has many effects (besides changing location of vehicle/occupants)

- It takes up time, consumes fuel, generates pollution
- And, it changes the agent (travel is broadening)

Our formulation takes into account only changes in location

There are many actions that we necessarily omit altogether

- Turning on the radio
- Looking out of the window
- Slowing down for law enforcement officers, ...

No action specification at level of 'turn steering wheel to the left by 1 degree'

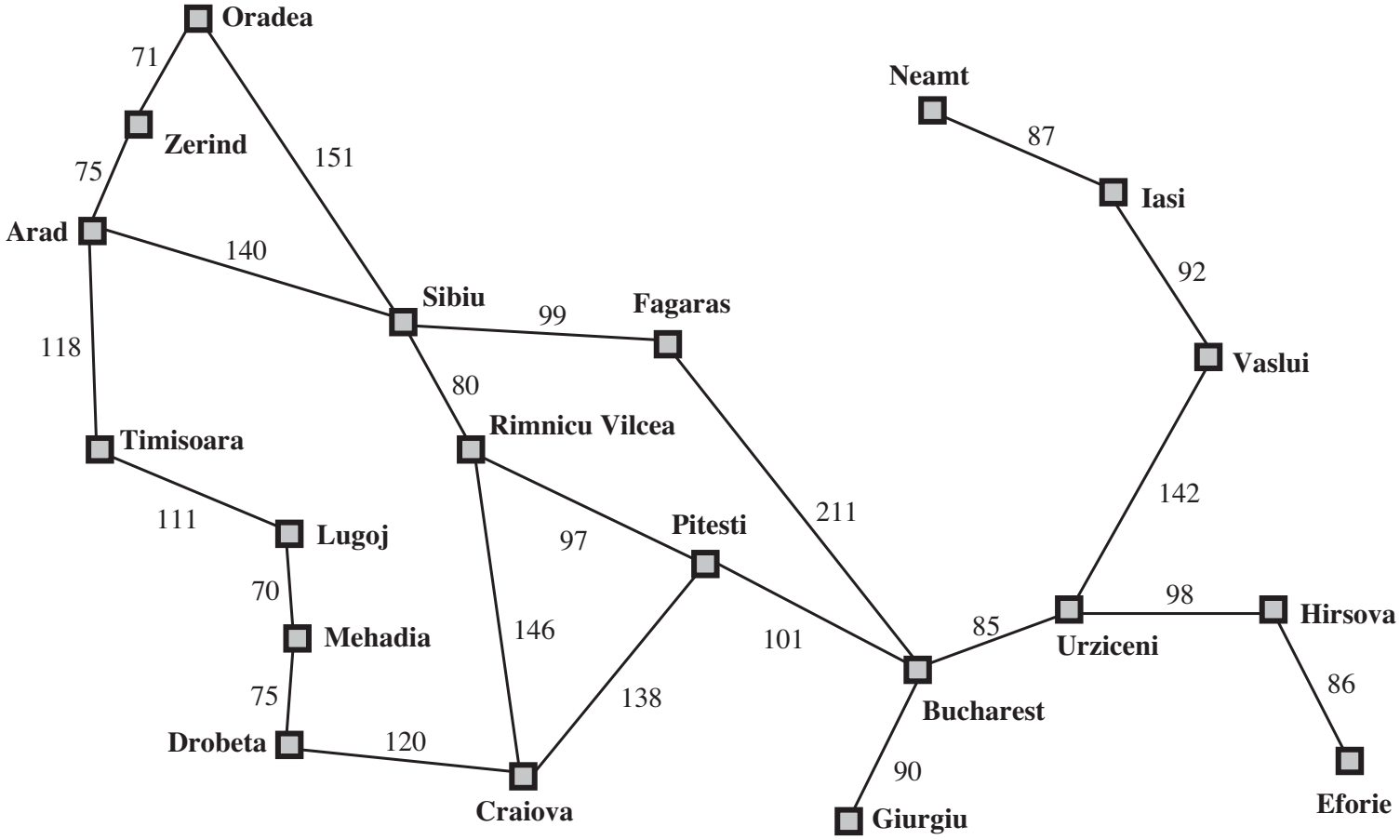
Problem formulation (cont.)

More precise about defining the appropriate level of abstraction?

- Abstract states and actions we chose can be understood as seen to large sets of detailed world states and detailed action sequences
- Now consider a solution to the abstract problem

Problem formulation (cont.)

Path from Arad to Sibiu, to Rimnicu Vilcea, to Pitesti, to Bucharest



This abstract solution corresponds to a number of more detailed paths

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Problem formulation (cont.)

- We could drive with the radio on between Sibiu and Rimnicu Vilcea
- We could then switch it off for the rest of the trip

Problem formulation (cont.)

Definition

The abstraction is valid if we can expand any abstract solution into a solution in the more detailed world

A sufficient condition

For every detailed state that is '`in(Arad)`,' there is a detailed path to some state that is '`in(Sibiu)`,' and so on

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Problem formulation (cont.)

What makes the abstraction useful?

Carrying out the actions in the solution must be easier than original problem

- In this case, they are easy enough that they can be carried out
- Without further search or planning by an average driving agent

Remark

On the choice of a good abstraction

Remove as much detail as possible, while retaining validity

- Ensure that abstract actions are easy to carry out

W/O ability to construct abstractions, agents are swamped by real world

Examples
Problem-solving agents

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Examples

Problem-solving has been applied to an array of task environments

Toy problems

$\rightsquigarrow$ To illustrate/exercise problem-solving methods

- It can be given a concise, exact description
- Usable to compare the performance of algorithms

Real-world problem

$\rightsquigarrow$ To solve tasks people actually care about

- It tends not to have a single agreed-upon description
- Just a general flavour of its formulation

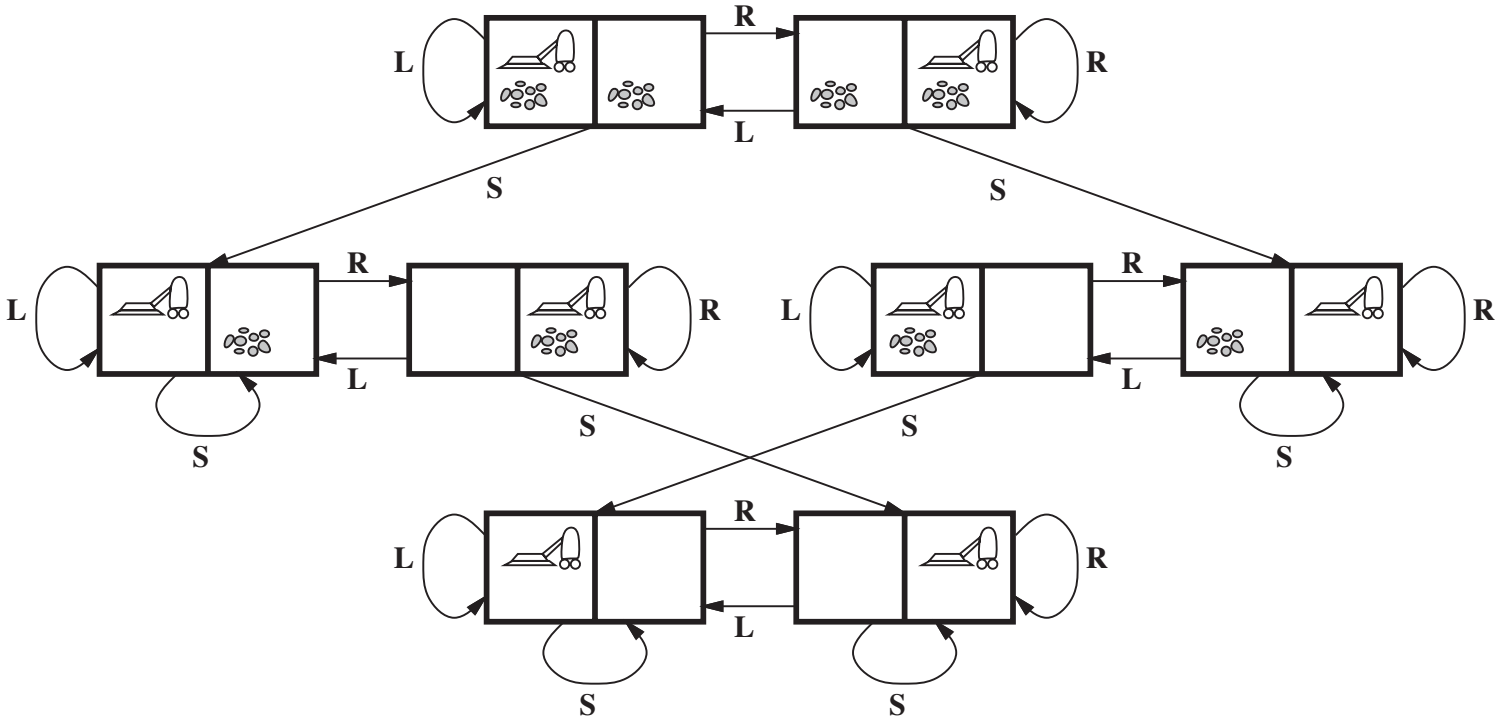
Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Examples - Toy problems

Consider a formulation of the **vacuum cleaner world**

States: State is determined by agent location and dirt locations

- The agent is in one of two locations
- Each might or might not contain dirt



There are $2 \times 2^2 = 8$ possible world states

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

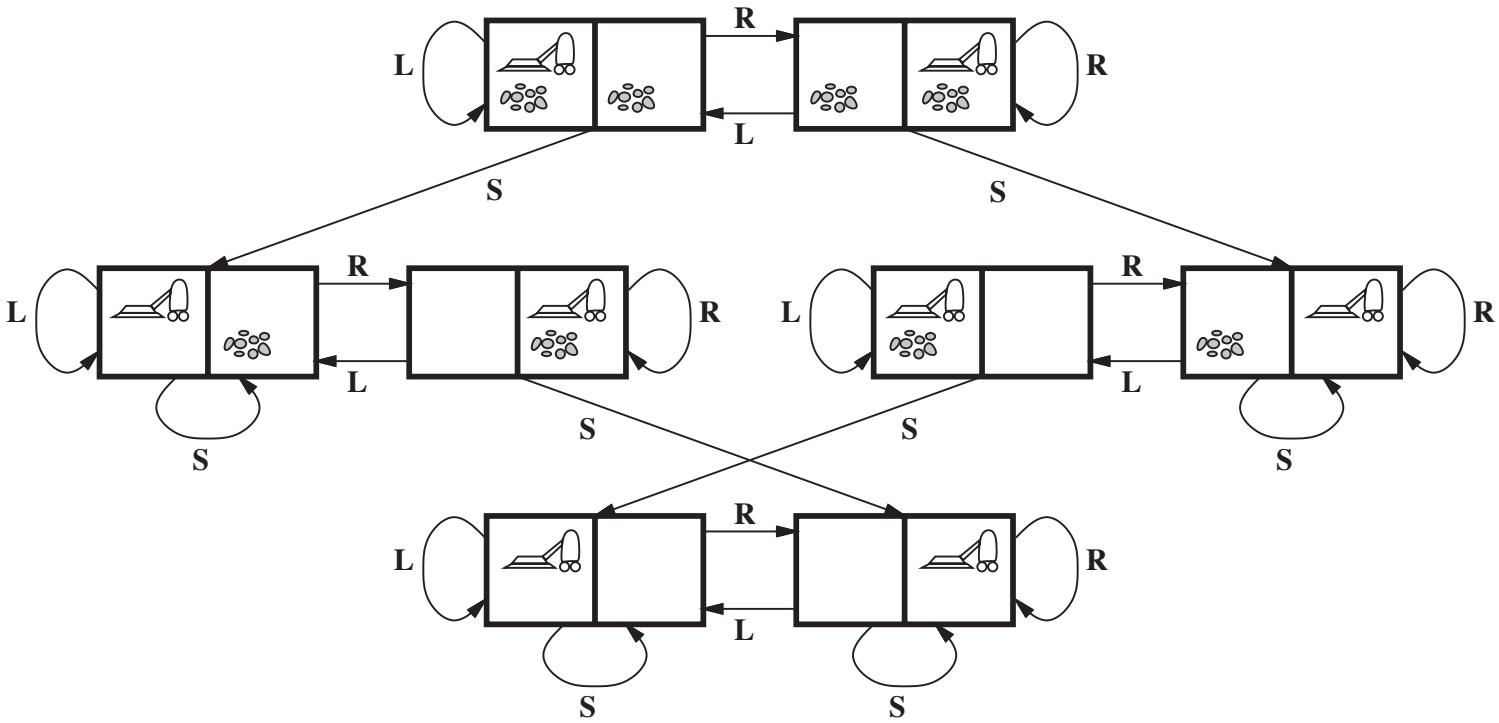
Examples - Toy problems (cont.)

Remark

- A larger environment with n locations has $n \times 2^n$ states

Examples - Toy problems (cont.)

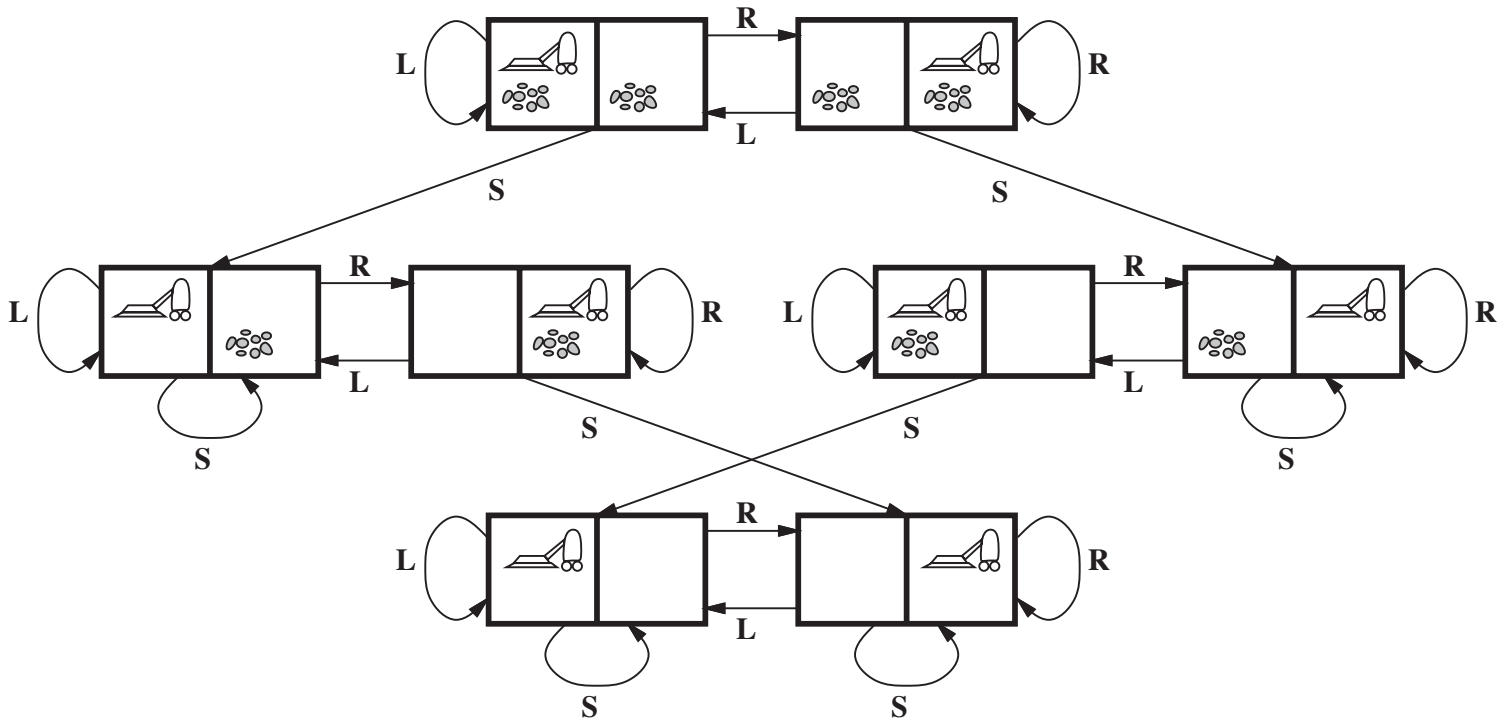
Initial state: Any state can be designated as the initial state



Actions: Each state has three actions, `Left`

- `Right`, and `Suck`

Examples - Toy problems (cont.)

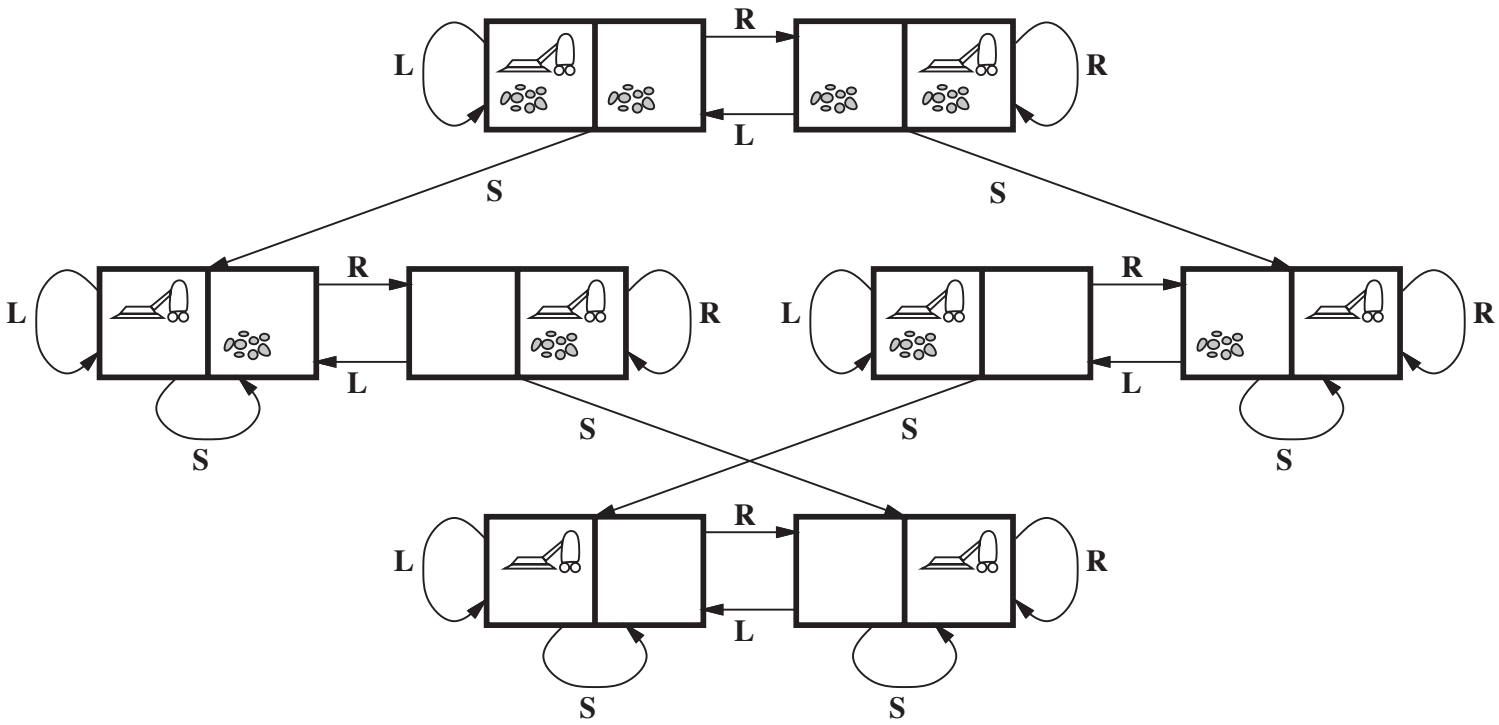


Transition model: Actions have expected effects

- Except moving `Left` in leftmost square, moving `Right` in rightmost square, and `Sucking` in clean square have no effect

Examples - Toy problems (cont.)

Goal test: This checks whether all the squares are clean



Path cost: Each step costs 1

- Path cost is the number of steps in the path

Examples - Toy problems (cont.)

Conversely to real world, in the toy problem

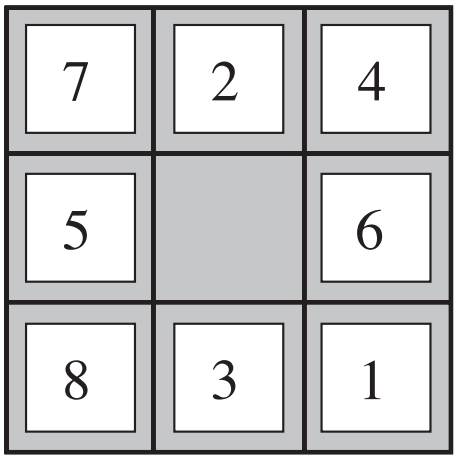
- Discrete locations
- Discrete dirt
- Reliable cleaning
- It never gets any dirtier

Some of these assumptions can be relaxed

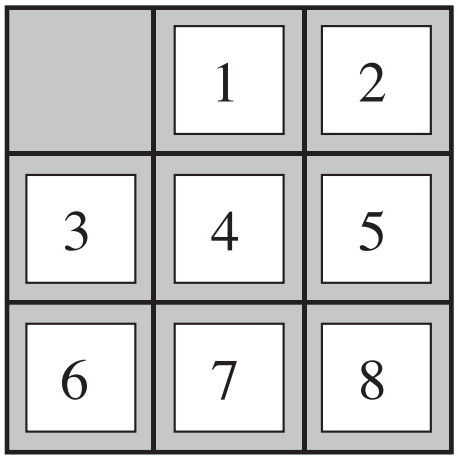
Examples - Toy problems (cont.)

The 8-puzzle is of a 3×3 board, 8 numbered tiles and a blank space

- A tile adjacent to the blank space can slide into the space
- The objective is to reach a specified goal state



Start State



Goal State

Examples - Toy problems (cont.)

States

- Location of each of the eight tiles and the blank
- In one of the nine squares

Initial state

- Any state can be designated as the initial state[1]

Actions

- Actions as movements of the blank space `Left`, `Right`, `Up`, or `Down`[2]

Transition model

- Given state and action, the resulting state is returned [3]

Goal test

- This checks whether the state matches goal configuration

Path cost

- Each step costs 1, path cost is the number of steps in path

---

[1] Any goal can be reached from exactly half of the possible initial states
[2] Different subsets of these are possible depending on where the blank is
[3] Apply `Left` to the start state, start state and blank are switched

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Examples - Toy problems (cont.)

What abstractions have we included here?

Actions are abstracted to their beginning and final states

- Intermediate locations (block is sliding) are ignored

We abstracted away some actions (shaking the board when pieces get stuck)

We ruled out extracting the pieces with a knife and putting them back

Remark

We have a description of the rules of the 8-puzzle

We avoid all the details of physical manipulations

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Examples - Toy problems (cont.)

The 8-puzzle belongs to the family of **sliding-block puzzles**

- Often used as test problems for new search algorithms
- Known to be NP-complete

The 8-puzzle (3×3 board) has 9!/2=181,440 reachable states (easily solved)

- The 15-puzzle (4×4 board) has around 1.3 trillion states
- Random instances solved optimally in milliseconds
- (by the best search algorithms)

The 24-puzzle (5×5 board) has around 10^{25} states

- Random instances take several hours to solve optimally

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

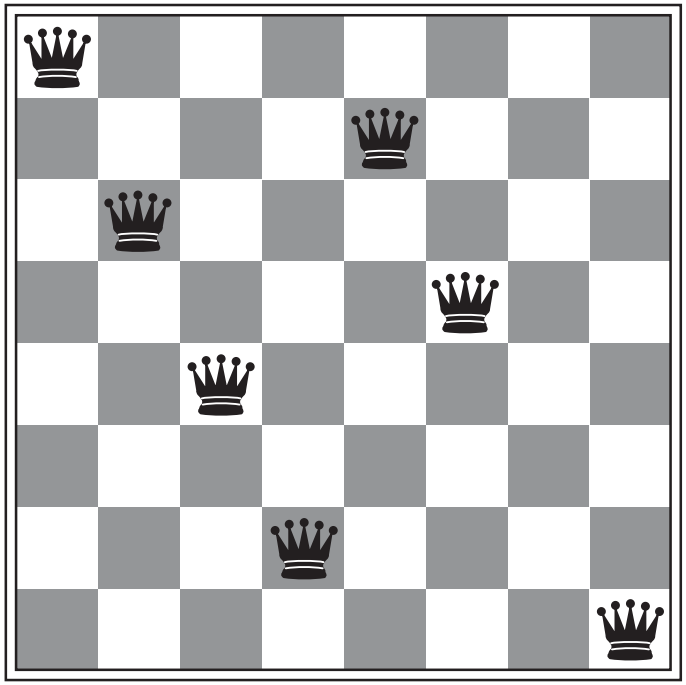
Examples - Toy problems (cont.)

The goal of the 8-**queens** problem is to place eight queens on a chessboard

⇝ No queen must attack any other queen

A queen attacks any piece in the same row, column or diagonal

Queen in the rightmost column is attacked by queen at the top left



Efficient special-purpose algorithms exist

- For the whole n-**queens** family
- Usseful test for search algorithms

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Examples - Toy problems (cont.)

There are two main kinds of formulation

An **incremental formulation**

- Augment the state description, starting with an empty state

A **complete-state formulation**

- Start with all 8 queens on the board and move them around

In either case, the **path cost is of no interest**

- Only the final state matters

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Examples - Toy problems (cont.)

An **incremental formulation**

States

- Any arrangement of 0 to 8 queens on the board is a state

Initial state

- No queens on the board

Actions

- Add a queen to any empty square

Transition model

- Returns the board with a queen added to the specified square

Goal test

- 8 queens are on the board, none attacked

Possible sequences to investigate: $64 \cdot 63 \cdot \cdots \cdot 57 \approx 1.8 \times 10^{14}$

Examples - Toy problems (cont.)

Prohibit placing a queen in any square that is already attacked

States

- All possible arrangements of n queens ($0 \leq n \leq 8$), one per column in the leftmost n columns, with no queen attacking another

Actions

- Add a queen to any square in the leftmost empty column such that it is not attacked by any other queen

This formulation reduces the 8-queens state space

- From 1.8×10^{14} to 2057
- Solutions are easy to find

Examples - Real-world problems

The formulation of the **route-finding problem**

- Given locations and transitions along links between them

Route-finding algorithms are used in a variety of applications

- Websites and in-car systems that provide driving directions

They are extensions of the Romania example

- Routing video streams in computer nets
- Airline travel-planning systems
- Military operations planning
- ...

Much more complex specifications

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Examples - Real-world problems (cont.)

Consider the airline travel task solved by **travel-planning** sites

States

- Each state includes a location (e.g., an airport) and current time
- The state must record extra info about 'temporal/spacial' aspects
- The cost of an action (flight segment) may depend on previous segments, fare bases, and status as domestic/international

Initial state

- Specified by the user's query

Actions

- Any flight from current location, in any seat class, leaving after current time and leaving enough time for within-airport transfer if needed

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Examples - Real-world problems (cont.)

Transition model

- The state resulting from taking a flight will have the flight's destination as the current location and the flight's arrival time as the current time

Goal test

- Is it the final destination specified by the user?

Path cost

- This depends on monetary cost, waiting time, flight time, customs/immigration procedures, seat quality, time of day, type of airplane, frequent-flyer mileage awards, ...

Examples - Real-world problems (cont.)

Commercial travel advice systems use a similar formulation

A really good system should include contingency plans

- (such as backup reservations on alternate flights)

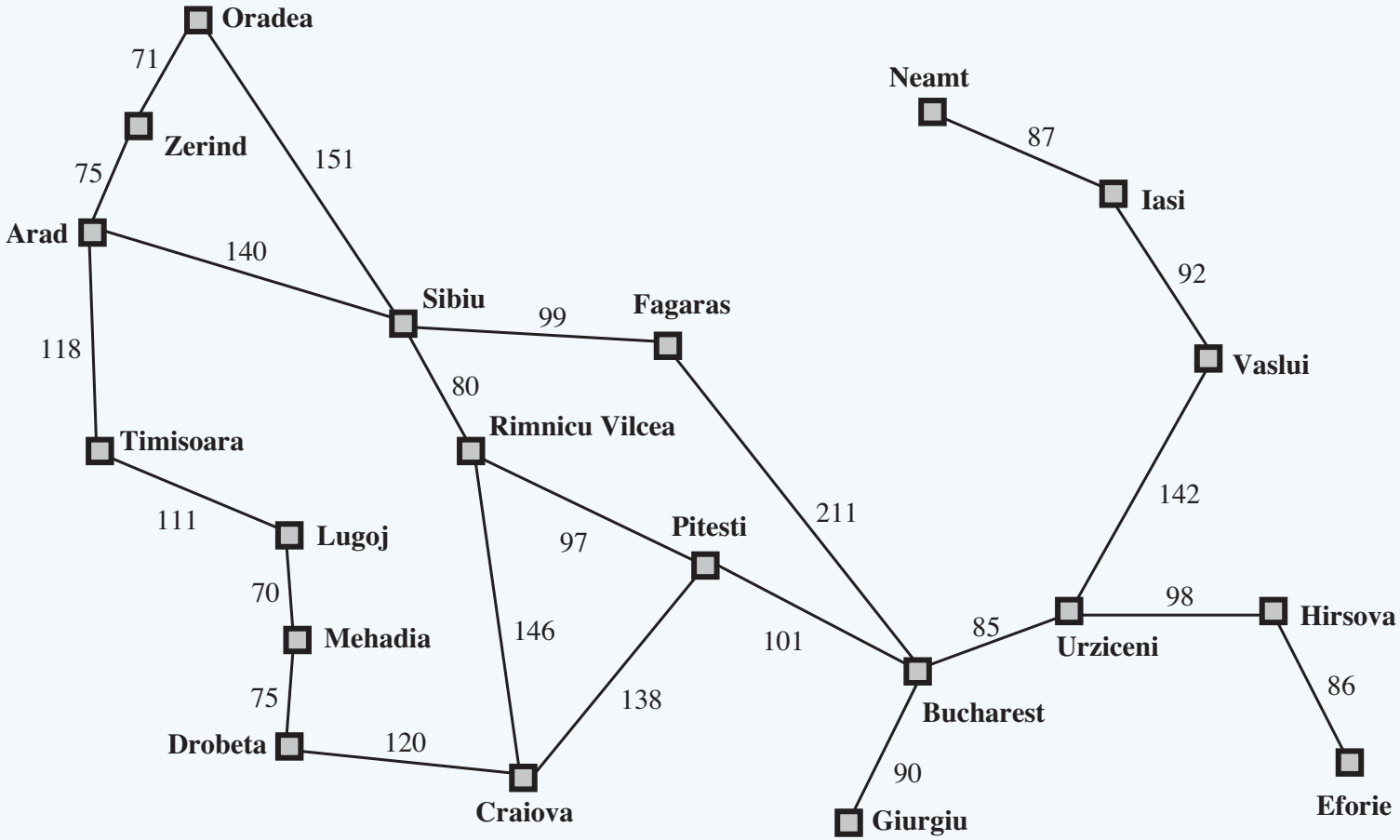
Justified by cost and likelihood of failure of original plan

Examples - Real-world problems (cont.)

Touring problems are closely related to route-finding problems

Example

'*Visit every city at least once, starting and ending in Bucharest*'



Examples - Real-world problems (cont.)

The actions correspond to trips between adjacent cities

Each state must include current location and visited cities

- Initial state: `In`(`Bucharest`), `Visited`({`Bucharest`})
- `In`(`Vaslui`), `Visited`({`Bucharest`, `Urziceni`, `Vaslui`})

The goal test would check whether the agent is in Bucharest

- And, all 20 cities have been visited

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Examples - Real-world problems (cont.)

The **traveling salesperson** problem (**TSP**) is a touring problem

- Each city must be visited exactly once
- The aim is to find the shortest tour
- The problem is known to be NP-hard

Efforts have been expended to improve TSP algorithms

These algorithms have been used for tasks such as planning movements of automatic circuit-board drills and of stocking machines on shop floors

$\rightsquigarrow$ Not only planning trips for traveling salespersons

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Examples - Real-world problems (cont.)

The **VLSI layout** problem

It requires positioning components and connections on a chip to minimize area, circuit delays, stray capacitances, and maximise manufacturing yield

The layout problem comes after the logical design phase

It is usually split into two parts: 1) cell layout and 2) channel routing

- In cell layout, primitive components of the circuit are grouped into cells, each of which performs some function
- Each cell has a fixed footprint (size and shape) and requires a certain number of connections to each of the other cells

The aim is to place cells on chip so that they do not overlap

- There must be room for connecting wires between cells

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Examples - Real-world problems (cont.)

Robot navigation is a generalisation of the route-finding problem

- A robot can move in a continuous space
- (in principle, an infinite set of possible actions and states)
- Rather than following a discrete set of routes

For a circular robot moving on a flat surface, the space is two dimensional

- The robot has arms and legs or wheels that must be controlled
- Search space is many dimensional

Examples - Real-world problems (cont.)

An important assembly problem is **protein design**

The goal is to find a sequence of amino acids that will fold into a three dimensional protein with the right properties to cure some disease

Solving by searching

UFC/DC
CK0031/CK0248
2017.2

Problem solving

Solving agents

Well-definedtness

Problem formulation

Solution search

Search algorithms

Performance metrics

Uninformed search

Breadth-first search

Uniform-cost search

Depth-first search

Depth-limited search

Iterative deepening depth-first search

Bidirectional search

Informed searches

Greedy best-first

A^* search

Memory-bounds

Learning to search

Heuristics

Performance

Admissibility

Learning heuristics

Searching for solutions

Solving by searching

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Searching for solutions

Having formulated some problems, we now need to solve them

$\rightsquigarrow$ A solution is an action sequence

Search algorithms work by considering possible action sequences

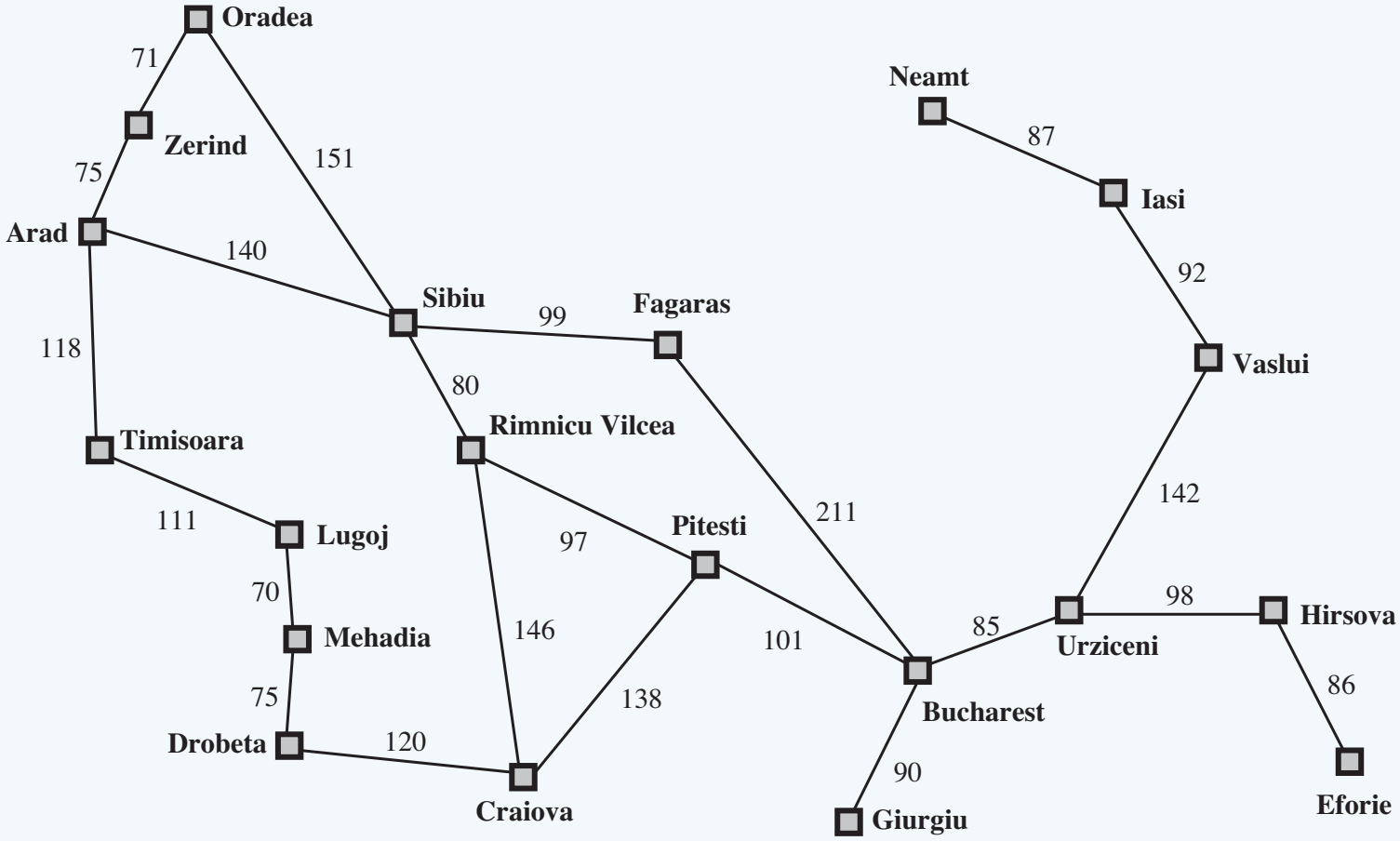
Possible action sequences starting at initial state form a **search tree**

- The initial state at the root
- The **branches are actions**
- The **nodes are states**

Searching for solutions (cont.)

Example

The **root node** is the **initial state** `In(Arad)`



First step: Check whether this is the goal state

Searching for solutions (cont.)

The initial state is not the goal state

$\rightsquigarrow$ We need to take actions

Definition

*We do this by **expanding** the current state*

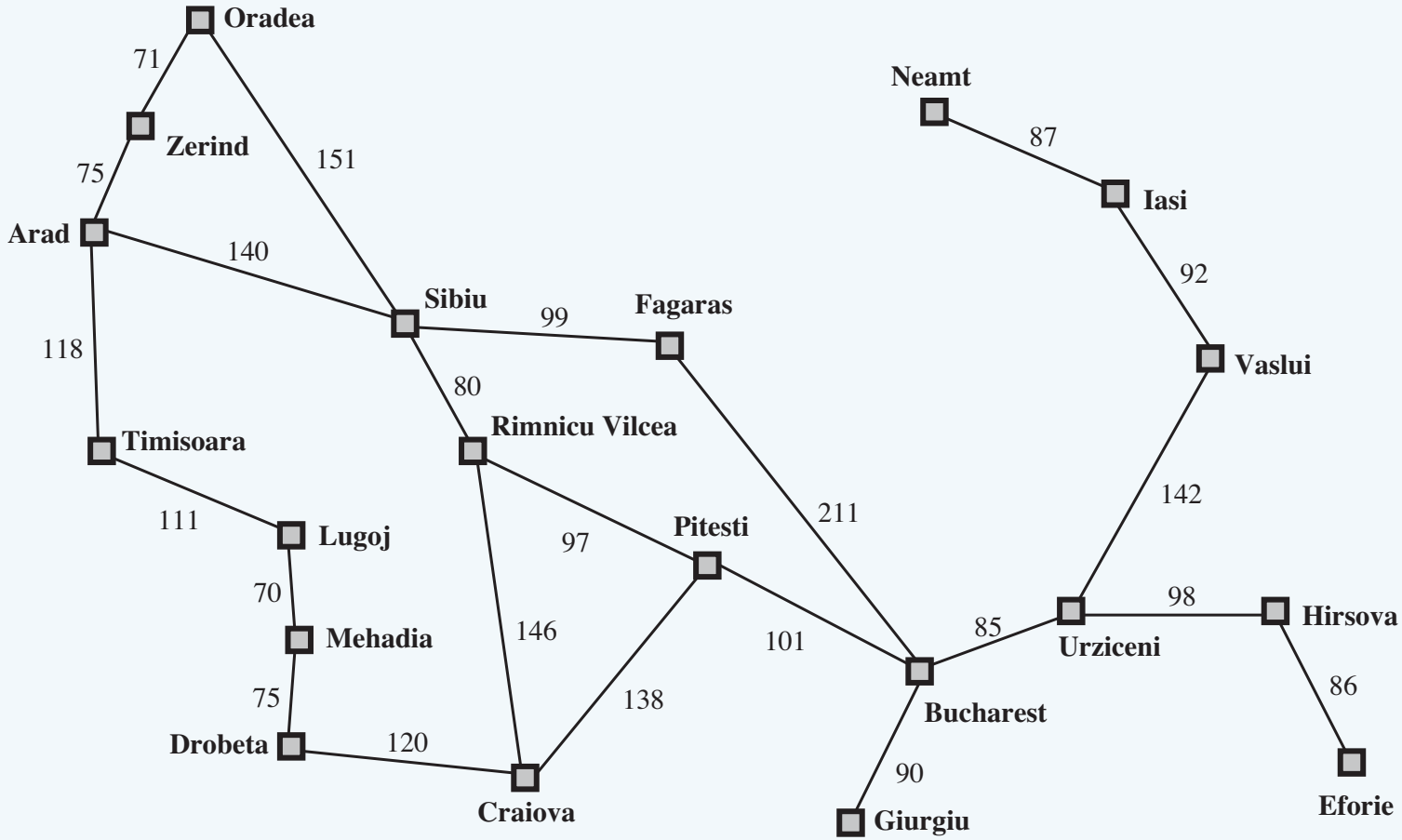
*Apply each legal action to current state, **generate** a new set of states*

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Example

Three branches from **parent node** `In(Arad)` to three **child nodes**

- `In(Sibiu)`, `In(Timisoara)`, and `In(Zerind)`



Searching for solutions (cont.)

We must choose which of these options to consider further

- If the first choice is not a solution

Follow up one option and put the others aside for later

This is the essence of search

Searching for solutions (cont.)

Suppose we choose Sibiu first

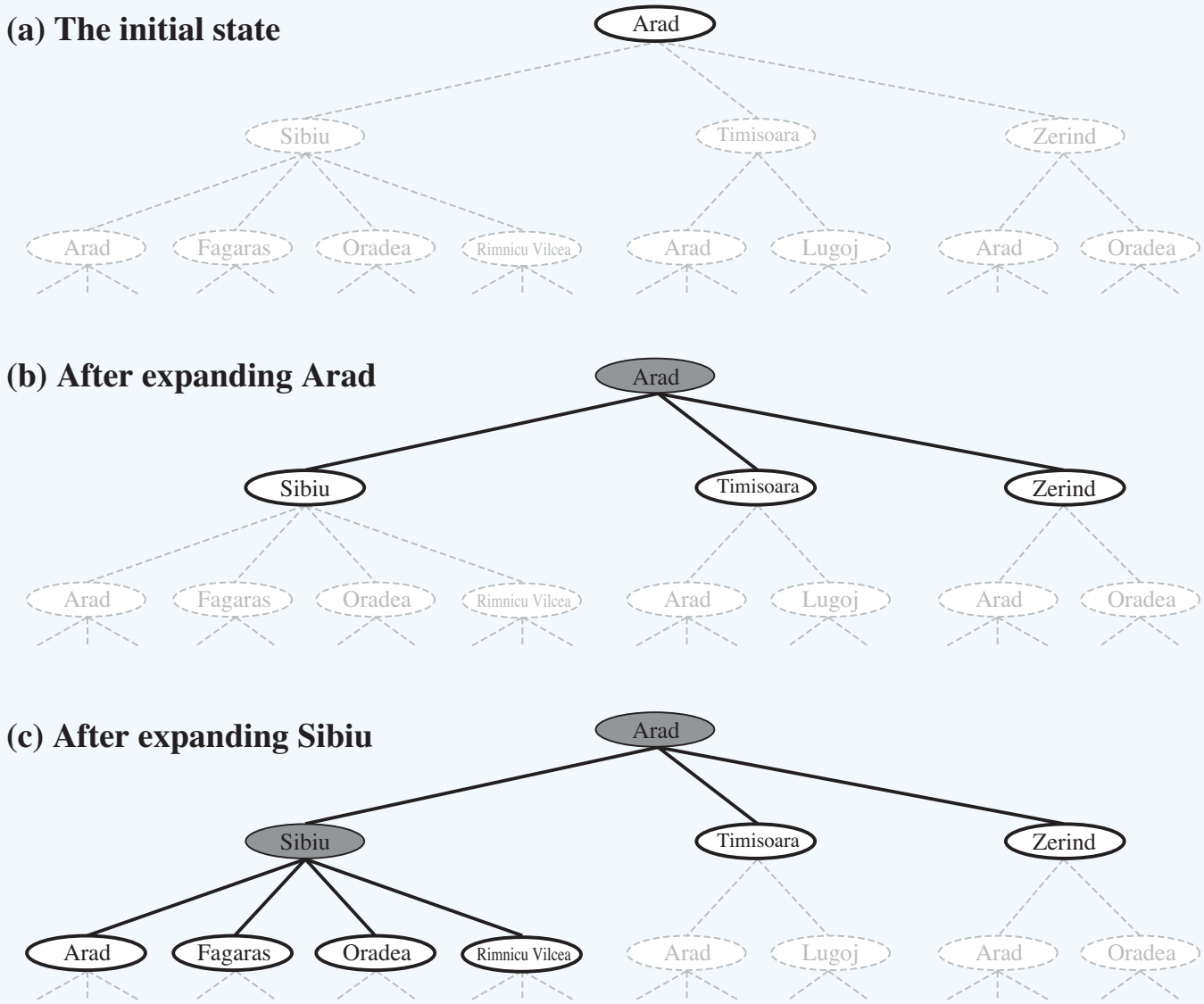
- We check/test it
- Not a goal state

We expand it

- `In(Arad)`
- `In(Sibiu)`
- `In(Timisoara)`
- `In(Zerind)`

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Example



Searching for solutions (cont.)

We can pick any of these options or go back and pick Timisoara or Zerind

- Each of these six nodes is a **leaf node**
- A node with no children in the tree

Frontier

Set of all leaves available for expansion, at any point

- Many authors call it the **open list**

The frontier of each tree consists of nodes with bold outlines

Searching for solutions (cont.)

The process of expanding nodes on the frontier continues until either

1. A solution is found
2. There are no more states to expand

Searching for solutions (cont.)

The general **TREE-SEARCH** algorithm is shown informally

function TREE-SEARCH(*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state of *problem*
  loop do
    if the frontier is empty **then return** failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state **then return** the corresponding solution
    expand the chosen node, adding the resulting nodes to the frontier

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Searching for solutions (cont.)

Search algorithms all share this basic structure

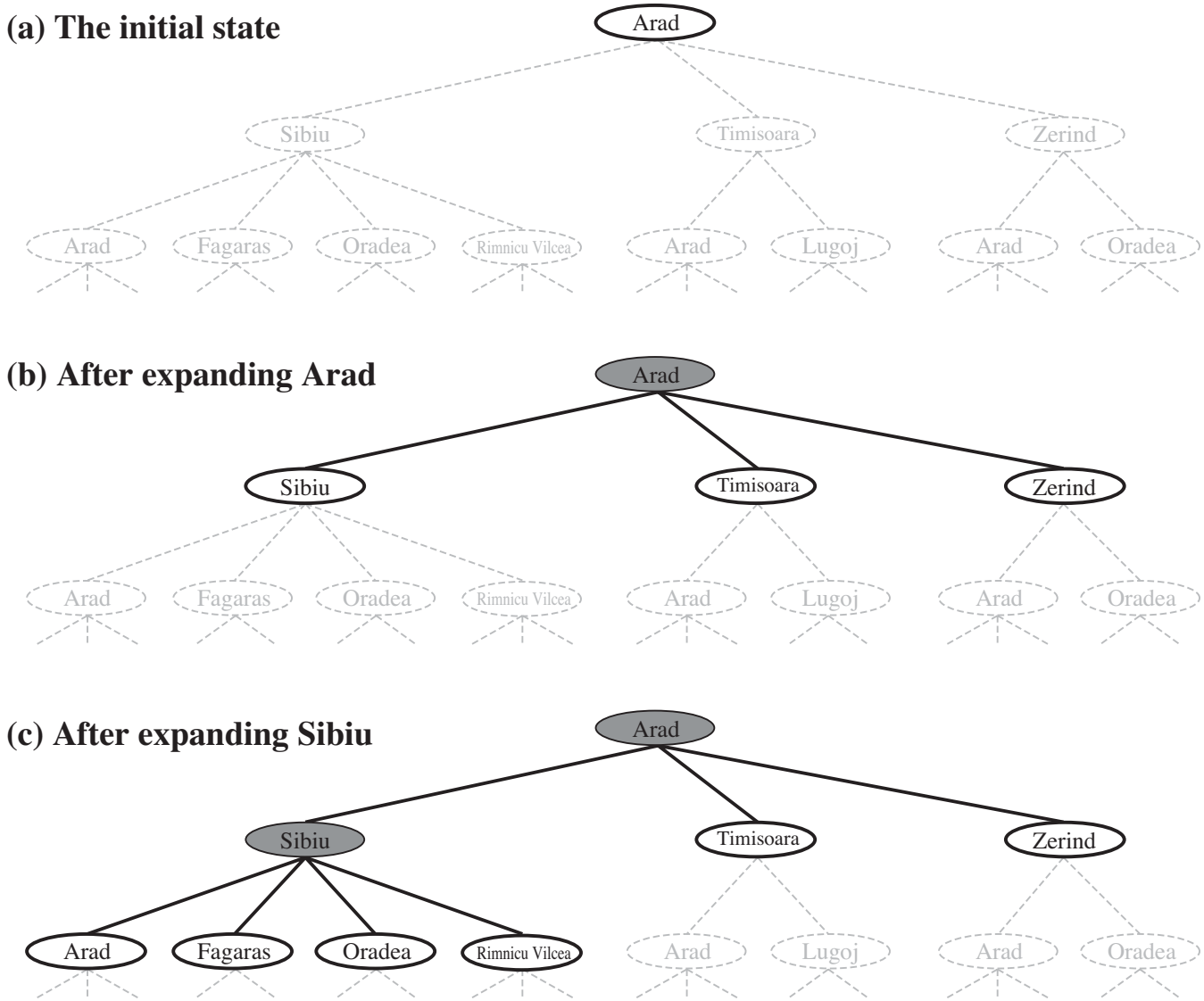
What varies mostly is how they choose which state to expand next

- ('choose a leaf node and remove it from the frontier')

$\rightsquigarrow$ The **search strategy**

Searching for solutions (cont.)

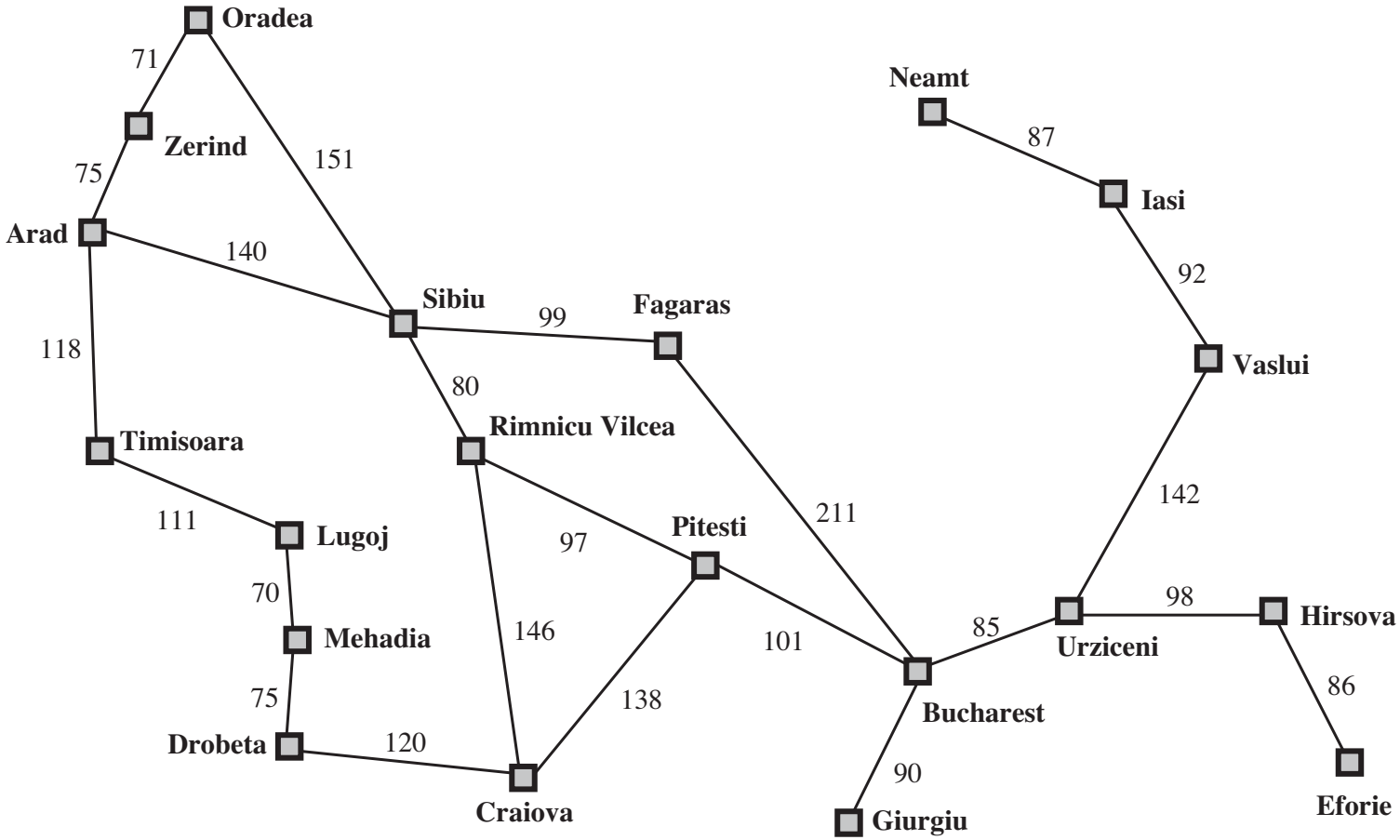
Search tree includes the path from Arad to Sibiu and back to Arad



Searching for solutions (cont.)

In(Arad) is a **repeated state** in the search tree

- It was generated by a **loopy path**



Searching for solutions (cont.)

Considering loopy paths makes the complete search tree infinite

- No limit to how often one can traverse a loop

On a problem whose state space has only 20 states

Searching for solutions (cont.)

Loops can cause certain algorithms to fail

- Solvable problems can be made unsolvable

There is no need for loopy paths, clearly

- Step costs are non-negative and path costs are additive

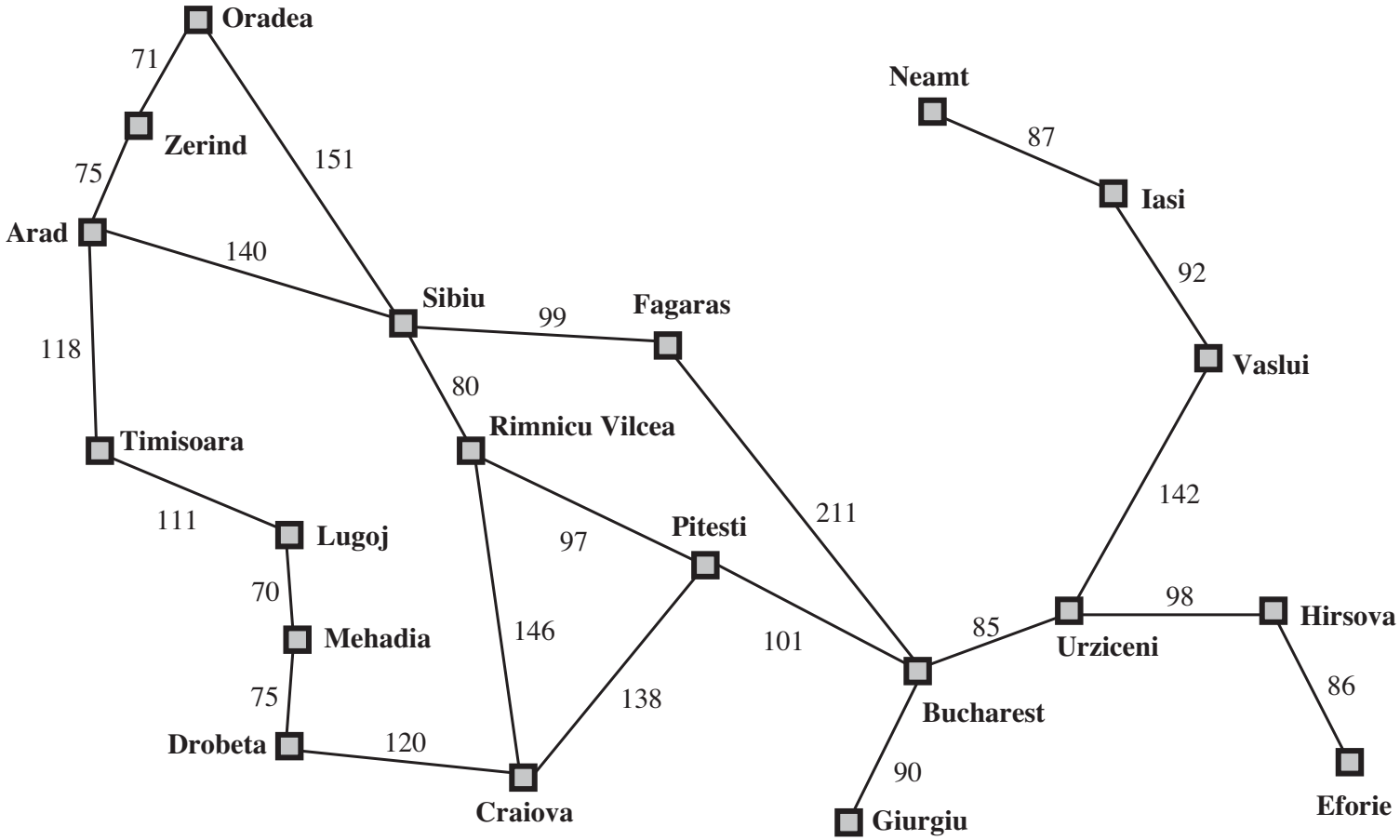
Loopy paths are never better than same paths with loops removed

Loops are special cases of the general concept of **redundant paths**

- (there is more than one way to get from one state to another)

Searching for solutions (cont.)

Paths Arad-Sibiu (140km) and Arad-Zerind-Oradea-Sibiu (297km)



Second path is redundant and a worse way to get to the same state

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Searching for solutions (cont.)

Remark

There's never any reason to keep more than one path to any given state

Consider a goal state that is reachable by extending one path

- That state is also reachable by extending the other path

Searching for solutions (cont.)

It may be possible to define the problem so as to eliminate redundant paths

- In other cases, redundant paths are unavoidable

This includes all problems where actions are reversible

- Route-finding problems, sliding-block puzzles, ...

Remark

Redundant paths can cause a tractable problem to turn intractable

- This is true even for algorithms that avoid infinite loops

Searching for solutions (cont.)

Algorithms that forget their history are doomed to repeat it

To avoid exploring redundant paths, remember where one has been

We augment the **TREE-SEARCH** algorithm with a data structure

- The **explored set** or **closed list**, it remembers every expanded node
- Newly generated nodes that match previously generated nodes
- Those in the explored set or the frontier, can be discarded
- They are not added to the frontier

Searching for solutions (cont.)

The new algorithm is called the **GRAPH-SEARCH**

function GRAPH-SEARCH(*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state of *problem*
  initialize the explored set to be empty
  loop do
    if the frontier is empty **then return** failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state **then return** the corresponding solution
    add the node to the explored set
    expand the chosen node, adding the resulting nodes to the frontier
      only if not in the frontier or explored set

Searching for solutions (cont.)

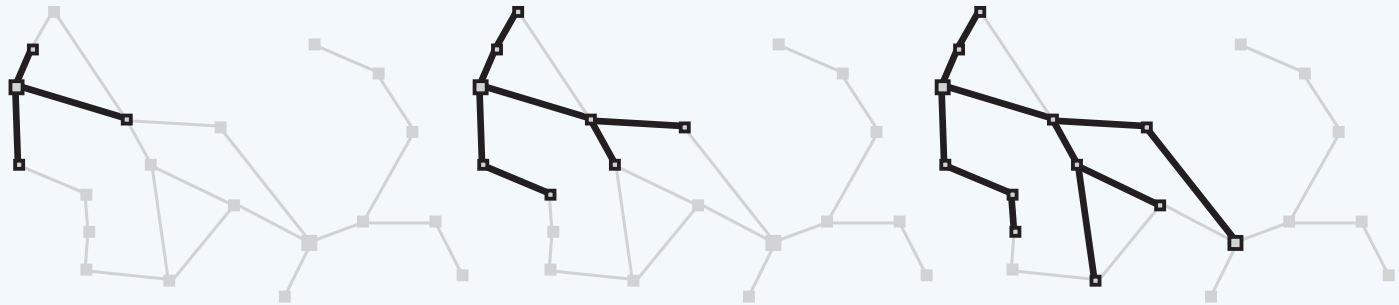
The **GRAPH-SEARCH** algorithm contains max one copy of each state

$\rightsquigarrow$ We can grow a tree on the state-space graph

Example

A sequence of search trees by a graph search on Romania

- At each stage, we have extended each path by one step



Northernmost city (Oradea) has become a dead end (3rd stage)

- Both of its successors are already explored via other paths

Searching for solutions (cont.)

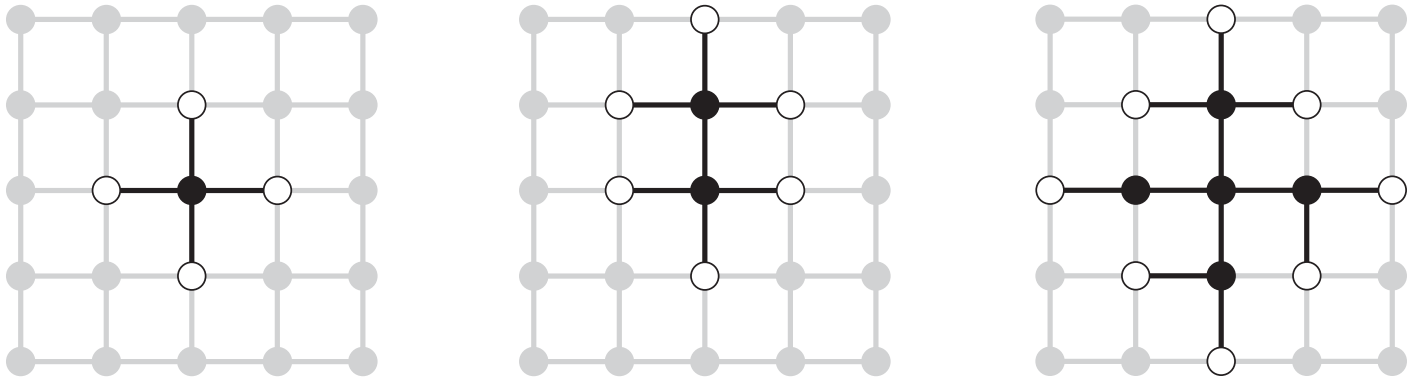
The frontier splits the state-space graph into explored/unexplored

- ⇝ For every path from the initial state to an unexplored state
- ⇝ One has to pass through a state in the frontier

Searching for solutions (cont.)

The frontier (white nodes) separates between the explored and unexplored

- Explored region of the state-space (black nodes)
- Unexplored region of the state space (gray nodes)



(a) (b) (c)

- In (a), just the root has been expanded
- In (b), one leaf node has been expanded
- In (c), remaining successors of root have been expanded (CW)

Solving by searching

UFC/DC
CK0031/CK0248
2017.2

Problem solving

Solving agents
Well-definedtness
Problem formulation

Solution search
Search algorithms
Performance metrics

Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search

Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search

Heuristics
Performance
Admissibility
Learning heuristics

Search algorithms

Searching for solutions

Search algorithms

Search algorithms require a data structure to keep track of the search tree

- The one that is being constructed

For each node n of the tree, a structure with four components

n.STATE

$\rightsquigarrow$ The state in the state space to which the node corresponds

n.PARENT

$\rightsquigarrow$ The node in the search tree that generated this node

n.ACTION

$\rightsquigarrow$ The action that was applied to the parent to generate the node

n.PATH-COST

$\rightsquigarrow$ The cost, $g(n)$, of the path from the initial state to the node

- (as indicated by the parent pointers)

Search algorithms (cont.)

We are given the components for a parent node

We must compute the necessary components for a child node

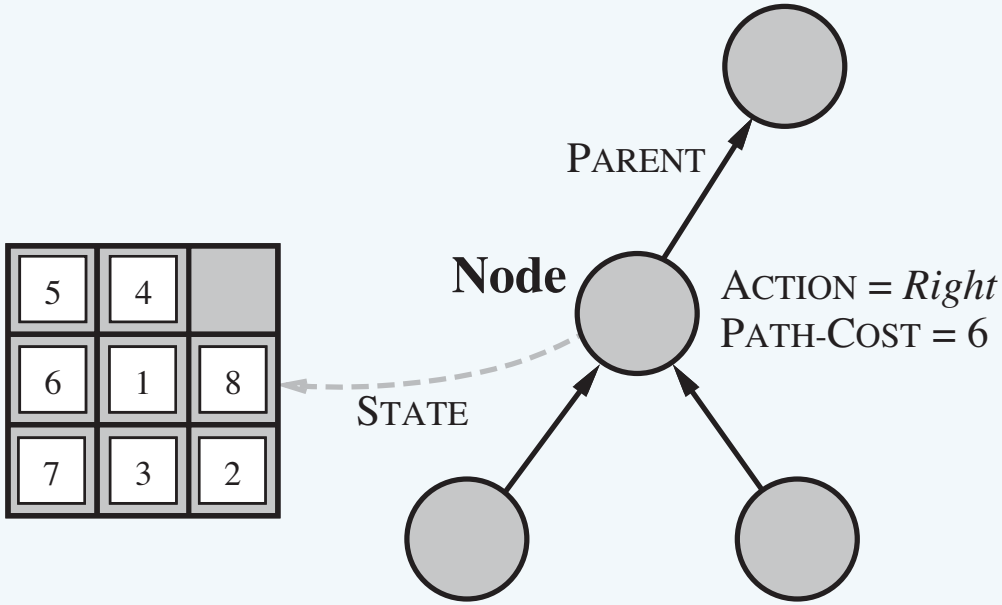
- We use function **CHILD-NODE**

It takes a parent node and an action, returns the resulting child

function CHILD-NODE(*problem*, *parent*, *action*) **returns** a node
  return a node with
    STATE = *problem*.RESULT(*parent*.STATE, *action*),
    PARENT = *parent*, ACTION = *action*,
    PATH-COST = *parent*.PATH-COST + *problem*.STEP-COST(*parent*.STATE, *action*)

Search algorithms (cont.)

Example



Nodes are the data structures from which a search tree is built

- Each has a parent, a state, and some book-keeping fields
- Arrows point from child to parent

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Search algorithms (cont.)

Remark

We were not very careful to distinguish between nodes and states

- It's important to make that distinction

- A node is a data structure (used to represent the search tree)
- A state corresponds to a configuration of the world

Nodes are on paths (defined by PARENT pointers), states are not

- Two different nodes can contain the same world state
- If that state is generated via two different search paths

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Search algorithms (cont.)

How to store the frontier?

In a way that search algos can easily choose next node to expand

- According to preferred strategy

The appropriate data structure for this is a **queue**

The operations on a queue

`EMPTY?`(`queue`)

⇝ It returns true only if there are no more elements in the queue

`POP`(`queue`)

⇝ It removes the first element of the queue and returns it

`INSERT`(`element`,`queue`)

⇝ It inserts an element and returns the resulting queue

Search algorithms (cont.)

Queues are characterised by the order in which they store inserted nodes

Three common variants

The **first-in, first-out** or **FIFO queue**

- It pops the oldest element of the queue

The **last-in, first-out** or **LIFO queue** or **stack**

- It pops the newest element of the queue

The **priority queue**

- It pops the element of the queue with highest priority
- Ranking according to some ordering function

Solving by searching

UFC/DC
CK0031/CK0248
2017.2

Problem solving

Solving agents
Well-definedtness
Problem formulation

Solution search
Search algorithms
Performance metrics

Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search

Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search

Heuristics
Performance
Admissibility
Learning heuristics

Measuring performance

Searching for solutions

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Measuring performance

We consider the criteria that might be used to choose search algorithms

We can evaluate an algorithm's performance in four ways

Completeness

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

Optimality

- Does the strategy find the optimal solution?

Time complexity

- How long does it take to find a solution?

Space complexity

- How much memory is needed to perform the search?

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Measuring performance (cont.)

Time and space complexity

They are considered with respect to some measure of problem difficulty

Remark

In TCS, the typical measure is the size of the state space graph

$$|V| + |E|$$

V is the set of vertices (nodes) and E is the set of edges (links)

This is appropriate when the graph is an explicit data structure that is input to the search program (for example, the map of Romania)

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Measuring performance (cont.)

Remark

In AI, the graph is often represented implicitly

- By initial state, actions, and transition model

Moreover, it is often infinite

Definition

Complexity is expressed in terms of three quantities

- b, ***branching factor*** *or maximum number of successors of any node*
- d, ***depth*** *of the shallowest goal node*
- m, ***maximum length*** *of any path in the state space*

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Measuring performance (cont.)

Time is often measured as number of nodes generated during search

Space is often measured as maximum number of nodes stored in memory

We describe time and space complexity for search on a tree

- For a graph, it depends on how 'redundant' paths are

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Measuring performance (cont.)

To assess the effectiveness of a search algorithm, we can consider two costs

Search cost

- It often depends on time complexity but can include memory usage

Total cost

- It combines the search cost and the path cost of the solution found

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Measuring performance (cont.)

Example

The problem of finding a route from Arad to Bucharest

- Search cost is the amount of time taken by the search
- Solution cost is the total path length in kilometres

To compute the total cost, we add milliseconds and kilometres

- No direct link between them
- Reasonable to convert kilometres into milliseconds (time is important)
- We can use an estimate from car's average speed

Solving by searching

UFC/DC
CK0031/CK0248
2017.2

Problem solving

Solving agents
Well-definedtness
Problem formulation

Solution search
Search algorithms
Performance metrics

Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search

Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search

Heuristics
Performance
Admissibility
Learning heuristics

Uninformed search
Solving by searching

Solving by searching

UFC/DC CK0031/CK0248 2017.2

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Uninformed search

We discuss search strategies known as **uninformed**/**blind search**

- The strategies have no additional info about states
- All the info is provided in the problem definition

They can generate successors and distinguish goal/non-goal states

- **Breadth-first search**
- **Uniform-cost search**
- **Depth-first search**
- **Depth-limited search**
- **Iterative deepening depth-first search**
- **Bidirectional search**

Search strategies are distinguished by the node expansion order

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Uninformed search

Remark

Informed/**heuristic search** strategies

They know whether a non-goal state is 'more promising' than others

Solving by searching

UFC/DC
CK0031/CK0248
2017.2

Problem solving

Solving agents
Well-definedtness
Problem formulation

Solution search
Search algorithms
Performance metrics

Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search

Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search

Heuristics
Performance
Admissibility
Learning heuristics

Breadth-first search
Uninformed search

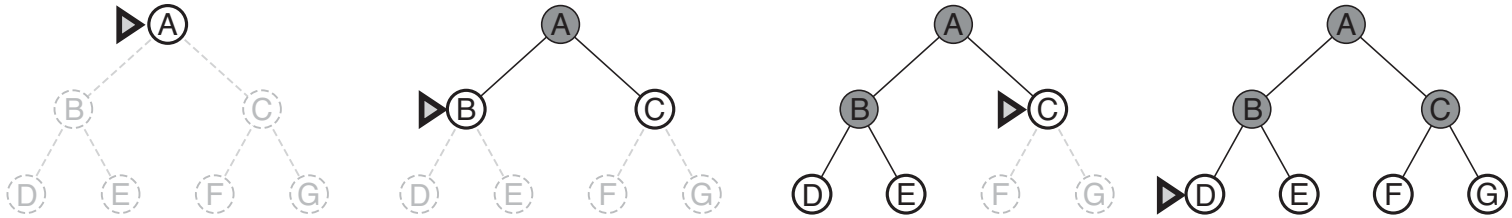
Breadth-first search

Breadth-first (BF) search

1. Root node is expanded first
2. All successors of root node are then expanded
3. Then, their successors are expanded
4. ... and so on

In general, all the nodes are expanded at a given depth in the search tree

- Before any nodes at the next level are expanded



At each stage, the node to be expanded is indicated by a marker

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Breadth-first search (cont.)

Breadth-first search is an instance of the general graph-search algo

- The shallowest unexpanded node is chosen for expansion

This is achieved by using a FIFO queue for the frontier

- New nodes (always deeper than their parents) go to queue's back
- Old nodes (shallower than new ones) get expanded first

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Breadth-first search (cont.)

Remark

There is one slight tweak on the general graph-search algorithm

- ⇝ Goal test is applied to each node when it is generated
  - (Rather than when it is selected for expansion)

Breadth-first search (cont.)

The algo discards paths to states already in the frontier or explored set

- It follows the template for graph search

Any such path must be at least as deep as the one already found

- BF search always has the shallowest path to every frontier node

Breadth-first search (cont.)

function BREADTH-FIRST-SEARCH(*problem*) **returns** a solution, or failure

```
node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
frontier ← a FIFO queue with node as the only element
explored ← an empty set
loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier)   /* chooses the shallowest node in frontier */
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
        child ← CHILD-NODE(problem, node, action)
        if child.STATE is not in explored or frontier then
            if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
            frontier ← INSERT(child, frontier)
```

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Breadth-first search (cont.)

How does it rate according to the four criteria?

- It is complete: If the shallowest goal node is at some finite depth d, breadth-first search will find it after generating all shallower nodes

$\rightsquigarrow$ (branching factor b need be finite)

- As a goal node is generated, it is the shallowest goal node

$\rightsquigarrow$ (all shallower nodes must have been generated and failed goal test)

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Breadth-first search (cont.)

The shallowest goal node is not necessarily the optimal one

- Technically, breadth-first search is optimal if the path cost is a non-decreasing function of the depth of the node

Most common such scenario: All actions have the same cost

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Breadth-first search (cont.)

What about time complexity?

Imagine searching a uniform tree, every state has b successors

- The root of the search tree generates b nodes at level one
- Each of which generates b more nodes (a total of b^2 at level two)
- Each of these generates b more nodes (b^3 nodes at the third level)
- And so on ...

Now suppose that the solution is at depth d

- In the worst case, it is the last node generated at that level

The number of nodes generated is $b + b^2 + b^3 + \cdots + b^d = \mathcal{O}(b^d)$

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Breadth-first search (cont.)

Remark

If the algo were to apply the goal test to nodes when selected for expansion

- Rather than when generated

Whole layer at depth d would be expanded before the goal was detected

Time complexity would be $\mathcal{O}(b^{d+1})$

Breadth-first search (cont.)

What about space complexity?

For any kind of graph search that stores every expanded node in the explored set, the space complexity is always within a factor of b of the time complexity

For breadth-first graph search

- Every node generated remains in memory
- There will be $\mathcal{O}(b^{d-1})$ nodes in the explored set
- There will be $\mathcal{O}(b^d)$ nodes in the frontier

Space complexity is $\mathcal{O}(b^d)$, dominated by size of the frontier

Breadth-first search (cont.)

Remark

Switching to tree search would not save much space

In a state space with redundant paths, switching would be time expensive

Breadth-first search (cont.)

An exponential complexity bound such as $\mathcal{O}(b^d)$ is scary stuff

Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabyte
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabyte
14	10^{14}	3.5 years	99 petabytes
16	10^{16}	350 years	10 exabytes

Time and memory for a breadth-first search, branching factor $b = 10$

- The table assumes that 1 million nodes can be generated per second
- A node requires 1000 bytes of storage
- For various values of the solution depth d

Many search problems fit roughly within these assumptions

- (give or take a factor of 100, when run on a modern PC)

Breadth-first search (cont.)

The memory requirements are a bigger problem for breadth-first search than is the execution time

- I could wait 13 days for a 12-deep problem to get solved
- I don't have a petabyte of memory

Exponential complexity search problems cannot be solved by uninformed methods for any but the smallest instances

- I don't have 350 years either, for a 16-deep problem

Solving by searching

UFC/DC
CK0031/CK0248
2017.2

Problem solving

Solving agents
Well-definedtness
Problem formulation

Solution search
Search algorithms
Performance metrics

Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search

Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search

Heuristics
Performance
Admissibility
Learning heuristics

Uniform-cost search
Uninformed search

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Uniform-cost search

When all step costs are equal, breadth-first search is optimal

- It always expands the shallowest unexpanded node

We can find an algorithm that is optimal with any step-cost function

Uniform-cost search expands the node n with the lowest path cost $g(n)$

- Instead of expanding the shallowest node

By storing the frontier as a priority queue ordered by g

Uniform-cost search (cont.)

function UNIFORM-COST-SEARCH(*problem*) **returns** a solution, or failure

node ← a node with STATE = *problem*.INITIAL-STATE, PATH-COST = 0
frontier ← a priority queue ordered by PATH-COST, with *node* as the only element
explored ← an empty set
loop do
  if EMPTY?(*frontier*) **then return** failure
  node ← POP(*frontier*) /* chooses the lowest-cost node in *frontier* */
  if *problem*.GOAL-TEST(*node*.STATE) **then return** SOLUTION(*node*)
  add *node*.STATE to *explored*
  for each *action* **in** *problem*.ACTIONS(*node*.STATE) **do**
    child ← CHILD-NODE(*problem*, *node*, *action*)
    if *child*.STATE is not in *explored* or *frontier* **then**
      frontier ← INSERT(*child*, *frontier*)
    else if *child*.STATE is in *frontier* with higher PATH-COST **then**
      replace that *frontier* node with *child*

Uniform-cost search (cont.)

The algorithm is almost identical to general graph search

- Use of a **priority queue** and the addition of an **extra check**
- (in case a shorter path to a frontier state is discovered)

The data structure for the frontier needs to support efficient membership testing

- It should combine capabilities of a priority queue and a hash table

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Uniform-cost search (cont.)

There are two other significant differences from breadth-first search

- In addition to the ordering of the queue by path cost

The goal test is applied to a node when it is selected for expansion

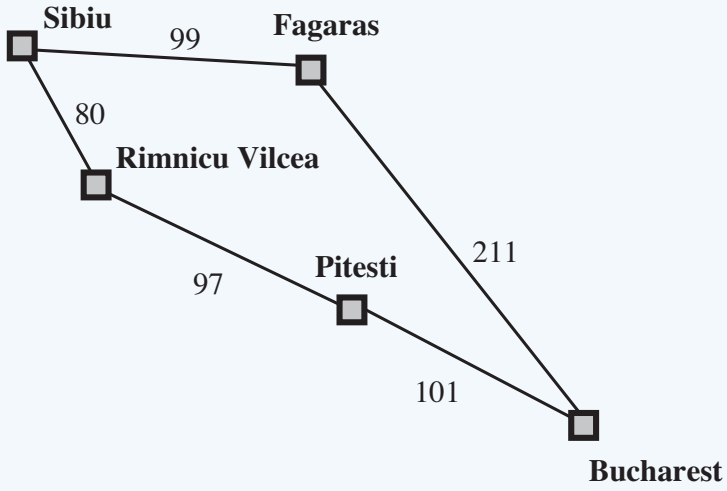
- Rather than when it is first generated

Because the first goal node that is generated may be on a suboptimal path

A test is added in case a better path is found to a current frontier node

Uniform-cost search (cont.)

Example



From Sibiu to Bucharest

The successors of Sibiu

- Rimnicu Vilcea and Fagaras
- Costs 80 and 99

1. The least-cost node, Rimnicu Vilcea, is expanded next

⇝ Adding Pitesti with cost $80 + 97 = 177$

2. The least-cost node is Fagaras, it is expanded

⇝ Adding Bucharest with cost $99 + 211 = 310$

A goal node has been generated, uniform-cost search keeps going

- Choose Pitesti for expansion

⇝ Adding a second path to Bucharest with cost $80 + 97 + 101 = 278$

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Uniform-cost search (cont.)

Example

The algo checks to see if this new path is better than the old one

- It is (278 v 310), so the old one is discarded

Bucharest, now with a g-cost of 278, is selected for expansion

- The solution is returned

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Uniform-cost search (cont.)

It is easy to see that uniform-cost search is optimal, in general

Whenever uniform-cost search selects a node n for expansion

$\rightsquigarrow$ the optimal path to that node has been found

- Otherwise, there would have to be another frontier node n' on the optimal path from the start node to n
- By definition, n would have lower g-cost than n
- It would have been selected first

Non-negative step costs, paths never get shorter as nodes are added

Nodes are expanded in order of their optimal path cost

$\rightsquigarrow$ First goal node selected for expansion must be the optimal solution

Uniform-cost search (cont.)

Uniform-cost search does not care about the number of steps on a path has

$\rightsquigarrow$ Only their total cost matters

It will get stuck in an infinite loop if there is a path with an infinite sequence of zero-cost actions (like `NoOp`'s[4])

- Completeness is guaranteed provided the cost of every step exceeds some small constant ε

[4] 'No Operation', as in an instruction that does nothing

Uniform-cost search (cont.)

Uniform-cost search is guided by path costs rather than depths

- complexity is not easily characterised in terms of b and d

Let C^* be the cost of the optimal solution

- Assume that every action costs at least ε
- The algorithm's worst-case time and space complexity

$$\mathcal{O}(b^{1+\lfloor C^*/\varepsilon \rfloor})$$

  It can be much greater than b^d

This is because uniform-cost search can explore large trees of small steps before exploring paths with large and perhaps useful steps

- When all step costs are equal, $b^{1+\lfloor C^*/\varepsilon \rfloor}$ is just b^{d+1}

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Uniform-cost search (cont.)

Remark

When all step costs are equal, uniform-cost search is similar to breadth-first search

- The latter stops as soon as it generates a goal
- Uniform-cost search examines all the nodes at the goal's depth
- (to see if one has a lower cost)

Thus, uniform-cost search does strictly more work

- It expands nodes at depth d unnecessarily

Solving by searching

UFC/DC
CK0031/CK0248
2017.2

Problem solving

Solving agents
Well-definedtness
Problem formulation

Solution search
Search algorithms
Performance metrics

Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search

Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search

Heuristics
Performance
Admissibility
Learning heuristics

Depth-first search
Uninformed search

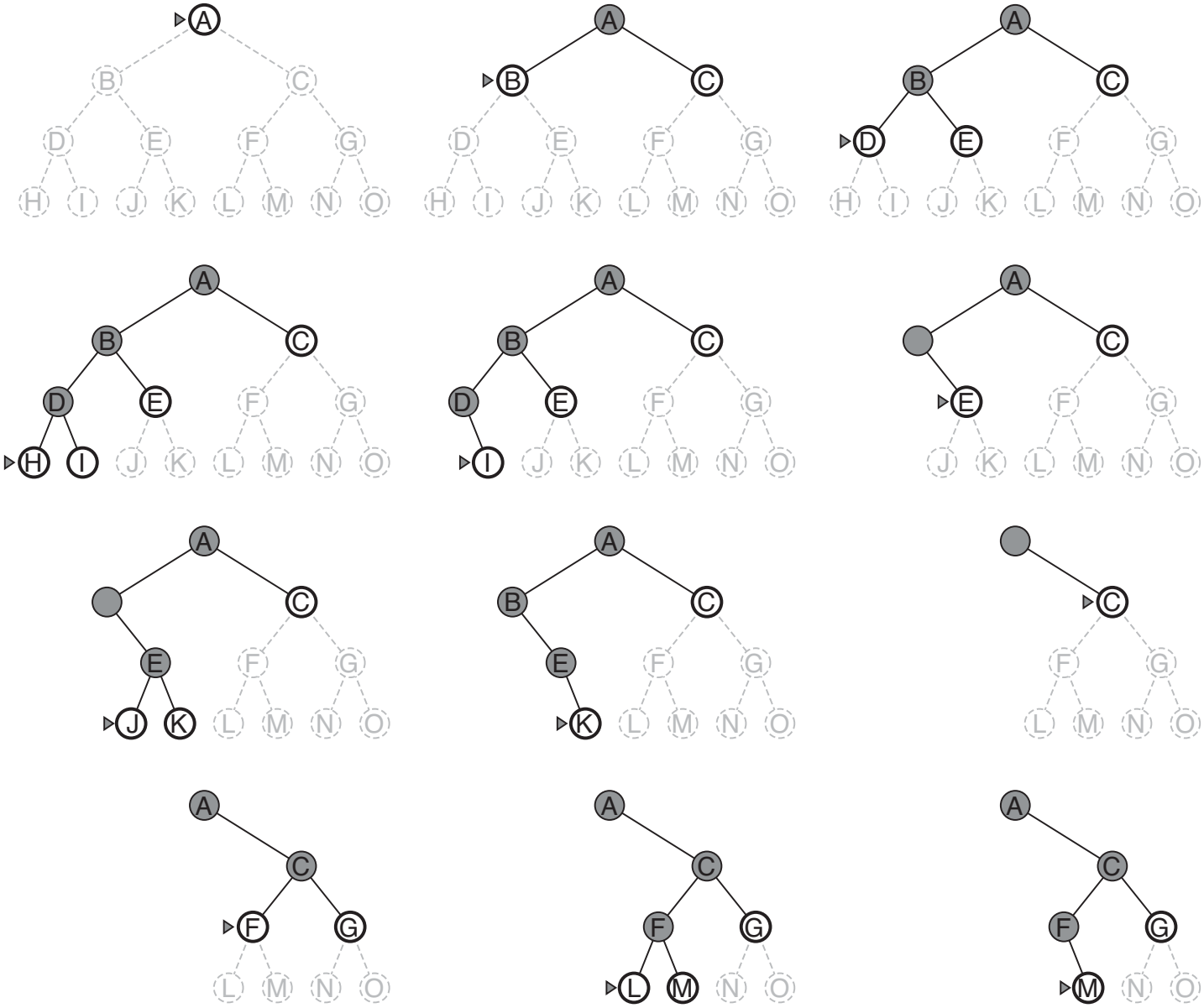
Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Depth-first search

Depth-first search always expands the deepest node in the current frontier of the search tree

- The search proceeds immediately to the deepest level of the search tree
- (where the nodes have no successors)
- As those nodes are expanded, they are dropped from the frontier
- Then the search 'backs up' to the next deepest node that still has unexplored successors

Depth-first search (cont.)



Depth-first search (cont.)

Depth-first search algorithm is an instance of graph-search algos

- Breadth-first-search uses a FIFO queue
- Depth-first search uses a LIFO queue

Thus, the most recently generated node is chosen for expansion

This must be the deepest unexpanded node because it is one deeper than its parent (which was the deepest unexpanded node when it was selected)

Depth-first search (cont.)

As an alternative to the **GRAPH-SEARCH**-style implementation

It is common to implement depth-first search with a recursive function

- It calls itself on each of its children

function DEPTH-LIMITED-SEARCH(*problem*, *limit*) **returns** a solution, or failure/cutoff
  return RECURSIVE-DLS(MAKE-NODE(*problem*.INITIAL-STATE), *problem*, *limit*)

function RECURSIVE-DLS(*node*, *problem*, *limit*) **returns** a solution, or failure/cutoff
  if *problem*.GOAL-TEST(*node*.STATE) **then return** SOLUTION(*node*)
  else if *limit* = 0 **then return** *cutoff*
  else
    cutoff_occurred? ← false
    for each *action* **in** *problem*.ACTIONS(*node*.STATE) **do**
      child ← CHILD-NODE(*problem*, *node*, *action*)
      result ← RECURSIVE-DLS(*child*, *problem*, *limit* − 1)
      if *result* = *cutoff* **then** *cutoff_occurred*? ← true
      else if *result* ≠ *failure* **then return** *result*
    if *cutoff_occurred*? **then return** *cutoff* **else return** *failure*

A recursive depth-first algorithm incorporating a depth limit

Depth-first search (cont.)

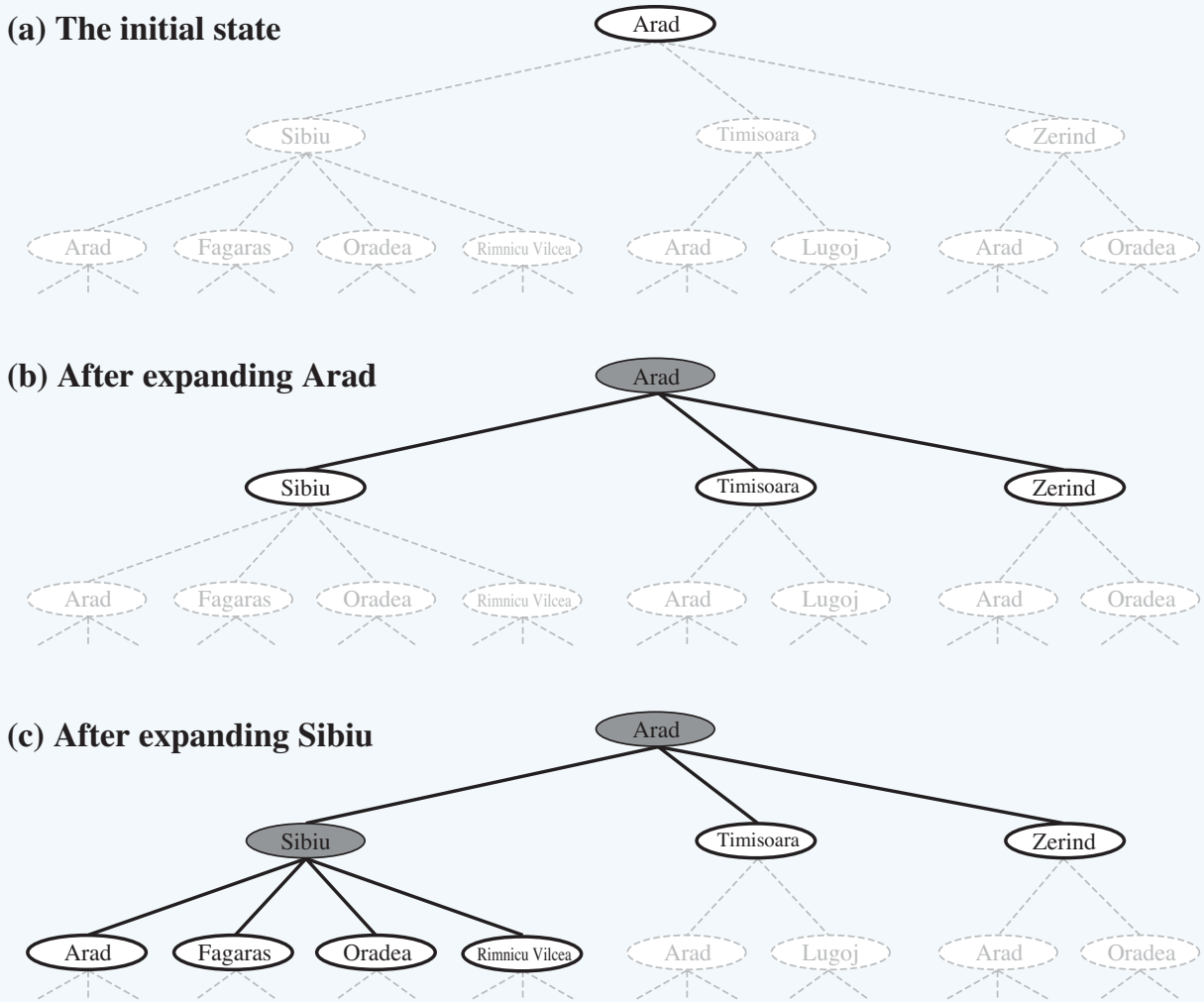
The properties of depth-first search depend on whether the graph-search or tree-search version is used

- The graph-search version, which avoids repeated states and redundant paths, is complete in finite state spaces, because it will eventually expand every node
- The tree-search version, on the other hand, is not complete

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Example

The tree-search version of algo will get stuck in Arad-Sibiu loop



Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Depth-first search (cont.)

Depth-first tree search can be modified at no extra memory cost

- Check new states against those on path from root to current node

This avoids infinite loops in finite state spaces

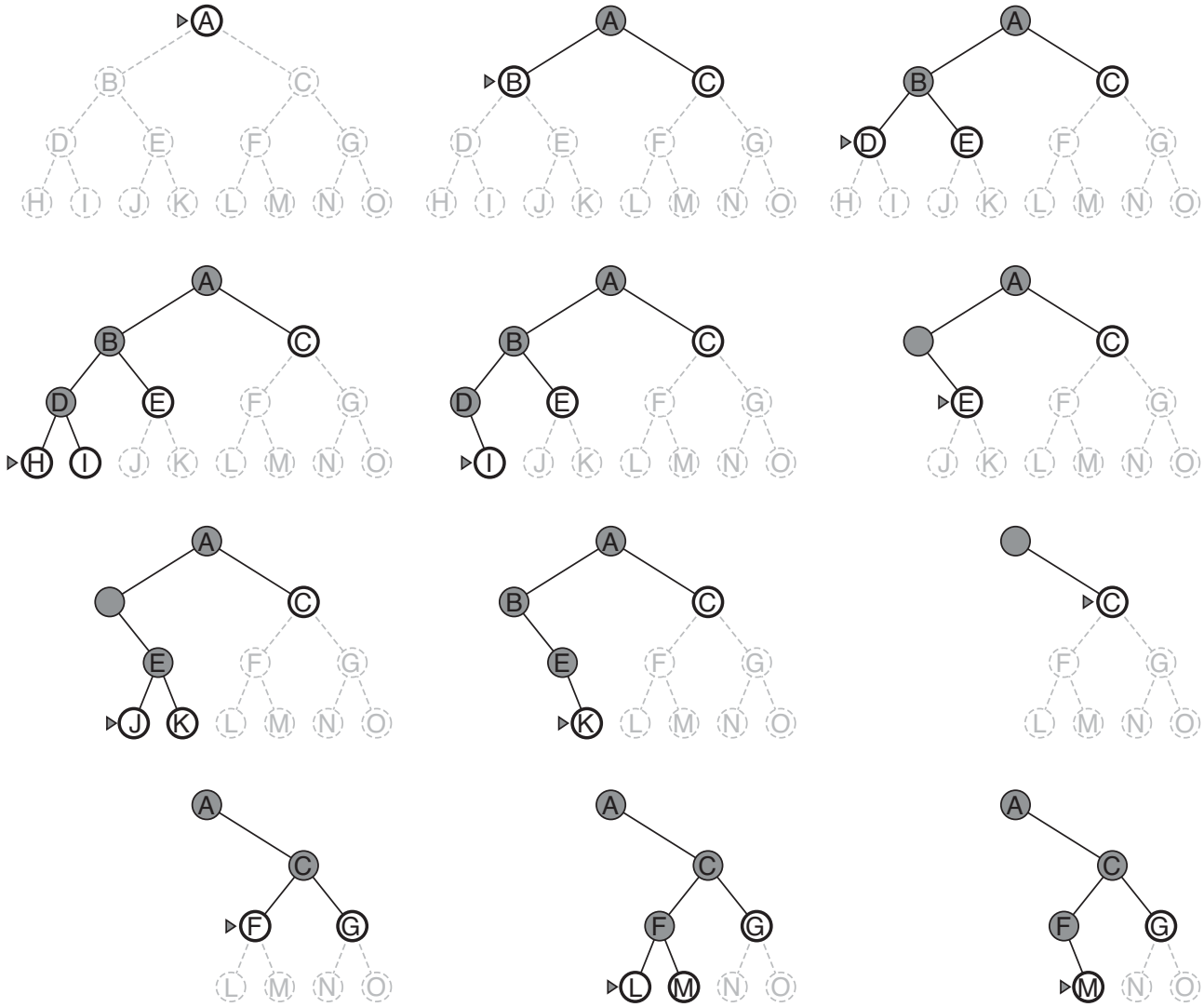
- It does not avoid proliferation of redundant paths

In infinite state spaces, both versions may fail

- If an infinite non-goal path is encountered

For similar reasons, both versions are non-optimal

Depth-first search: Entire left subtree, even if C is a goal node



If node J were also a goal node, then depth-first search would return it as a solution instead of C (clearly, a better solution)

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Depth-first search (cont.)

The time complexity of depth-first graph search is bounded

- The size of the state space (which may be infinite)

Depth-first tree search may generate all $\mathcal{O}(b^m)$ nodes in the search tree

- m is the maximum depth of any node
- (it can be much greater than the size of the state space)

Remark

Note that m can be much larger than d (the depth of the shallowest solution)

- It is infinite if the tree is unbounded

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Depth-first search (cont.)

So far, depth-first search seems better than breadth-first search

- So why do we include it?
- The reason is space complexity

Depth-first tree search needs to store a single path, from root to leaf node

- plus remaining unexpanded sibling nodes for each node on the path
- (for a graph search, there is no advantage)

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Depth-first search (cont.)

Once a node has been expanded, it can be removed from memory

- As soon as all of its descendants have been fully explored

Consider a state space with branching factor b and maximum depth m

- Depth-first search requires storage of only $\mathcal{O}(b^m)$ nodes

Example

Assume that nodes at the same depth as the goal node have no successor

Depth-first search requires 156Kbytes instead of 10Exabytes at depth $d = 16$

$\rightsquigarrow$ 7 trillion times less space

Depth-first search (cont.)

Depth-first tree search is the basic workhorse of many areas of AI

- We focus on the tree-search version of depth-first search

Backtracking: A variant of depth-first search uses less memory

- Only one successor is generated at a time rather than all successors
- Each partially expanded node remembers which successor to generate
- Only $\mathcal{O}(m)$ memory is needed rather than $\mathcal{O}(b^m)$

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Depth-first search (cont.)

Backtracking facilitates another memory- and time-saving trick

- The idea of generating a successor by modifying the current state description directly, rather than copying it first

Memory requirements: One state description and $\mathcal{O}(m)$ actions

- We must be able to undo each modification when we go back to generate the next successor

Example

For problems with large state descriptions (robotic assembly) these techniques are critical to success

Depth-limited search

Uninformed search

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Depth-limited search

The failure of depth-first search in infinite state spaces can be alleviated

- Supply depth-first search with a depth limit
- Nodes at depth l are treated as if they have no successors
- This approach is called **depth-limited search**
- The depth limit solves the infinite-path problem

This introduces an additional source of incompleteness if we choose $l < d$

- The shallowest goal is beyond the depth limit
- (likely when d is unknown)

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Depth-limited search (cont.)

Depth-limited search will also be non-optimal if we choose $l > d$

- Its time complexity is $\mathcal{O}(b^l)$
- Its space complexity is $\mathcal{O}(bl)$

Depth-first search can be viewed as a special case of depth-limited search

- with $l = \infty$

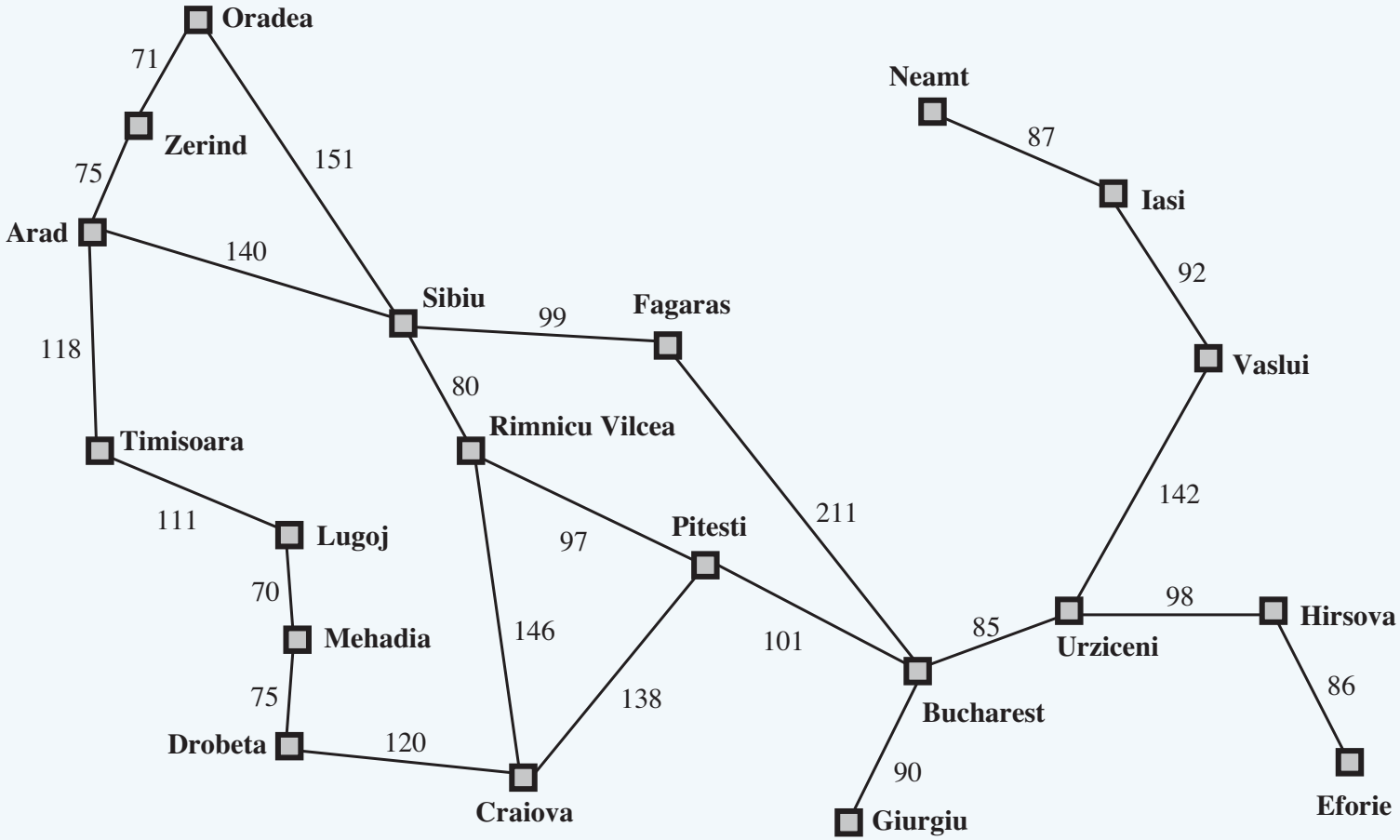
Sometimes, depth limits can be based on knowledge of the problem

Example

On 20 cities, we know that if there is a solution

It must be of length 19 at the longest

- $l = 19$ is a possible choice



In fact any city can be reached from any other city in max 9 hops

Depth-limited search (cont.)

Definition

*The **diameter of the state space**, gives us a better depth limit*

It leads to a more efficient depth-limited search

Remark

Usually, we do not know a good depth limit, until we solved the problem

Depth-limited search (cont.)

Depth-limited search can be implemented as modification to general tree or graph-search

- or as a recursive algorithm

function DEPTH-LIMITED-SEARCH(*problem*, *limit*) **returns** a solution, or failure/cutoff
  return RECURSIVE-DLS(MAKE-NODE(*problem*.INITIAL-STATE), *problem*, *limit*)

function RECURSIVE-DLS(*node*, *problem*, *limit*) **returns** a solution, or failure/cutoff
  if *problem*.GOAL-TEST(*node*.STATE) **then return** SOLUTION(*node*)
  else if *limit* = 0 **then return** *cutoff*
  else
    cutoff_occurred? ← false
    for each *action* **in** *problem*.ACTIONS(*node*.STATE) **do**
      child ← CHILD-NODE(*problem*, *node*, *action*)
      result ← RECURSIVE-DLS(*child*, *problem*, *limit* − 1)
      if *result* = *cutoff* **then** *cutoff_occurred*? ← true
      else if *result* ≠ *failure* **then return** *result*
    if *cutoff_occurred*? **then return** *cutoff* **else return** *failure*

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Depth-limited search (cont.)

Remark

Depth-limited search can terminate with two kinds of failure:

- The standard failure value indicates no solution
- The cutoff value indicates no solution, within the depth limit

Problem solving

Solving agents

Well-definedtness

Problem formulation

Solution search

Search algorithms

Performance metrics

Uninformed search

Breadth-first search

Uniform-cost search

Depth-first search

Depth-limited search

Iterative deepening depth-first search

Bidirectional search

Informed searches

Greedy best-first

A^* search

Memory-bounds

Learning to search

Heuristics

Performance

Admissibility

Learning heuristics

Iterative deepening depth-first search

Uninformed search

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Iterative deepening depth-first search

Iterative deepening search (or iterative deepening depth-first search)

A general strategy, often used in combination with depth-first tree search

- It finds the best depth limit

It does this by gradually increasing the limit

- First 0, then 1, then 2, and so on until a goal is found

This occurs when depth limit reaches d, the depth of shallowest goal node

function ITERATIVE-DEEPENING-SEARCH(*problem*) **returns** a solution, or failure
  for *depth* = 0 **to** ∞ **do**
    result $\leftarrow$ DEPTH-LIMITED-SEARCH(*problem*, *depth*)
    if *result* $\neq$ cutoff **then return** *result*

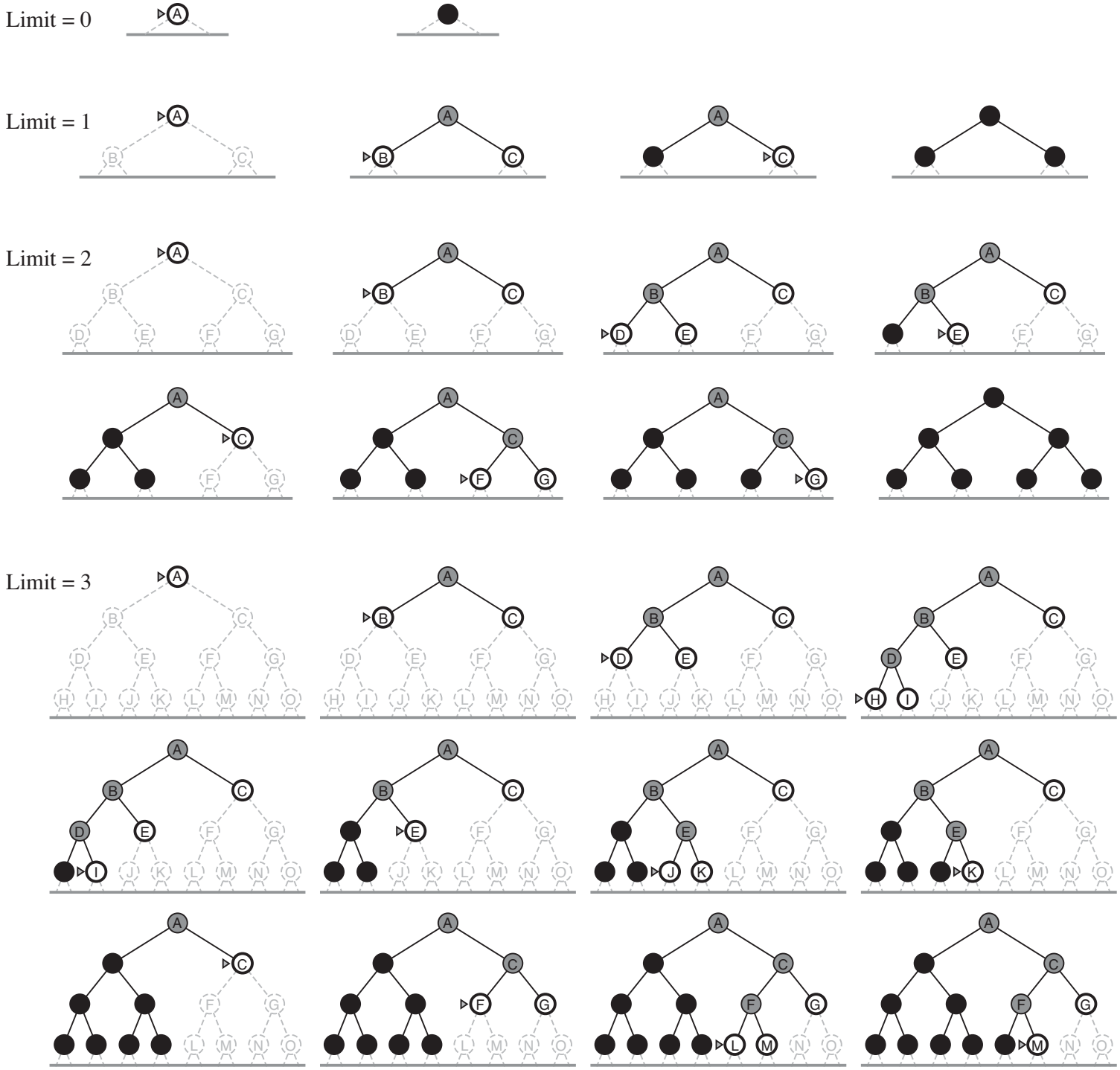
Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Iterative deepening depth-first (cont.)

Four iterations of **ITERATIVE-DEEPENING-SEARCH**

- On a binary search tree

The solution is found on the fourth iteration


Limit = 0
Limit = 1
Limit = 2
Limit = 3

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Iterative deepening depth-first (cont.)

Iterative deepening joins benefits from depth-first and breadth-first search

- Like depth-first search, memory requirements are modest

$$\mathcal{O}(bd)$$

- Like breadth-first search, complete when the branching factor is finite
- Like breadth-first search, optimal when the path cost is a non-decreasing function of the depth of the node

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Iterative deepening depth-first (cont.)

Iterative deepening search may seem wasteful

- States are generated multiple times
- It turns out this is not too costly

Consider a search tree with (nearly) the same branching factor at each level

- Most of the nodes are in the bottom level
- It does not matter much that upper levels are generated multiple times

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Iterative deepening depth-first (cont.)

In an iterative deepening search

- Nodes on bottom level (depth d) are generated once
- Nodes on next-to-bottom level are generated twice
- And so on, up to the children of the root, generated d times

The total number of nodes generated in the worst case is

$$N(\text{IDS}) = (d)b + (d-1)b^2 + \cdots + (1)b^d$$

It is a time complexity of $\mathcal{O}(b^d)$ (breadth-first, asymptotically)

Example

Some extra cost, for generating the upper levels multiple times

$$N(\text{IDS}) = 50 + 400 + 3000 + 20000 + 100000 = 123500$$
$$N(\text{BFS}) = 10 + 100 + 1000 + 10000 + 100000 = 111110$$

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Iterative deepening depth-first (cont.)

Remark

If repeating the repetition is a concern

Hybrid approaches can run breadth-first search until almost all available memory is used, then run iterative deepening from all nodes in the frontier

Remark

When search space is large and the solution depth is unknown

- ⇝ Iterative deepening is the preferred uninformed search
  - (In general)

Solving by searching

UFC/DC
CK0031/CK0248
2017.2

Problem solving

Solving agents

Well-definedtness

Problem formulation

Solution search

Search algorithms

Performance metrics

Uninformed search

Breadth-first search

Uniform-cost search

Depth-first search

Depth-limited search

Iterative deepening depth-first search

Bidirectional search

Informed searches

Greedy best-first

A^* search

Memory-bounds

Learning to search

Heuristics

Performance

Admissibility

Learning heuristics

Bidirectional search

Uninformed search

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Bidirectional search

The idea behind is to run two parallel searches

- one forward from the initial state
- the other backward from the goal

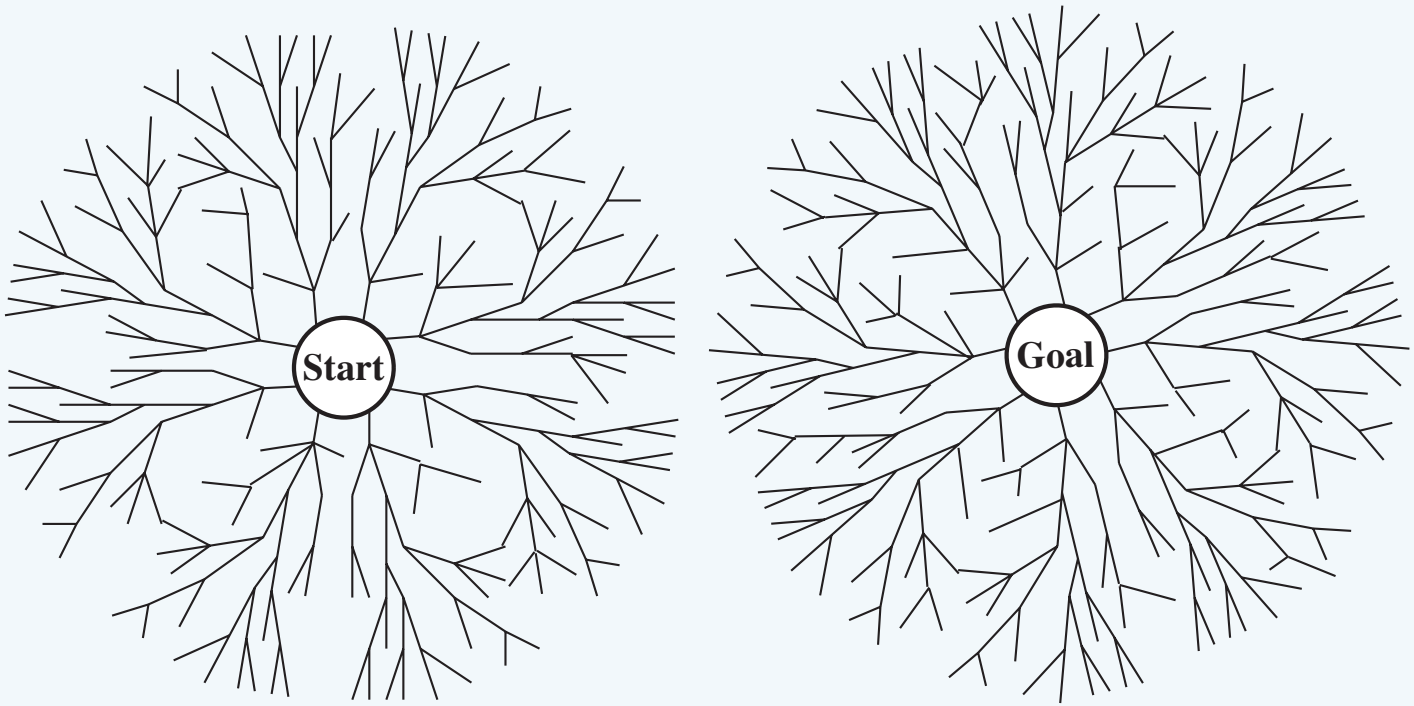
hoping that the two searches meet in the middle

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Bidirectional search (cont.)

Example

The motivation is that $b^{d/2} + b^{d/2}$ is much less than b^d



The area of the two small circles is less than the area of a big circle centred on the start and reaching to the goal

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Bidirectional search (cont.)

Bidirectional search is implemented by replacing the goal test with a check

- See whether the frontiers of the two searches intersect
- If they do, a solution has been found

The first such solution found may not be optimal

- Even if the two searches are both breadth first
- Some additional search is required
- (to make sure there is no other short-cut across the gap)

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Bidirectional search (cont.)

The check can be done when each node is generated or selected for expansion

- With a hash table, will take constant time

Example

Consider a problem with solution depth $d = 6$

Assume each direction runs BFS one node at a time

Then, in the worst case the two searches meet when they have generated all of the nodes at depth 3

For $b = 10$, this means a total of 2220 node generations

- For a stadard breath-first search, 1111110

Time complexity of bidirectional search with breadth-first searches in both directions

$$\mathcal{O}(b^{d/2})$$

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Bidirectional search (cont.)

The space complexity is also $\mathcal{O}(b^{d/2})$

This can be reduced by roughly half

- If one of the two searches is done by iterative deepening
- But at least one of the frontiers must be kept in memory
- (to do intersection check)

Remark

Space requirement is the weakness of bidirectional search

Bidirectional search (cont.)

The reduction in time complexity makes bidirectional search attractive

How do we search backward?

Let the **predecessors** of a state x be all those states that have x as a successor

Bidirectional search requires a method for computing predecessors

When all the actions in the state space are reversible, then the predecessors of x are just its successors

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Bidirectional search (cont.)

What we mean by 'the goal' in searching 'backward from goal?'

Example

Consider the 8-puzzle and finding a route in Romania

There is one goal state, so backward search is like forward search

- With several explicitly listed goal states (say, the two dirt-free goal states), then we can construct a new dummy goal state whose immediate predecessors are all the actual goal states
- But if the goal is an abstract description, such as the goal that 'no queen attacks another queen' in the n-queens problem, then bidirectional search is difficult to use

Comparison
Uninformed search

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Uninformed searches, comparison

We compare tree-search strategies using four evaluation criteria:

- Depth-first search is complete for finite state spaces
- Space and time complexities are bounded by the size of the state space

Uninformed searches, comparison (cont.)

- b is the branching factor;
- d is the depth of the shallowest solution;
- m is the maximum depth of the search tree;
- l is the depth limit

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

Superscript caveats:

- ^{a} complete if b is finite;
- ^{b} complete if step costs $\geq \varepsilon$, for $\varepsilon > 0$;
- ^{c} optimal if step costs are all identical;
- ^{d} if both directions use breadth-first

Solving by searching

UFC/DC
CK0031/CK0248
2017.2

Problem solving

Solving agents

Well-definedtness

Problem formulation

Solution search

Search algorithms

Performance metrics

Uninformed search

Breadth-first search

Uniform-cost search

Depth-first search

Depth-limited search

Iterative deepening depth-first search

Bidirectional search

Informed searches

Greedy best-first

A^* search

Memory-bounds

Learning to search

Heuristics

Performance

Admissibility

Learning heuristics

Informed searches
Solving by searching

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Informed searches

An **informed search** strategy uses a problem-specific knowledge

$\rightsquigarrow$ Beyond the definition of the problem itself

It can find solutions more efficiently than an uninformed strategy

The general approach we consider is **best-first search**

- An instance of general **TREE-** or **GRAPH-SEARCH** algos

A node is selected for expansion based on an **evaluation function**, $f(n)$

The evaluation function is construed as a cost estimate

- The node with the lowest evaluation is expanded first

Informed searches (cont.)

Best-first graph search is identical to uniform-cost search

function UNIFORM-COST-SEARCH(*problem*) **returns** a solution, or failure

node ← a node with STATE = *problem*.INITIAL-STATE, PATH-COST = 0
frontier ← a priority queue ordered by PATH-COST, with *node* as the only element
explored ← an empty set
loop do
  if EMPTY?(*frontier*) **then return** failure
  node ← POP(*frontier*) /* chooses the lowest-cost node in *frontier* */
  if *problem*.GOAL-TEST(*node*.STATE) **then return** SOLUTION(*node*)
  add *node*.STATE to *explored*
  for each *action* **in** *problem*.ACTIONS(*node*.STATE) **do**
    child ← CHILD-NODE(*problem*, *node*, *action*)
    if *child*.STATE is not in *explored* or *frontier* **then**
      frontier ← INSERT(*child*, *frontier*)
    else if *child*.STATE is in *frontier* with higher PATH-COST **then**
      replace that *frontier* node with *child*

Except for the use of f instead of g to order the priority queue

- The choice of f determines the search strategy

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Informed searches (cont.)

Best-first tree search includes depth-first search as a special case

Exercise

Prove each of the following statements, or give a counterexample

- Breadth-first search is a special case of uniform-cost search
- Depth-first search is a special case of best-first tree search
- Uniform-cost search is a special case of A^* search

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Informed searches (cont.)

Most best-first algos use as a component of f a **heuristic function** $h(n)$

- The estimated cost of cheapest path, from state at node n to goal state

Note that $h(n)$ takes a node as input

- Unlike $g(n)$, it depends only on the state at that node

Example

In Romania

An estimate of cost of the cheapest path from Arad to Bucharest

- The straight-line distance

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Informed searches (cont.)

Heuristic functions are the most common way to impart extra knowledge

- We shall study heuristics in more depth

Assume heuristics to be arbitrary, nonnegative, problem-specific functions

We require only one single constraint

- If n is a goal node, then $h(n) = 0$

Solving by searching

UFC/DC CK0031/CK0248 2017.2

Problem solving

Solving agents

Well-definedtness

Problem formulation

Solution search

Search algorithms

Performance metrics

Uninformed search

Breadth-first search

Uniform-cost search

Depth-first search

Depth-limited search

Iterative deepening depth-first search

Bidirectional search

Informed searches

Greedy best-first

A^* search

Memory-bounds

Learning to search

Heuristics

Performance

Admissibility

Learning heuristics

Greedy best-first search

Informed search

Greedy best-first search

Greedy best-first search

Try to expand the node that is closest to goal

- Likely to lead to a solution quickly

Greedy best-first search evaluates nodes by using the heuristic function

$$f(n) = h(n)$$

Greedy best-first search (cont.)

Example

Let us see how this works for route-finding problems in Romania

- We use the **straight-line distance** heuristic, h_{SLD}

If the goal is Bucharest, we need straight-line distances to Bucharest

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

$h_{SLD}(\texttt{In}(\texttt{Arad})) = 366$

Greedy best-first search (cont.)

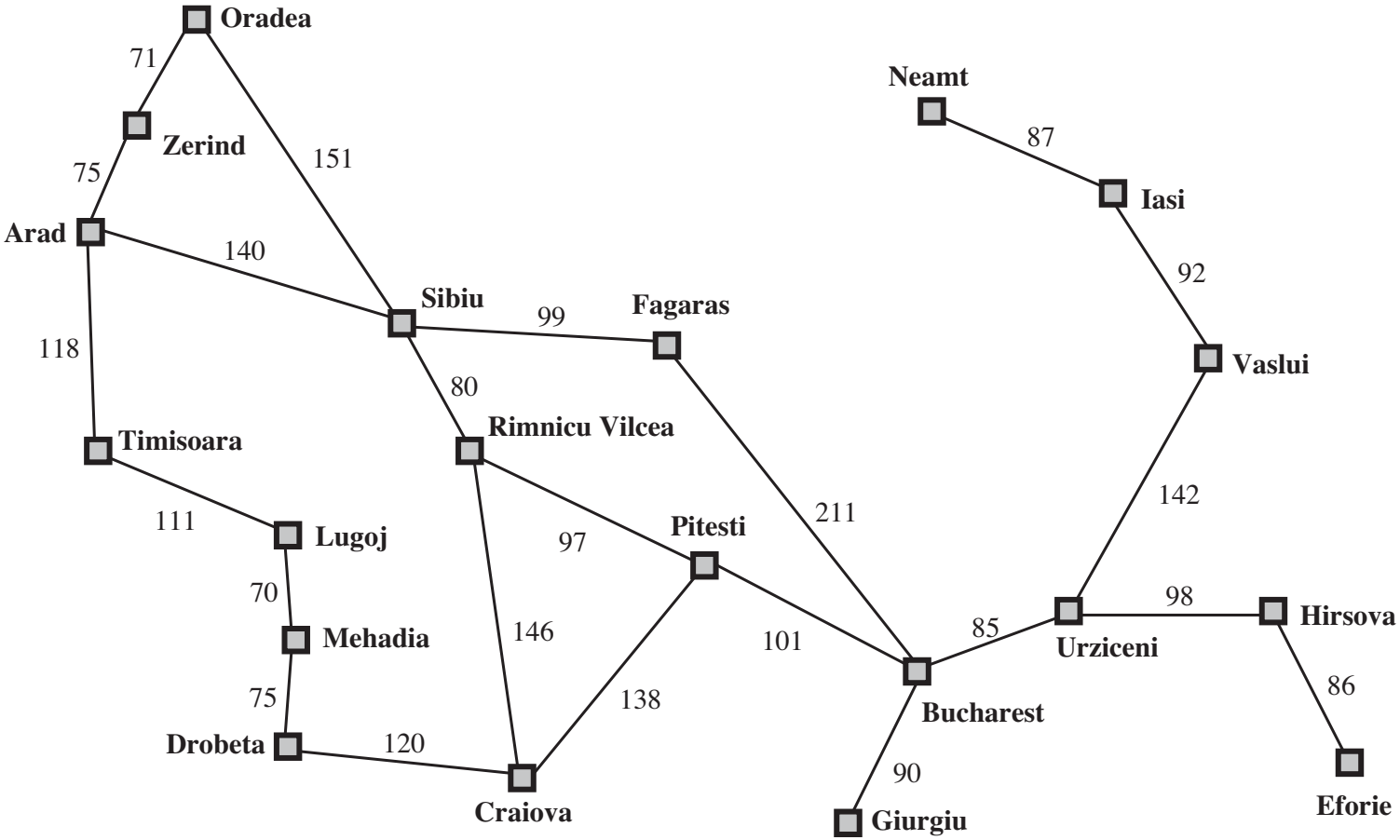
Values of h_{SLD} cannot be computed from the problem description

- Though h_{SLD}s are correlated with actual road distances
- It is a useful heuristic

Greedy best-first search (cont.)

The first node to be expanded from Arad is Sibiu

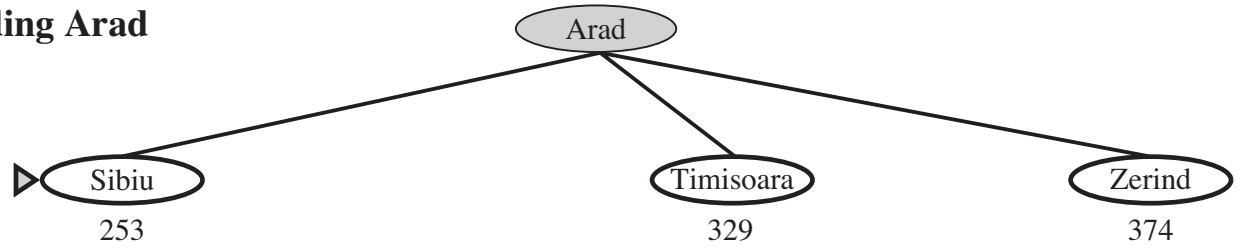
- Closer to Bucharest than either Zerind or Timisoara



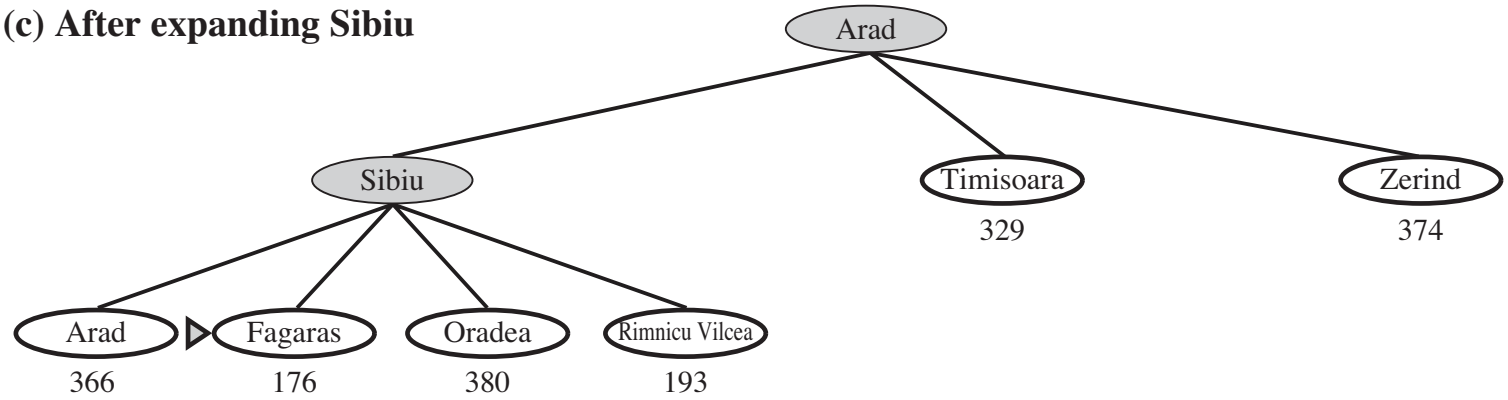
- Next node to be expanded is Fagaras, it is closest
- Fagaras in turn generates Bucharest, which is the goal

(a) The initial state

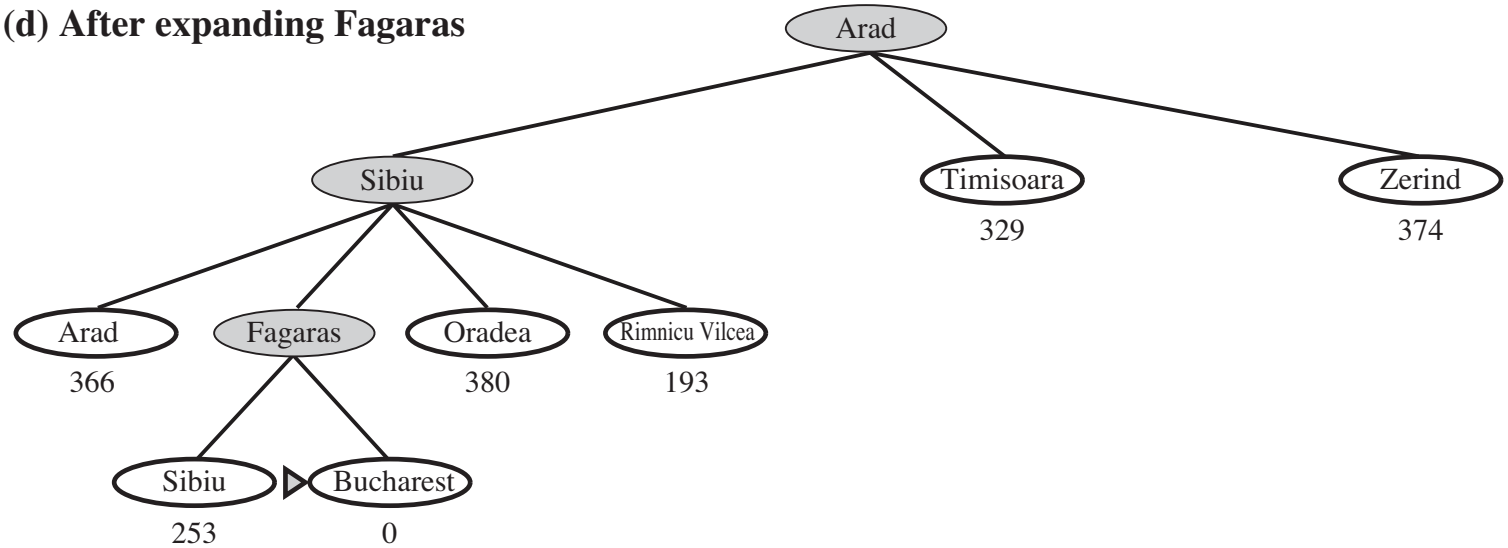
(b) After expanding Arad



(c) After expanding Sibiu



(d) After expanding Fagaras



Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Greedy best-first search (cont.)

Greedy best-first search using h_{SLD} finds a solution without expanding any node that is not on the solution path

- $\rightsquigarrow$ Its search cost is minimal

It is not optimal, as path via Sibiu and Fagaras to Bucharest is 32km longer than path through Rimnicu Vilcea and Pitesti

Remark

This shows why the algorithm is called 'greedy'

At each step, it tries to get as close to the goal as it can

Greedy best-first search (cont.)

Greedy best-first tree search is incomplete, even in finite state space

- The graph search version is complete in finite spaces
- But, it is not in infinite ones

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Greedy best-first search (cont.)

Example

Consider the problem of getting from Iasi to Fagaras

The heuristic suggests that Neamt be expanded first

- It is closest to Fagaras, but it is a dead end

Solution is to go first to Vaslui, a step that is farther from the goal according to the heuristic, and then to continue to Urziceni, Bucharest, and Fagaras

The algorithm will never find this solution

Expanding Neamt puts Iasi back into the frontier

- ⇝ Iasi is closer to Fagaras than Vaslui is
- ⇝ Iasi will be expanded again
- ⇝ Leading to an infinite loop

Greedy best-first search (cont.)

The worst-case time and space complexity for the tree version is

$$\mathcal{O}(b^m)$$

m is the maximum depth of the search space

With a good heuristic function, complexity can be reduced

A^* search
Informed search

A^* search

The most widely known form of best-first search is **A^* search**

It evaluates nodes by combining two costs

- $g(n)$, the cost to reach the node
- $h(n)$, the cost to get from the node to the goal

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

$g(n)$ gives the path cost from start node to node n

$h(n)$ is the estimated cost of the cheapest path from n to goal

$$f(n) = \text{estimated cost of the cheapest solution thru } n$$

We are trying to find the cheapest solution, a reasonable thing to try

- The node with the lowest value of $g(n) + h(n)$

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

A^* search (cont.)

It turns out that this strategy is more than just reasonable

- A^* search is both complete and optimal

Provided that the heuristic function $h(n)$ satisfies certain conditions

The algorithm is identical to **UNIFORM-COST-SEARCH**

- Except that A^* uses $(g + h)$ instead of g

A^*: Conditions for optimality

The first condition we require for optimality is about $h(n)$

- $h(n)$ must be an **admissible heuristic**

Admissible heuristics never overestimate the cost to reach goal

$g(n)$ is actual cost to reach n along current path and $f(n) = g(n) + h(n)$

An immediate consequence

$\rightsquigarrow$ $f(n)$ never overestimates the true cost

- (of a solution along the current path through n)

A^*: Conditions for optimality (cont.)

Remark

Admissible heuristics are by nature optimistic

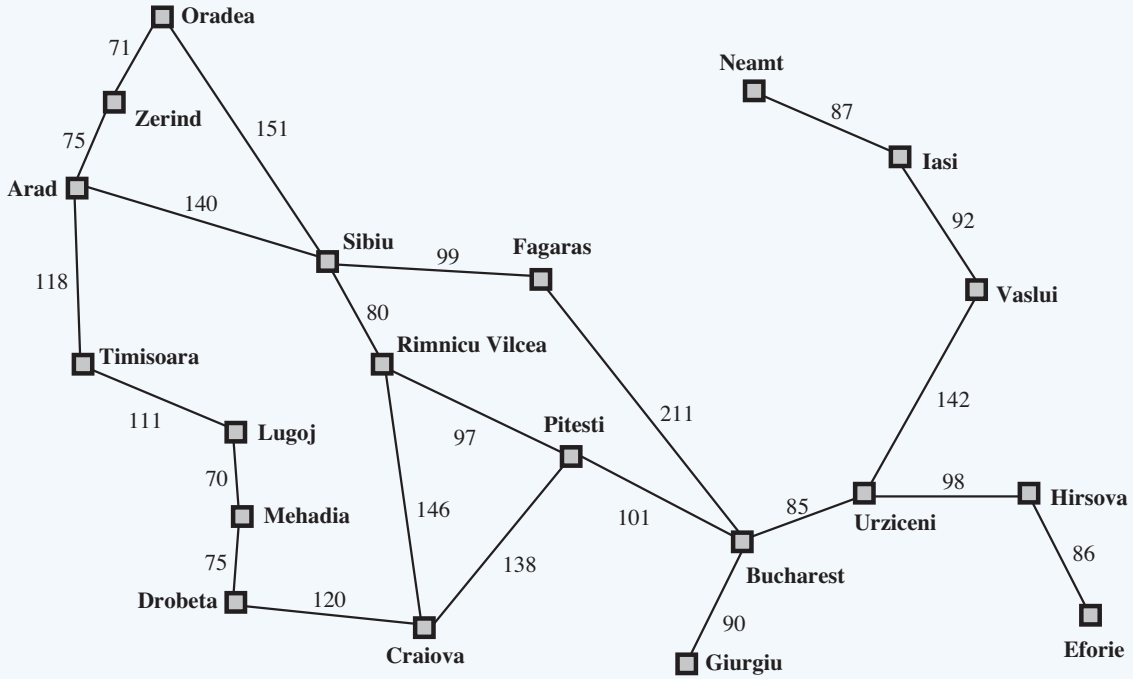
They think that the cost of solving the problem is less than it actually is

- An admissible heuristic is the straight-line distance h_{SLD}
- It is the shortest path between any two points
- It cannot be an overestimate

We show the progress of an A^* tree search for Bucharest

Example

Values of g are computed from step costs



The values of h_{SLD}

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

(a) The initial state

(b) After expanding Arad

(c) After expanding Sibiu

(d) After expanding Rimnicu Vilcea

(e) After expanding Fagaras

(f) After expanding Pitesti

A^*: Conditions for optimality (cont.)

Bucharest first appears on the frontier at step (e)

It is not selected for expansion

- f-cost (450) is higher than Pitesti's (417)

There might be a solution through Pitesti whose cost is as low as 417

- So, the algo won't settle for a solution that costs 450

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

A^*: Conditions for optimality (cont.)

A second, bit stronger condition is **consistency**/**monotonicity**

- It is required only for applications of A^* to graph search

Definition

Heuristic $h(n)$ is consistent if, for every node n and every successor n' of n generated by any action a, the estimated cost of reaching goal from n is no greater than the step cost of getting to n', plus the estimated cost of reaching goal from n'

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

This is a form of the general **triangle inequality**

Each side of a triangle cannot be longer than the sum of the other two

- Here, the triangle is formed by n, n' and goal G_n closet to n

A^*: Conditions for optimality (cont.)

For an admissible heuristic, the inequality makes perfect sense

Consider a route from n to G_n via n that is cheaper than $h(n)$

- It would violate that $h(n)$ is a lower bound on the cost to reach G_n

A^*: Optimality

A^* has two important properties

- The tree-search version is optimal, if $h(n)$ is admissible
- The graph-search version is optimal, if $h(n)$ is consistent

A^*: Optimality (cont.)

The argument to show that 'the graph-search version is optimal if $h(n)$ is consistent' mirrors the argument for optimality of uniform-cost search

- With g replaced by f, as in the A^* algo itself

Firstly, establishe that values of $f(n)$ along any path are non-decreasing

- If $h(n)$ is consistent

The proof follows directly from the definition of consistency

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

A^*: Optimality (cont.)

Suppose n' is a successor of n then $g(n') = g(n) + c(n, a, n')$

- for some action a

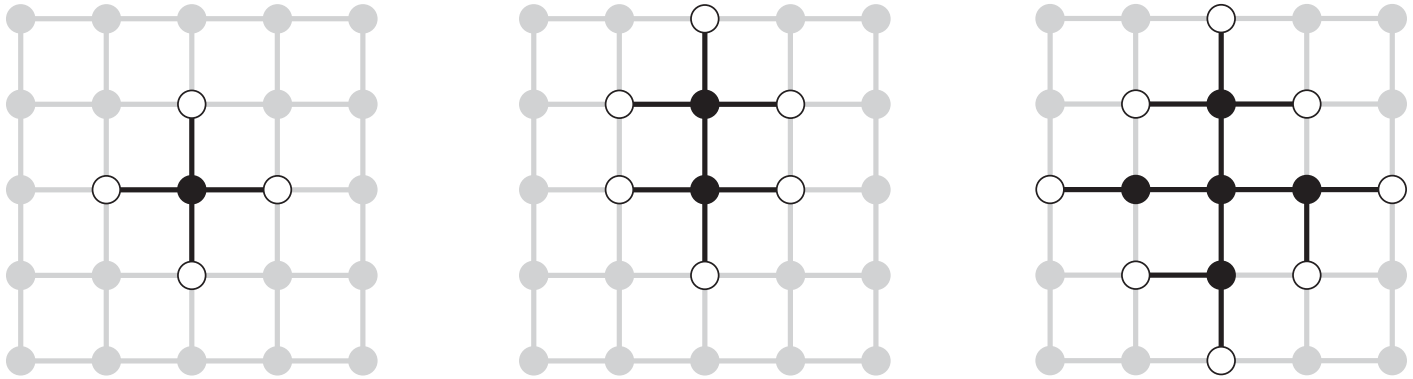
Then, we have

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

Secondly, prove that whenever A^* selects a node n for expansion, then the optimal path to that node has been found

- Otherwise, there would have to be another frontier node n' on the optimal path from start node to n
- (The graph separation property)

A^*: Optimality (cont.)



(a) (b) (c)

A^*: Optimality (cont.)

Consider the sequence of nodes expanded by A^* with GRAPH-SEARCH

- It is in a non-decreasing order of $f(n)$

The first goal node selected for expansion must be an optimal solution

- Because f is the true cost for goal nodes (with $h = 0$)

All later goal nodes will be at least as expensive

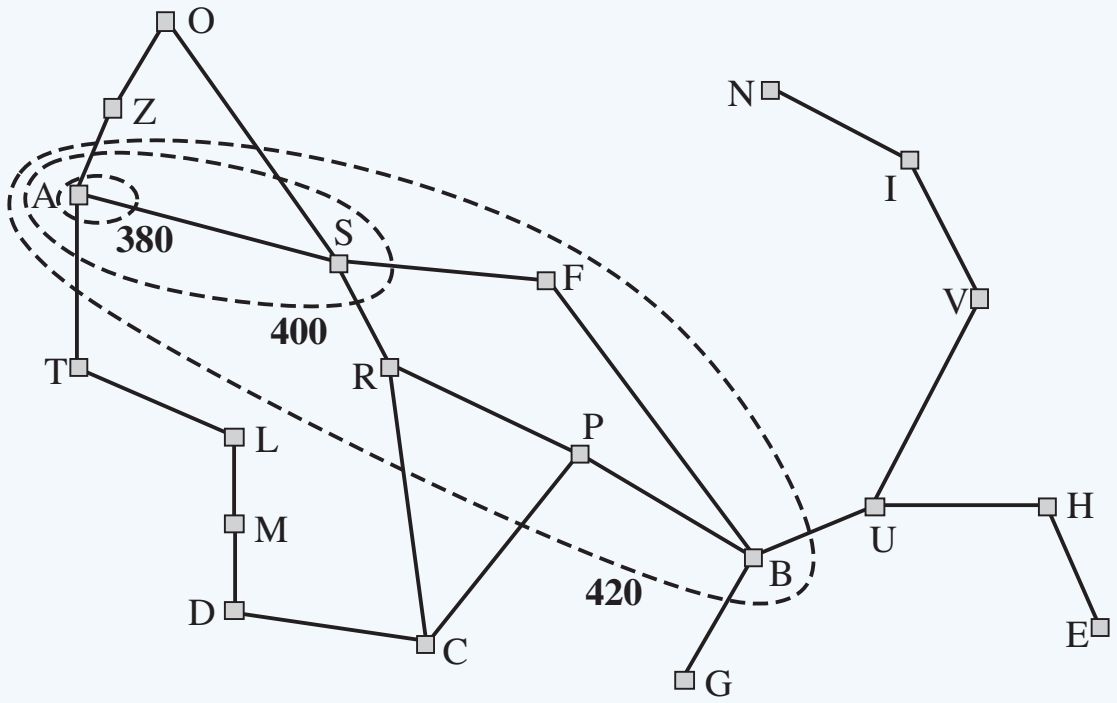
A^*: Optimality (cont.)

f-costs are non-decreasing along any path

$\rightsquigarrow$ we can draw **contours** in the state space

- Like in a topographic map

Example



Inside the contour labeled 400, all nodes have $f(n) \leq 400$

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

A^*: Optimality (cont.)

Consider uniform-cost search (A^* search using $h(n) = 0$)

- bands will be 'circular' around the start state

With accurate heuristics, the bands will stretch toward the goal state

- They become more narrowly focused around the optimal path

If C^* is the cost of the optimal solution path, then

- A^* expands all nodes with $f(n) < C^*$
- A^* might then expand some of the nodes right on the 'goal contour' where $f(n) = C^*$ before selecting a goal node

Completeness requires only finitely many nodes with cost $\leq C^*$

- This condition is true if all step costs exceed some finite ε
- and if b is finite

A^*: Optimality (cont.)

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

Example

Timisoara is not expanded even though it is a child of the root

- The subtree below Timisoara is **pruned**

h_{SLD} is admissible, the algorithm can safely ignore this subtree

- Optimality is still guaranteed

Pruning, or eliminating possibilities w/o having to examine them

- This is an important concept for AI

A^*: Optimality (cont.)

A^* is optimally efficient for any given consistent heuristic

- No other optimal algorithm is guaranteed to expand fewer nodes
- Except possibly through tie-breaking among nodes with $f(n) = C^*$

Any algorithm that does not expand all nodes with $f(n) < C^*$ is at risk

- It may miss the optimal solution

A^*: Optimality (cont.)

A^* search is complete, optimal, and optimally efficient

- Still A^* is not the answer to all searching needs

For most problems, the number of states within the goal contour search space is still exponential in the length of the solution

A^*: Optimality (cont.)

For problems with constant step costs

The growth in runtime as a function of the optimal solution depth d can be analysed in terms of **absolute error** or **relative error** of the heuristic

Definition

The absolute error is defined as

$$\Delta \equiv (h^* - h)$$

h^* is the actual cost of getting from root to goal

The relative error is

$$\varepsilon \equiv \frac{(h^* - h)}{h^*}$$

A^*: Optimality (cont.)

The complexity results depend on assumptions about state space

The simplest model studied is a state space with a **single goal**

- This is essentially a **tree** with **reversible actions**

Example

The 8-puzzle satisfies the first and third of these assumptions

Time complexity of A^* is exponential in maximum absolute error

$$\mathcal{O}(b^{\Delta})$$

For constant step costs, this is $\mathcal{O}(b^{\varepsilon d})$ with d the solution depth

A^*: Optimality (cont.)

For almost all heuristics in practical use, the absolute error is at least proportional to path cost h^*, so ε is constant or growing

- Time complexity is exponential in d

The effect of a more accurate heuristic

$$\mathcal{O}(b^{\varepsilon d}) = \mathcal{O}(b^{\varepsilon})^d$$

The effective branching factor (defined soon) is b

A^*: Optimality (cont.)

When the state space has many goal states (near-optimal ones, particularly)

- $\rightsquigarrow$ The search can be led astray from optimal path
- There is an extra cost proportional to the number of goals
- The cost is within a factor ε of the optimal cost

A^*: Optimality (cont.)

The general case of a graph, the situation is even worse

There can be exponentially many states with $f(n) < C^*$

- Even if the absolute error is bounded by a constant

A^*: Optimality (cont.)

Example

Consider a version of the vacuum world where agent can clean up any square for unit cost, without even having to visit it

- In that case, squares can be cleaned in any order

With N initially dirty squares, there are 2^N states

- Some subset has been cleaned

All of them are on an optimal solution path

- Satisfy $f(n) < C^*$, even if the heuristic has an error of 1

A^*: Optimality (cont.)

The complexity of A^* often makes it impractical

- Some variants can find suboptimal solutions

It is possible to design heuristics that are more accurate

- But, they are not strictly admissible

The use of a good heuristic still provides big savings

- Compared to uninformed search

A^*: Optimality (cont.)

Computation time is not, however, A^*'s main drawback

- All generated nodes are kept in memory
- (as do all GRAPH-SEARCH algorithms)
- A^* usually runs out of space before it runs out of time
- A^* is not practical for many large-scale problems

There are algorithms that overcome the space problem

- Without sacrificing optimality or completeness
- The cost is in execution time

We discuss these next

Solving by searching

UFC/DC
CK0031/CK0248
2017.2

Problem solving

Solving agents
Well-definedtness
Problem formulation

Solution search
Search algorithms
Performance metrics

Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search

Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search

Heuristics
Performance
Admissibility
Learning heuristics

Memory-bounded search
Informed search

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Memory-bounded search

The simplest way to reduce memory requirements for A^*

- Adapt the idea of iterative deepening to the heuristic search context
- **iterative-deepening** A^* (**ID**A^*) algorithm

The big difference between IDA^* and standard iterative deepening

- The cutoff used is the f-cost $(g + h)$ rather than the depth

At each iteration, the cutoff value is the smallest f-cost of any node that exceeded the cutoff on the previous iteration

Memory-bounded search (cont.)

IDA^* is practical for problems with unit step costs

- It avoids the overhead associated with keeping a sorted queue of nodes

IDA^* suffers from the difficulties with real valued costs

- As the iterative version of uniform-cost search

We briefly examine two other memory-bounded algorithms

Memory-bounded search (cont.)

Recursive best-first search (RBFS)

A recursive algorithm that attempts to mimic standard best-first search

- It uses a linear space

function RECURSIVE-BEST-FIRST-SEARCH(*problem*) **returns** a solution, or failure
  return RBFS(*problem*, MAKE-NODE(*problem*.INITIAL-STATE), ∞)

function RBFS(*problem*, *node*, *f_limit*) **returns** a solution, or failure and a new f-cost limit
  if *problem*.GOAL-TEST(*node*.STATE) **then return** SOLUTION(*node*)
  successors $\leftarrow$ [ ]
  for each *action* **in** *problem*.ACTIONS(*node*.STATE) **do**
    add CHILD-NODE(*problem*, *node*, *action*) into *successors*
  if *successors* is empty **then return** *failure*, ∞
  for each s **in** *successors* **do** /* update f with value from previous search, if any */
    $s.f \leftarrow \max(s.g + s.h, node.f))$
  loop do
    best $\leftarrow$ the lowest f-value node in *successors*
    if $best.f > f\_limit$ **then return** *failure*, $best.f$
    alternative $\leftarrow$ the second-lowest f-value among *successors*
    $result, best.f \leftarrow$ RBFS(*problem*, *best*, $\min(f\_limit, alternative)$)
    if $result \neq failure$ **then return** *result*

Memory-bounded search (cont.)

By structure, the algorithm is similar to recursive depth-first search

- It does not continue indefinitely down the current path
- It uses the f_limit variable to keep track of the f-value of best alternative path available from any ancestor of current node

Memory-bounded search (cont.)

If current node exceeds f_limit, recursion goes back to the alternative path

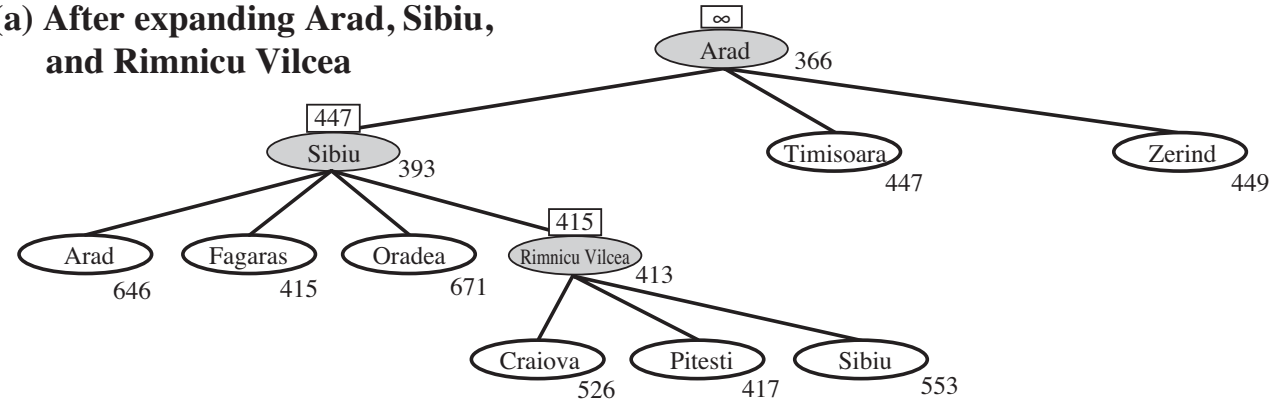
As recursion unwinds, the f-value of each node along the path is replaced

- A **backed-up value** is used (the best f-value of its children)

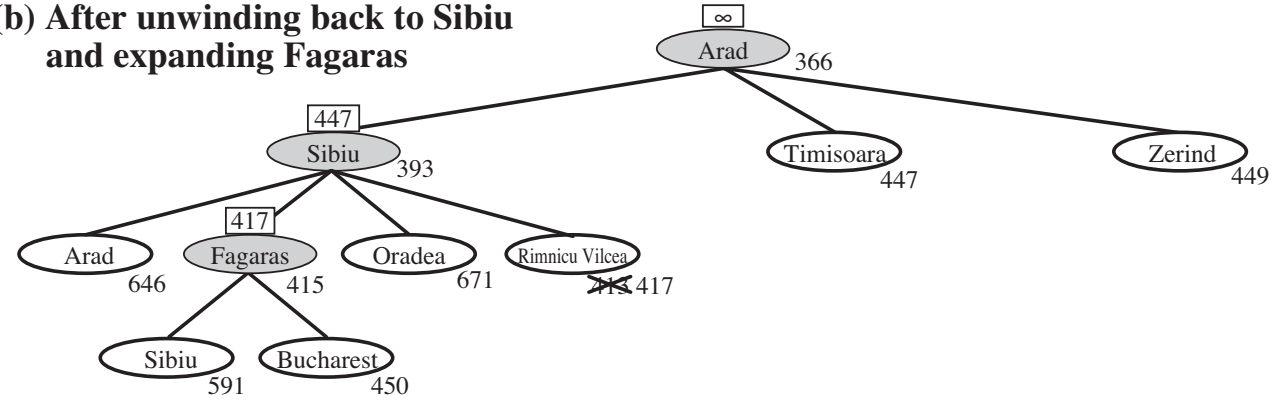
RBFS remembers the f-value of best leaf in the forgotten subtree

- Can decide whether it is worth re-expanding the subtree at later time

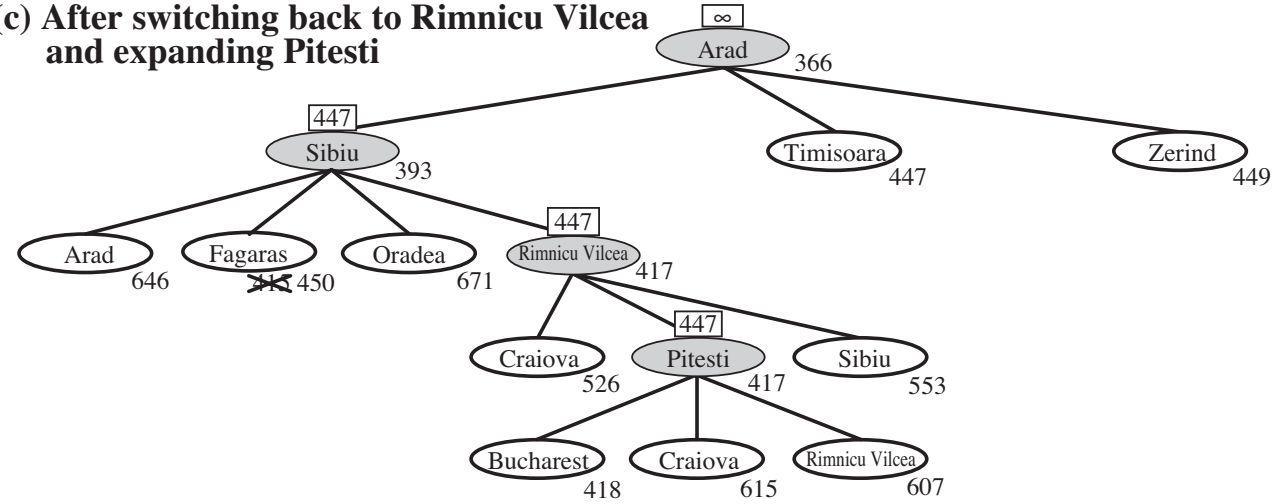
(a) After expanding Arad, Sibiu, and Rimnicu Vilcea



(b) After unwinding back to Sibiu and expanding Fagaras



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



Memory-bounded search (cont.)

f_limit value for each recursive call on top of each current node

- every node is labeled with its f-cost

Example

Path via Rimnicu Vilcea is followed

(a) Until current best leaf (Pitesti) is worse than best alternative path (Fagaras)

The recursion unwinds

(b) The best leaf value of the forgotten subtree (417) is backed up to Rimnicu Vilcea

Then Fagaras is expanded, revealing a best leaf value of 450

The recursion unwinds

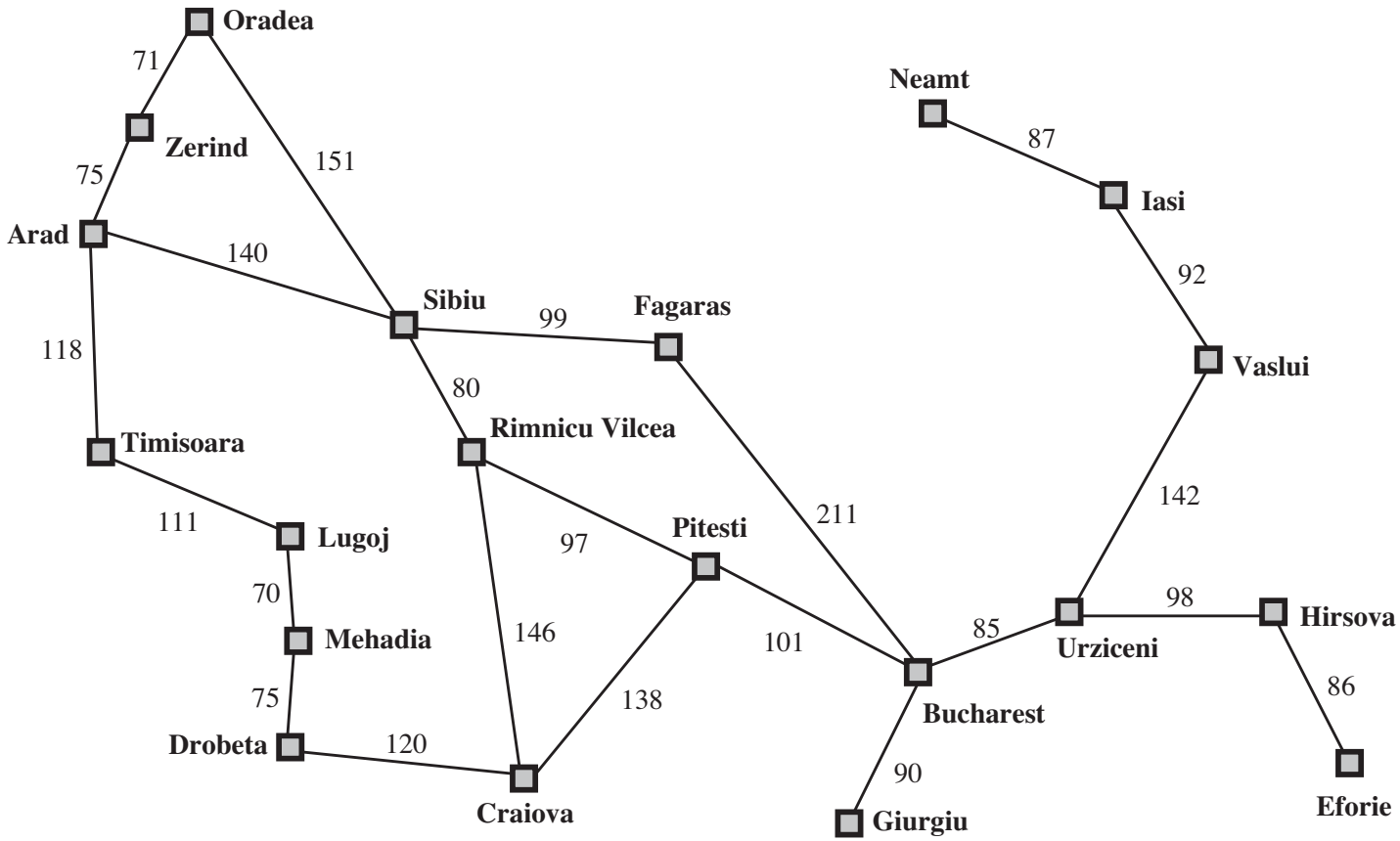
(c) The best leaf value of the forgotten subtree (450) is backed up to Fagaras

Then Rimnicu Vilcea is expanded

The best alternative path (through Timisoara) costs at least 447

- the expansion continues to Bucharest

Memory-bounded search (cont.)



Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Memory-bounded search (cont.)

RBFS is more efficient than IDA^*, still excessive node regeneration

Example

RBFS follows the path via Rimnicu Vilcea, then it 'changes its mind'

- It tries Fagaras, and then changes its mind back again

When current best path is extended, its f-value is likely to increase

- h is usually less optimistic for nodes closer to the goal

When this happens, the second-best path might become the best path

- The search has to backtrack to follow it

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Memory-bounded search (cont.)

Remark

Each mind change corresponds to an iteration of IDA^*

To recreate the best path and extend it one more node?

- It could require many re-expansions of forgotten nodes

Memory-bounded search (cont.)

Like A^* tree search, RBFS is an optimal algorithm

- If the heuristic function $h(n)$ is admissible

Its space complexity is linear in the depth of the deepest optimal solution

Its time complexity is rather difficult to characterise

- It depends on the accuracy of the heuristic function
- It depends on how often the best path changes as nodes are expanded

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Memory-bounded search (cont.)

IDA^* and RBFS suffer from using too little memory

Between iterations, IDA^* retains only a single number

$\rightsquigarrow$ The current f-cost limit

RBFS retains more information, but uses linear space

- RBFS has no way to make use of it
- (even if more memory were available)

Memory-bounded search (cont.)

Remark

IDA^* and RBFS may end up re-expanding the same states

- Because they forget most of what they have done

Also, they suffer the potentially exponential increase in complexity

- This is associated with redundant paths in graphs

Memory-bounded search (cont.)

It seems sensible to use all available memory

Two algorithms that do this

- **MA^* (memory-bounded A^*)**
- **SMA^* (simplified MA^*)**

SMA^* proceeds like A^*, best leaf is expanded until memory is full

- At this point, it cannot add a new node to he search tree
- It must first drop an old one
- SMA^* always drops the worst leaf node
- (the one with highest f-value)

Like RBFS, SMA^* backs up the value of the forgotten node to its parent

The ancestor of a forgotten subtree knows the best path in that subtree

Memory-bounded search (cont.)

With this info, SMA^* regenerates the subtree

- Only when all other paths look worse than forgotten path

What happens when all descendants of node n are forgotten?

- We do not know where to go from n

But, we still have an idea of how worthwhile it is to go anywhere from n

Memory-bounded search (cont.)

SMA^* expands the best leaf and deletes the worst leaf

- What if all the leaf nodes have the same f-value?

To avoid selecting the same node for deletion and expansion

- SMA^* expands newest best leaf and deletes oldest worst leaf

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Memory-bounded search (cont.)

SMA^* is complete, if there is any reachable solution

- (if the depth d of the shallowest goal node is less than the memory size in nodes)

SMA^* is optimal, if any optimal solution is reachable

- Otherwise, it returns the best reachable solution

Remark

SMA^* is a robust choice for finding optimal solutions

- Particularly when the state space is a graph
- Step costs are not uniform
- Node generation is expensive compared to the overhead of keeping frontier and explored set

Memory-bounded search (cont.)

On very hard problems, it can be the case that SMA^* is forced to switch back and forth continually among many candidate solution paths

- Only a small subset of which can fit in memory
- Memory limitations can make a problem intractable

Learning to search
Informed search

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Learning to search

We presented several fixed strategies (breadth-first, greedy best-first, ...)

Could an agent **learn how to search better**?

The answer is yes: The method rests on an important concept

- the **meta-level state space**

Each state in a meta-level state space captures the internal (computational) state of a program that is searching in an

- **object-level state space**, such as Romania

Learning to search (cont.)

The internal state of the A^* algorithm is the current search tree

Each action in the meta-level state space is a computation step

- It alters the internal state

Each computation step in A^* expands a leaf node

- It adds its successors to the tree

Solving by searching

UFC/DC
CK0031/CK0248
2017.2

Problem solving

Solving agents
Well-definedtness
Problem formulation

Solution search
Search algorithms
Performance metrics

Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search

Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search

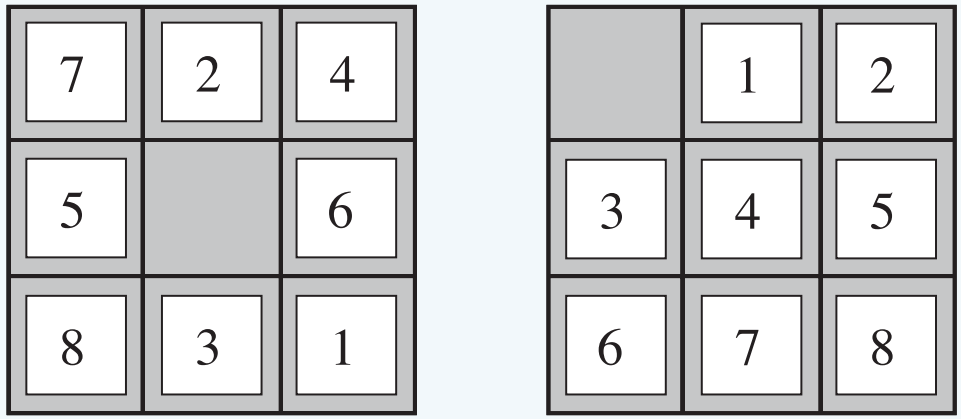
Heuristics
Performance
Admissibility
Learning heuristics

Heuristic functions
Solving by searching

Heuristic function

Heuristics, by looking at heuristics for the 8-puzzle

Example



Start State Goal State

It was one of the earliest heuristic search problems

- Slide tiles horizontally or vertically into the empty space
- Until the configuration matches the goal configuration

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Heuristic function (cont.)

The average solution cost for a randomly generated 8-puzzle is about 22 steps

The branching factor is about 3:

- When the empty tile is in the middle, four moves are possible
- When the empty tile is in a corner, two moves are available
- When the empty tile is along an edge, three available moves

An exhaustive tree search to depth 22 would look at about

$$3^{22} \approx 3.1 \times 10^{10} \text{ states}$$

- A graph search would cut this down by a factor of $\sim 170K$
- As only $9!/2 = 181440$ distinct states are reachable

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Heuristic function (cont.)

This is a manageable number

But, the number for the 15-puzzle is roughly 10^{13}

- Need to find a good heuristic function

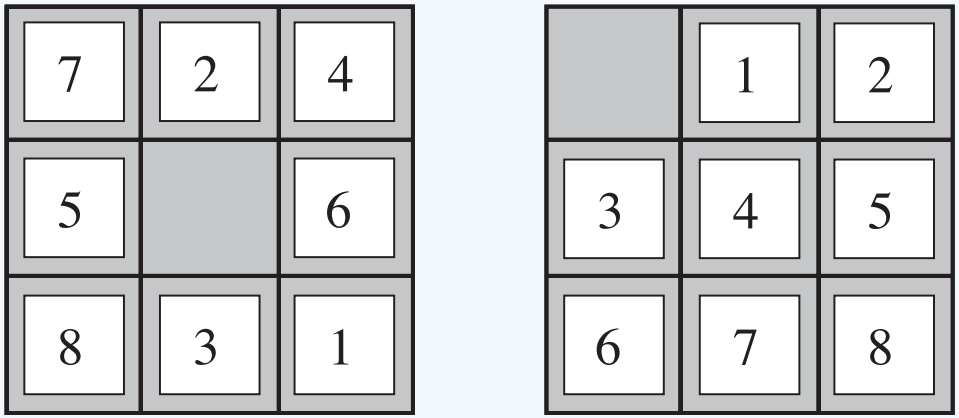
We want to find the shortest solutions by using A^*

We need a heuristic function that never overestimates the number of steps

- There is a long history of such heuristics for the 15-puzzle

Heuristic function (cont.)

Example



Start State Goal State

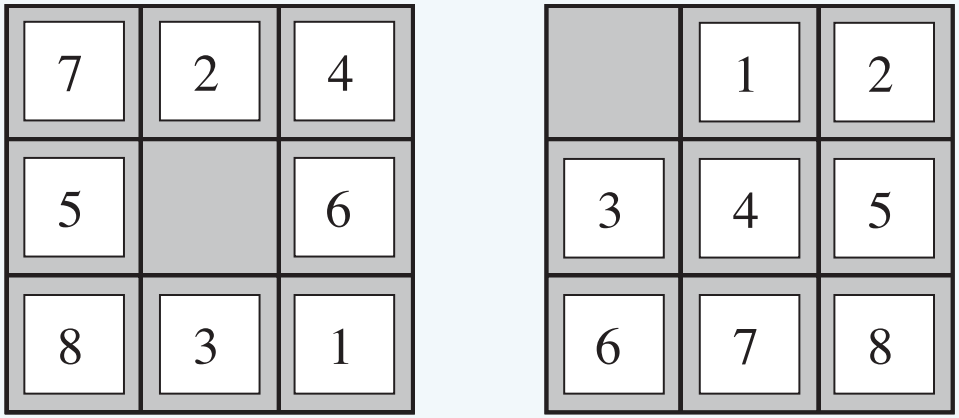
h_1 equals the number of misplaced tiles

- All of the eight tiles are out of position
- The start state would have $h_1 = 8$
- h_1 is an admissible heuristic

Any tile that is out of place must be moved at least once

Heuristic function (cont.)

Example



Start State Goal State

h_2 is the sum of the distances of the tiles from their goal positions

- The distance is the sum of horizontal and vertical distances
- This is called the **city block** or **Manhattan distance**
- h_2 is also admissible

All any move can do is move one tile one step closer to goal

- Tiles 1 to 8 in start state give a distance of

$$h_2 = 3 + 1 + 2 + 2 + 2 + 3 + 3 + 2 = 18$$

Heuristic function (cont.)

Neither of these overestimates the true solution cost, which is 26

Accuracy and performance

Heuristic function

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Accuracy and performance

To characterise heuristic's quality: **effective branching factor** b^*

- If the total number of nodes generated by A^* is N and the solution depth is d, then b^* is the branching factor that a uniform tree of depth d would have to have to contain $N + 1$ nodes

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

Example

If A^* finds a solution at depth $d = 5$ using $N + 1 = 52$ nodes, then

- The effective branching factor is $b^* = 1.92$

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Accuracy and performance

The effective branching factor can vary across problem instances, but usually it is fairly constant for sufficiently hard problems

- The existence of an effective branching factor follows from the result, mentioned earlier, that the number of nodes expanded by A^* grows exponentially with solution depth

Thus, experimental measurements of b^* on a small set of problems can provide a good guide to the heuristic's overall usefulness

Remark

A well-designed heuristic would have a value of b^* close to 1, allowing fairly large problems to be solved at reasonable cost

Accuracy and performance (cont.)

To test heuristic functions h_1 and h_2, consider 1.2K random probs with solution lengths from 2 to 24 (100 for each even number) and solve them with iterative deepening search and A^* tree search

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

Average number of nodes generated and effective branching factor

Accuracy and performance (cont.)

Results say that h_2 is better than h_1, and much better than IDS

- Even for small problems with $d = 12$, A^* with h_2 is 50K times more efficient than uninformed iterative deepening search

One might ask whether h_2 is always better than h_1

- The answer is 'essentially, yes'

From the definitions of h_1 and h_2, for any node n

- h_2 **dominates** h_1, or $h_2(n) \geq h_1(n)$

Accuracy and performance (cont.)

Domination translates directly into efficiency

- A^* using h_2 will never expand more nodes than A^* using h_1 (except possibly for some nodes with $f(n) = C^*$)

The argument is simple, recall the observation that every node with $f(n) < C^*$ will surely be expanded

- Every node with $h(n) < C^* - g(n)$ will surely be expanded

But because h_2 is at least as big as h_1 for all nodes, every node that is surely expanded by A^* search with h_2 will also surely be expanded with h_1, and h_1 may cause other nodes to be expanded

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Accuracy and performance (cont.)

Remark

It is generally better to use a heuristic function with higher values

- Provided it is consistent and that computation time for the heuristic is passable

Solving by searching

UFC/DC
CK0031/CK0248
2017.2

Problem solving

Solving agents

Well-definedtness

Problem formulation

Solution search

Search algorithms

Performance metrics

Uninformed search

Breadth-first search

Uniform-cost search

Depth-first search

Depth-limited search

Iterative deepening depth-first search

Bidirectional search

Informed searches

Greedy best-first

A^* search

Memory-bounds

Learning to search

Heuristics

Performance

Admissibility

Learning heuristics

Admissible heuristics

Heuristic function

Admissible heuristics from relaxed problems

Admissible heuristics

Admissible heuristics from relaxed problems

Both h_1 (misplaced tiles) and h_2 (Manhattan distance) are good heuristics

- (for the 8-puzzle)

We saw that h_2 is better

- How might one have come up with h_2?

Is it possible for a computer to invent such a heuristic mechanically?

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Admissible heuristics from relaxed problems

For the 8-puzzle, h_1 and h_2 are estimates of the remaining path length

- Perfectly accurate path lengths for simplified versions of the puzzle

Example

If the rules were changed so that a tile could move anywhere instead of just to the adjacent empty square, then

- h_1 would give the number of steps in the shortest solution

If a tile could move one square in any direction, even onto an occupied square, then

- h_2 would give the number of steps in the shortest solution

Admissible heuristics from relaxed problems (cont.)

Definition

Relaxed problems

Problems with fewer restrictions on actions

- *The state-space graph of a relaxed problem is a **super-graph** of the original state space*
- *The removal of restrictions creates added edges in the graph*

As the relaxed problem adds edges, any optimal solution in the original problem is, by definition, a solution in the relaxed problem

- The relaxed problem may have better solutions
- (if the added edges provide short cuts)

Admissible heuristics from relaxed problems (cont.)

The cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem

Because the derived heuristic is an exact cost for the relaxed problem, it obeys the triangle inequality and is thus **consistent**

Admissible heuristics from subproblems

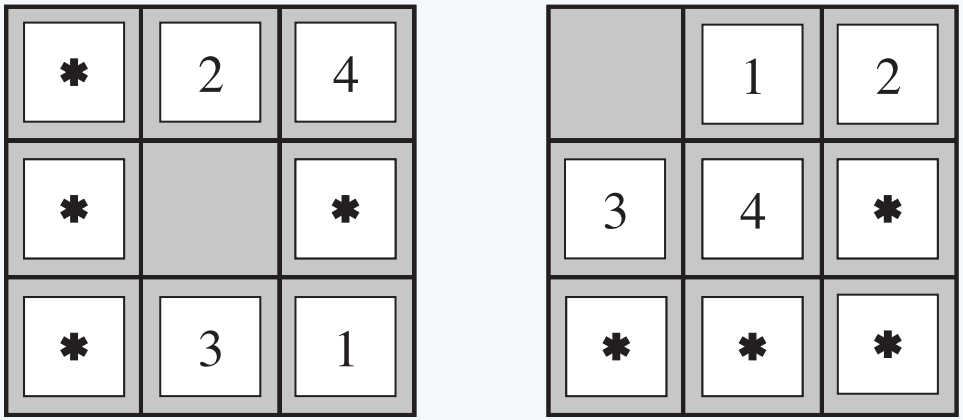
Admissible heuristics

Admissible heuristics from subproblems

Admissible heuristics can be derived from the solution cost of a **subproblem**

Example

The figure shows a subproblem of the 8-puzzle instance



Start State Goal State

The subproblem is getting tiles 1, 2, 3, 4 into position

- without worrying about what happens to the other ones

Admissible heuristics from subproblems (cont.)

The cost of the optimal solution of this subproblem is a lower bound

- On the cost of the complete problem

It can be more accurate than the Manhattan distance

1) The idea behind **pattern databases** is to store exact solution costs
   For every possible subproblem instance

Every possible configuration of the four tiles and the blank

- Location of other tiles is irrelevant for solving subproblem
- But, moves of those tiles do count toward the cost

2) Then compute an admissible heuristic h_{DB} for each complete state encountered during a search

By looking up the corresponding subproblem configuration in the database

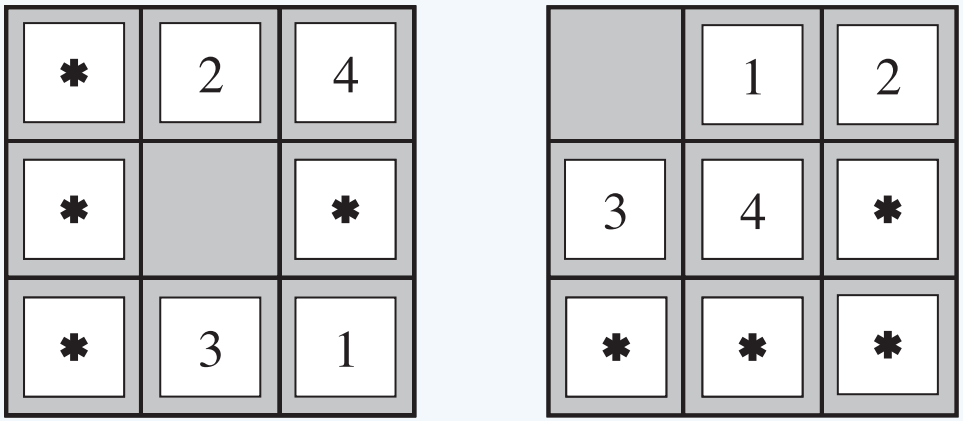
Admissible heuristics from subproblems (cont.)

The database itself is constructed by searching back from the goal

- Recording the cost of each new pattern encountered

The expense of this search is amortised over subsequent instances

Example



Start State Goal State

The choice of $1-2-3-4$ is fairly arbitrary

- We can construct databases for $5-6-7-8$, $2-4-6-8$, etc.

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Admissible heuristics from subproblems (cont.)

Each database yields an admissible heuristic

- These heuristics can be combined
- (by taking the maximum value)

A combined heuristic like this is more accurate than Manhattan distances

Remark

Number of nodes generated when solving random 15-puzzles can be reduced

- A factor of 1K

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Admissible heuristics from subproblems (cont.)

Example

Possible to heuristics from $1-2-3-4$ and $5-6-7-8$ databases?

The two seem not to overlap ...

- Would this still give an admissible heuristic? Answer is no!

Solutions to $1-2-3-4$ and $5-6-7-8$ subproblem could share moves

- It is unlikely that $1-2-3-4$ can be moved into place without touching $5-6-7-8$, and vice versa

What if we do not count those moves?

- We record not the total cost of solving the $1-2-3-4$ subproblem
- Just the number of moves involving $1-2-3-4$

Then it is easy to see that the sum of the two costs is still a lower bound

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Admissible heuristics from subproblems (cont.)

This is the idea behind **disjoint pattern databases**

Example

With such databases, we can solve random 15-puzzles in milliseconds

- (the number of nodes generated is reduced by a factor of 10K)
- (compared with the use of Manhattan distance)

For 24-puzzles, a speedup of a factor of 1M can be obtained

Disjoint pattern databases work for sliding-tile puzzles

- The problem can be divided up
- Each move affects only one subproblem
- Only one tile is moved at a time

Solving by searching

UFC/DC
CK0031/CK0248
2017.2

Problem solving

Solving agents

Well-definedtness

Problem formulation

Solution search

Search algorithms

Performance metrics

Uninformed search

Breadth-first search

Uniform-cost search

Depth-first search

Depth-limited search

Iterative deepening depth-first search

Bidirectional search

Informed searches

Greedy best-first

A^* search

Memory-bounds

Learning to search

Heuristics

Performance

Admissibility

Learning heuristics

Learning heuristics

Heuristic function

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Learning heuristics

A heuristic function $h(n)$ is supposed to estimate the cost of a solution

- From the state at node n

How could an agent build such a function?

$\rightsquigarrow$ Devise relaxed problems for which an optimal solution can be found

Another solution is to learn from **experience**

Example

- Experience means solving lots of 8-puzzles, for instance
- Each optimal solution to an 8-puzzle problem provides examples
- $h(n)$ can be learned from such examples
- Each example consists of a state from the solution path
- And, the actual cost of the solution from that point

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Learning heuristics (cont.)

A learning algorithm can be used to build a function $h(n)$

Predict solution costs for other states that arise during search

- Applicable techniques are neural nets, decision trees, ...
- The reinforcement learning methods are also applicable

Inductive learning methods work best when supplied with **features**

- Features of a state that are relevant to predicting the state's value

Works better than with just the raw state description

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Learning heuristics (cont.)

Example

The feature 'number of misplaced tiles'

It may be helpful in predicting the actual distance of a state from the goal

- Let's call this feature $x_1(n)$

We could take 100 randomly generated 8-puzzle configurations

- We gather statistics on their actual solution costs
- We may find that when $x_1(n)$ is 5, the average solution cost is ~ 14
- ... and so on

Given these data, the value of x_1 can be used to predict $h(n)$

Learning heuristics (cont.)

Of course, we can use several features

Example

A possible feature $x_2(n)$

'number of pairs of adjacent tiles not adjacent in the goal state'

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Learning heuristics (cont.)

How should $x_1(n)$ and $x_2(n)$ be combined to predict $h(n)$?

A common approach is to use a linear combination

$$h(n) = c_1 x_1(n) + c_2 x_2(n)$$

c_1 and c_2 are constant

They are adjusted to give the best fit to the data on solution costs

Problem solving
Solving agents
Well-definedtness
Problem formulation
Solution search
Search algorithms
Performance metrics
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first
A^* search
Memory-bounds
Learning to search
Heuristics
Performance
Admissibility
Learning heuristics

Learning heuristics (cont.)

Example

One expects both c_1 and c_2 to be positive

As misplaced tiles and incorrect adjacent pairs make the problem harder

This heuristic does satisfy the condition that $h(n) = 0$ for goal states

- It is not necessarily admissible or consistent