Solving by searching

Artificial intelligence (CK0031)

Francesco Corona

Solving by searching
UFC/DC
AI (CK0031)
2016.2

Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Problem solving

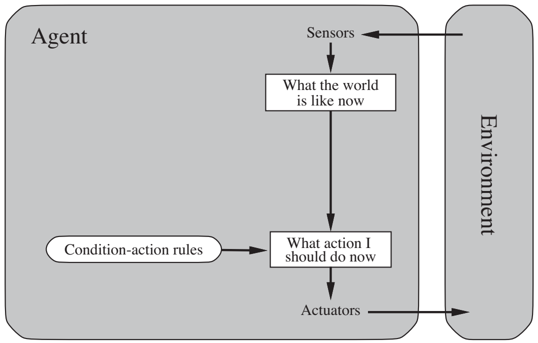
Solving by searching
UFC/DC
AI (CK0031)
2016.2

Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Problem solving

The simplest agents we discussed were the reflex agents, which base their actions on a direct mapping from states to actions

- They cannot operate well in environments for which this mapping would be too large to store and too long to learn

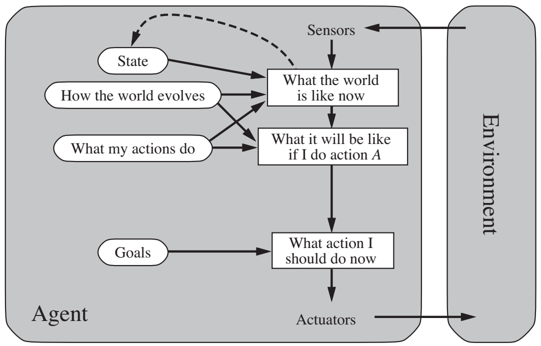


Solving by searching
UFC/DC
AI (CK0031)
2016.2

Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Problem solving (cont.)

Goal-based agents use future actions and desirability of outcomes



Problem solving (cont.)

We study one kind of goal-based agent: **Problem-solving agent**

- Problem-solving agents use atomic representations (states as wholes, no internal structure visible to the algorithms)

Goal-based agents that use factored or structured representations

- **Planning agents**

We begin with some definitions of problems and their solutions

- Several examples to illustrate these definitions

We then describe several general-purpose search algorithms

- They can be used to solve these problems

Solving by searching
UFC/DC
AI (CK0031)
2016.2

Problem solving
Problem-solving agents
Well-definedtness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Problem solving (cont.)

Several **uninformed search** algorithms, algorithms that are given no information about the problem other than its definition

- Although some of these algorithms can solve any solvable problem, none of them can do so efficiently

Informed search algorithms, on the other hand, can do quite well, given some guidance on where to look for solutions

Only the simplest kind of task environment, for which the solution to a problem is always a fixed sequence of actions

- The more general case (where the agent's future actions may vary depending on future percepts) is handled separately

We shall use the concepts of asymptotic complexity

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

Solving by searching
UFC/DC
AI (CK0031)
2016.2

Problem solving
Problem-solving agents
Well-definedtness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Problem-solving agents

Solving by searching
UFC/DC
AI (CK0031)
2016.2

Problem solving
Problem-solving agents
Well-definedtness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Problem-solving agents

Agents are expected to maximize their performance measure

- Achieving this is sometimes simplified if the agent can adopt a goal and aim at satisfying it

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

Problem-solving agents (cont.)

Example

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

The agent's performance measure contains many factors

- It wants to improve suntan, improve Romanian, take in the sights, enjoy nightlife (such as it is), avoid hangovers, etc.

The decision problem is a complex one involving many trade-offs

Suppose the agent has a nonrefundable ticket
to fly out of Bucharest the following day

It makes sense for the agent to adopt the **goal**: Get to Bucharest

Courses of action that do not reach Bucharest on time
can be rejected, and need no further consideration

- The agent's decision problem is greatly simplified

Problem-solving agents (cont.)

Goals help organise behaviour by limiting the objectives the agent is trying to achieve and hence the actions it needs to consider

- **Goal formulation**, based on current situation and agent's performance measure, is the first step in problem solving

Definition

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

- exactly those states in which the goal is satisfied

The agent's task is to find out how to act, now
and in the future, so that it reaches a goal state

Problem-solving agents (cont.)

Before it can do this, it needs to decide (or we need to decide on its behalf) what sorts of actions and states it should consider

- If it were to consider actions at the level of 'move left foot forward an inch' or 'turn steering wheel one degree left,' the agent would prolly never find its way out of the parking lot
- At that level of detail there is too much uncertainty in the world and there would be too many steps in a solution

Definition

Problem formulation is the process of deciding
what actions and states to consider, given a goal

Problem-solving agents (cont.)

Example

- Let us assume that the agent will consider actions at the level of driving from one major town to another
- Each state thus corresponds to being in a some town

Our agent has now adopted the goal of driving **to Bucharest**

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

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

None of these achieves the goal, so unless the agent is familiar with the geography of Romania, it will not know which road to follow

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

Problem-solving agents (cont.)

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

- With no additional information (**environment is unknown**) then it is has no choice but to try one action at random

Example

Suppose the agent has a map of Romania

- The point of a map is to provide the agent with information about states it might get itself into and actions it can take

The agent can use this information to consider subsequent stages of a hypothetical journey via each of the three towns

- Find a journey that eventually gets to Bucharest

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

- Achieve goal by carrying out the actions (drive)

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Problem-solving agents (cont.)

In general, an agent with several immediate options of unknown value can decide what to do by first *examining future actions* that eventually lead to states of known value

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Problem-solving agents (cont.)

Assumption

Environment is observable: Agent always knows current state

- For the agent driving in Romania, it is reasonable to suppose that each city on the map has a sign indicating its presence

Environment is discrete: At any given state there are only finitely many actions to choose from

- This is true for navigating in Romania because each city is connected to a small number of other cities

Environment is known: Agent knows which states are reached by each action

- An accurate map suffices to meet this condition for navigation

Environment is deterministic: Each action has one outcome

- Ideally, this is true for the agent in Romania as it means that if it chooses to drive from Arad to Sibiu, it ends up in Sibiu

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Problem-solving agents (cont.)

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

- In general it could be a branching strategy that recommends different future actions depending on what percepts arrive

Example

Under sub-ideal conditions, the agent may plan to drive from Arad to Sibiu and to Rimnicu Vilcea but may need a contingency plan in case it gets by accident to Zerind instead of Sibiu

If the agent knows the initial state and the environment is known and deterministic, it knows exactly where it will be after the first action and what it will perceive

- Since only one percept is possible after the first action, the solution can specify only one possible second action, ...

Problem-solving agents (cont.)

Definition

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

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

Execution phase: Once a solution is found, the actions it recommends can be carried out

Simple design for the agent: Formulate $\Rightarrow$ search $\Rightarrow$ execute

Solving by searching
UFC/DC
AI (CK0031)
2016.2

Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
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
```

Solving by searching
UFC/DC
AI (CK0031)
2016.2

Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Problem-solving agents (cont.)

While the agent is executing the solution sequence it ignores its percepts when choosing an action because it knows in advance

- An agent that carries out its plans with its eyes closed, so to speak, must be quite certain of what is going on

Control theorists call this an **open-loop** system, because ignoring the percepts breaks the loop between agent and environment

Solving by searching
UFC/DC
AI (CK0031)
2016.2

Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Problem-solving agents (cont.)

- After formulating a goal and a problem to solve, the agent calls a search procedure to solve it
- It then uses the solution to guide its actions, doing whatever the solution recommends as the next thing to do (typically, the first action of the sequence) and then removing that step from the sequence
- Once the solution has been executed, the agent will formulate a new goal

Well-definedness

Problem solving agents

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Well-definedness

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

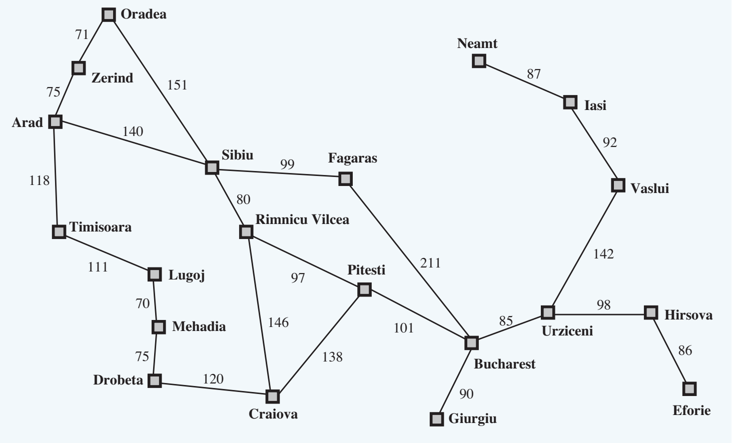
- **Initial state**
- **Actions**
- **Transition model**
- **Goal test**
- **Path cost**

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

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)`

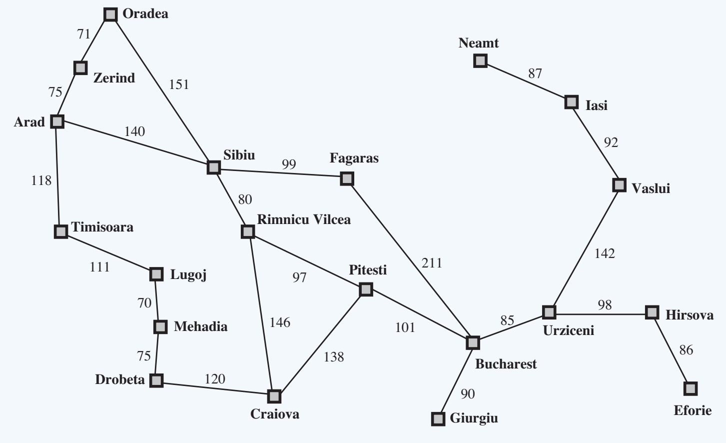


Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

The **actions** are the possible actions available to the agent
Given a particular state `s`, function `ACTIONS(s)` returns
the set of actions that can be executed in `s`

Example

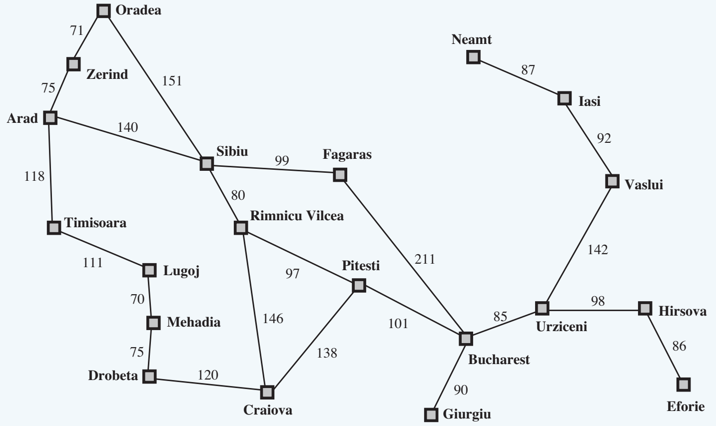
From state `In(Arad)`: {`Go(Sibiu)`, `Go(Timisoara)`, `Go(Zerind)`}



A **transition model** is formal description of what each action does
Function RESULT(s, a) returns the state
that results from doing action a in state s

Example

$\texttt{RESULT}(\texttt{In}(\texttt{Arad}), \texttt{Go}(\texttt{Zerind})) = \texttt{In}(\texttt{Zerind})$



Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Well-definedness (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** in which
the nodes are states and the links between nodes are actions

- The map of Romania can be interpreted as a state-space graph if we view each road as standing for two driving actions, one in each direction

Definition

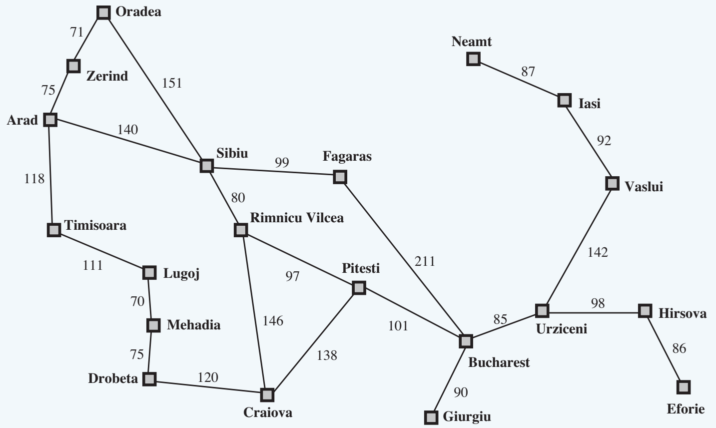
A **path** in the state space is a sequence of
states connected by a sequence of actions

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

The **goal test** determines whether a given state is a goal state
Sometimes there is an explicit set of possible goal states, and
the test simply checks whether the given state is one of them

Example

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



Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Well-definedness (cont.)

Sometimes the goal is specified by an abstract property
rather than an explicitly enumerated set of states

- In chess, the goal is to reach a state called 'checkmate,
'where the opponent's king is under attack and can't escape

Well-definedness (cont.)

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

- The problem-solving agent chooses a cost function that reflects its own performance measure

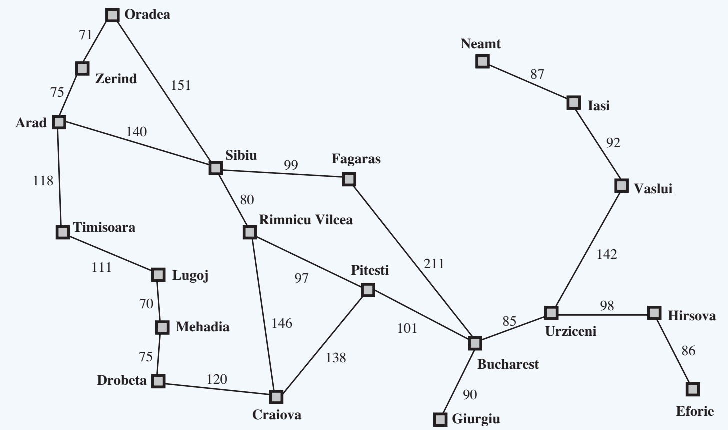
Example

For the agent trying to get to Bucharest, time essential, so the cost of a path might be its length in kilometers

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

The path cost is the sum of costs of individual actions along path
The **step cost** of taking action `a` in state `s` to reach state `s'` is nonnegative $c(\texttt{s}, \texttt{a}, \texttt{s'})$

Example



Step costs for Romania can be defined as route distances

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Well-definedness (cont.)

These elements define a problem and 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 that leads from the initial state to a goal state

Solution quality is measured by the path cost function, and an **optimal solution** has the lowest path cost among all solutions

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Problem formulation

Problem solving agents

Problem formulation

Example

A formulation of the problem of getting to Bucharest in terms of the initial state, actions, transition model, goal test, and path cost

- This formulation seems reasonable, but it is still a model (an abstract mathematical description) and not the real thing

Compare the simple state description we have chosen, `In(Arad)`, to an actual trip, where the state of the world is the real state

- Traveling companions, current radio program, scenery out of the window, proximity of law enforcement officers, distance to the next rest stop, condition of the road, weather, ...

All these considerations are left out of state descriptions because they are irrelevant to the problem of finding a route to Bucharest

Solving by searching

UFC/DC
AI (CK0031)
2016.2

Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Problem formulation (cont.)

- **Abstraction**: Process of removing detail from a representation

In addition to abstracting the state description, we must abstract the actions themselves

Example

A driving action has many effects, besides changing the location of the vehicle and its occupants, it takes up time, consumes fuel, generates pollution, and changes the agent (travel is broadening)

- Our formulation takes into account only change 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, ...
- We don't specify actions at the level of 'turn steering wheel to the left by one degree'

Solving by searching

UFC/DC
AI (CK0031)
2016.2

Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Problem formulation (cont.)

More precise about defining the appropriate level of abstraction?

- Think of the abstract states and actions we have chosen as corresponding to large sets of detailed world states and detailed action sequences
- Now consider a solution to the abstract problem

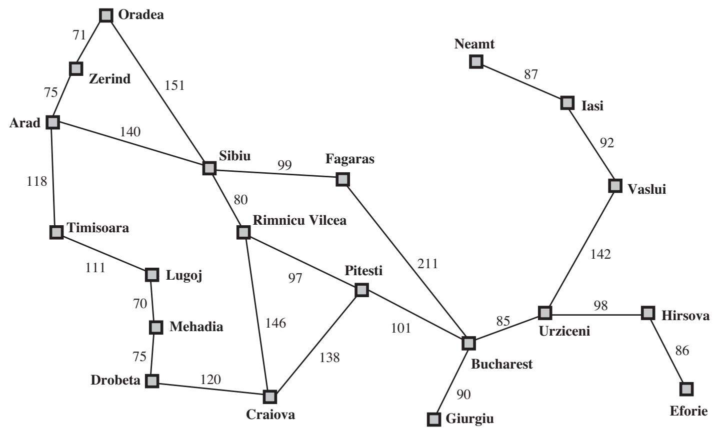
Solving by searching

UFC/DC
AI (CK0031)
2016.2

Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Problem formulation (cont.)

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



This abstract solution corresponds to a large number of more detailed paths

Problem solving
Problem-solving agents
Well-definedtness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Problem formulation (cont.)

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

Definition

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

- A sufficient condition is that for every detailed state that is 'in Arad,' there is a detailed path to some state that is 'in Sibiu,' and so on

Solving by searching

UFC/DC
AI (CK0031)
2016.2

Problem solving
Problem-solving agents
Well-definedtness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Problem formulation (cont.)

The abstraction is useful if carrying out each of the actions in the solution is easier than the 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

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

Were it not for the ability to construct useful abstractions, intelligent agents would be swamped by the real world

Solving by searching

UFC/DC
AI (CK0031)
2016.2

Problem solving
Problem-solving agents
Well-definedtness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Examples

Solving by searching

Solving by searching

UFC/DC
AI (CK0031)
2016.2

Problem solving
Problem-solving agents
Well-definedtness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Examples

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

A **toy problem**: To illustrate/exercise problem-solving methods

- It can be given a concise, exact description and hence is usable to compare the performance of algorithms

A **real-world problem**: To solve tasks people actually care about

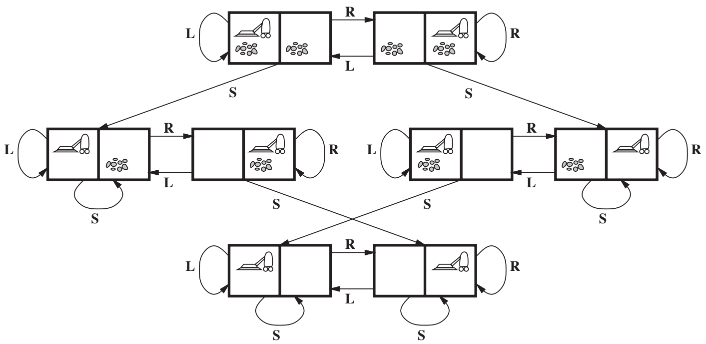
- Such problems tend not to have a single agreed-upon description, just a general flavour of their formulations

Examples - Toy problems

The **vacuum cleaner world** problem can be formulated as

- **States**: The state is determined by both the agent location and the dirt locations

The agent is in one of two locations, each of which might or might not contain dirt: Thus, there are $2 \times 2^2 = 8$ possible world states



Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedtness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

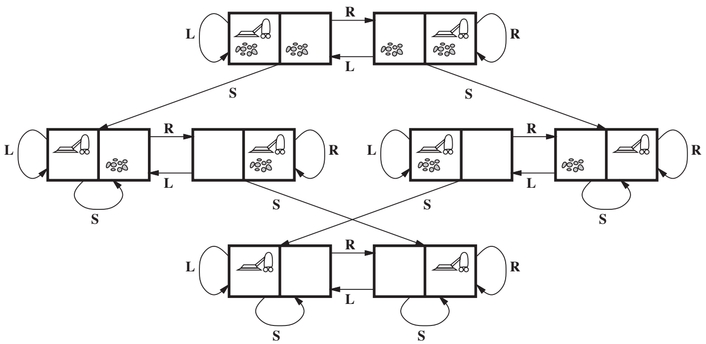
Examples - Toy problems (cont.)

Remark

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

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedtness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Examples - Toy problems (cont.)



- **Initial state**: Any state can be designated as the initial state
- **Actions**: Each state has three actions: `Left`, `Right`, `Suck`
- **Transition model**: Actions have the expected effects, except that moving `Left` in leftmost square, moving `Right` in rightmost square, and `Sucking` in clean square have no effect

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedtness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Examples - Toy problems (cont.)

- **Goal test**: This checks whether all the squares are clean
- **Path cost**: Each step costs 1, so the path cost is given by the number of steps in the path

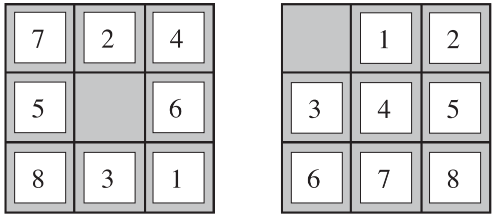
Compared with real world, the toy problem has discrete locations, discrete dirt, reliable cleaning, and it never gets any dirtier

- Some of these assumptions can be relaxed

Examples - Toy problems (cont.)

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

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



Start State Goal State

Solving by searching
UFC/DC
AI (CK0031)
2016.2

Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Examples - Toy problems (cont.)

The standard formulation is as follows:

- **States**: A state description specifies the 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**: The simplest formulation defines the actions as movements of the blank space `Left`, `Right`, `Up`, or `Down`[2]
- **Transition model**: Given state and action, it returns the resulting state[3]
- **Goal test**: This checks whether the state matches the 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 in figure, 5 and blank are switched

Solving by searching
UFC/DC
AI (CK0031)
2016.2

Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Examples - Toy problems (cont.)

What abstractions have we included here?

The actions are abstracted to their beginning and final states, ignoring the intermediate locations where the block is sliding

- We abstracted away some actions (such as shaking the board when pieces get stuck) and ruled out extracting the pieces with a knife and putting them back again

Remark

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

We avoid all the details of physical manipulations

Solving by searching
UFC/DC
AI (CK0031)
2016.2

Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
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 in AI

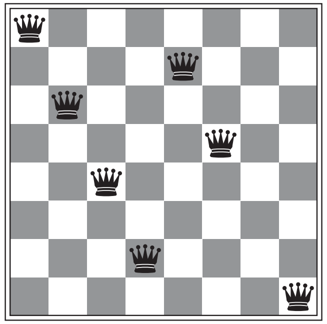
This family is known to be NP-complete, so we do not expect to find methods truly better in the worst case than search algorithms

- The 8-puzzle (our 3×3 board) has $9!/2 = 181{,}440$ reachable states and is easily solved
- The 15-puzzle (on a 4×4 board) has around 1.3 trillion states, and random instances can be solved optimally in a few milliseconds by the best search algorithms
- The 24-puzzle (on a 5×5 board) has around 10^{25} states, random instances take several hours to solve optimally

Examples - Toy problems (cont.)

The goal of the **8-queens** problem is to place eight queens on a chessboard such that no queen attacks any other

- A queen attacks any piece in the same row, column or diagonal
- Queen in the rightmost column is attacked by the queen at the top left



Efficient special-purpose algorithms exist for this problem and for the whole ***n*-queens** family, it is a useful test for search algorithms

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Examples - Toy problems (cont.)

There are two main kinds of formulation

- An **incremental formulation** involves operators that augment the state description, starting with an empty state;
- A **complete-state formulation** starts with all 8 queens on the board and moves them around

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

- Only the final state matters

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Examples - Toy problems (cont.)

An **incremental formulation** one might try is the following:

- **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: $64 \cdot 63 \cdot \cdots \cdot 57 \approx 1.8 \times 10^{14}$

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

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

We have seen how the **route-finding problem** is defined in terms of specified locations and transitions along links between them

Route-finding algorithms are used in a variety of applications

- Some (websites and in-car systems that provide driving directions) are extensions of the Romania example
- Others, (routing video streams in computer networks, military operations planning, and airline travel-planning systems) involve much more complex specifications

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
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 the current time. Furthermore, because the cost of an action (a flight segment) may depend on previous segments, their fare bases, and their status as domestic or international, the state must record extra info about these 'historical' aspects
- **Initial state**: This is specified by the user's query
- **Actions**: Take any flight from current location, in any seat class, leaving after the current time, leaving enough time for within-airport transfer if needed
- **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, ...

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

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) to the extent that these are justified by the cost and likelihood of failure of the original plan

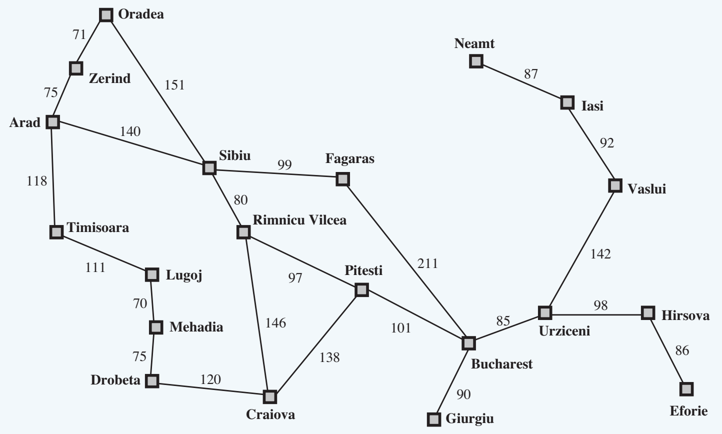
Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

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.)

As with route finding, the actions correspond to trips between adjacent cities,but the state space, however, is quite different

Each state must include not just the current location but also the set of cities the agent has already visited

- Initial state: $\texttt{In}(\texttt{Bucharest}), \texttt{Visited}(\{\texttt{Bucharest}\})$
- $\texttt{In}(\texttt{Vaslui}), \texttt{Visited}(\{\texttt{Bucharest}, \texttt{Urziceni}, \texttt{Vaslui}\})$
- The goal test would check whether the agent is in Bucharest and all 20 cities have been visited

Solving by searching
UFC/DC
AI (CK0031)
2016.2

Examples - Real-world problems (cont.)

The **traveling salesperson** problem (**TSP**) is a touring problem in which each city must be visited exactly once

- The aim is to find the shortest tour

The problem is known to be NP-hard, but an enormous effort has been expended to improve the capabilities of TSP algorithms

Not only planning trips for traveling salespersons, these algorithms have been used for tasks such as planning movements of automatic circuit-board drills and of stocking machines on shop floors

Solving by searching
UFC/DC
AI (CK0031)
2016.2

Examples - Real-world problems (cont.)

A **VLSI layout** problem requires positioning components and connections on a chip to minimize area, minimize circuit delays, minimize stray capacitances, and maximize manufacturing yield

The layout problem comes after the logical design phase and is usually split into two parts: 1) cell layout and 2) channel routing

- In cell layout, the 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 and so that there is room for connecting wires between cells

Solving by searching
UFC/DC
AI (CK0031)
2016.2

Examples - Real-world problems (cont.)

Robot navigation is a generalisation of the route-finding problem

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

For a circular robot moving on a flat surface, the space is two dimensional and when 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
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Searching for solutions

Solving by searching

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Searching for solutions

Having formulated some problems, we now need to solve them

A solution is an action sequence, so search algorithms work by considering various possible action sequences

The possible action sequences starting at the initial state form a **search tree** with the initial state at the root

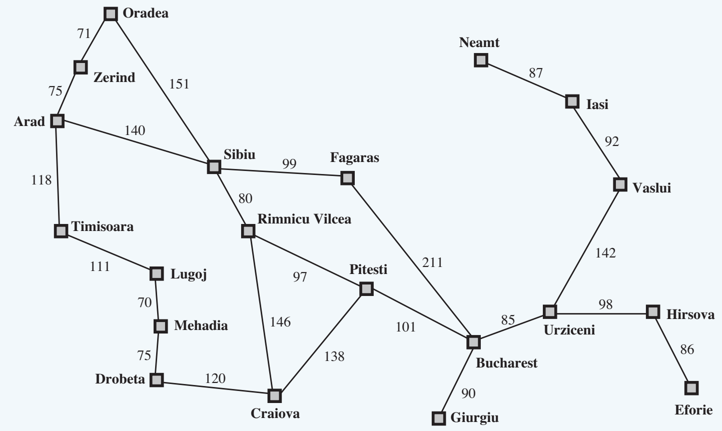
- the **branches are actions**
- the **nodes are states**

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

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, so we need to take actions

Definition

We do this by **expanding** the current state: By applying each legal action to the current state, thereby **generating** a new set of states

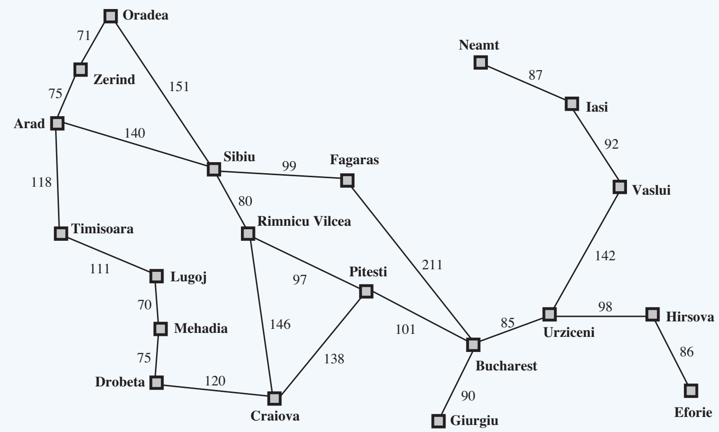
Solving by searching
UFC/DC
AI (CK0031)
2016.2

Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Example

In this case, we add three branches from **parent node** `In(Arad)`

- Leading to three new **child nodes**: `In(Sibiu)`, `In(Timisoara)`, and `In(Zerind)`

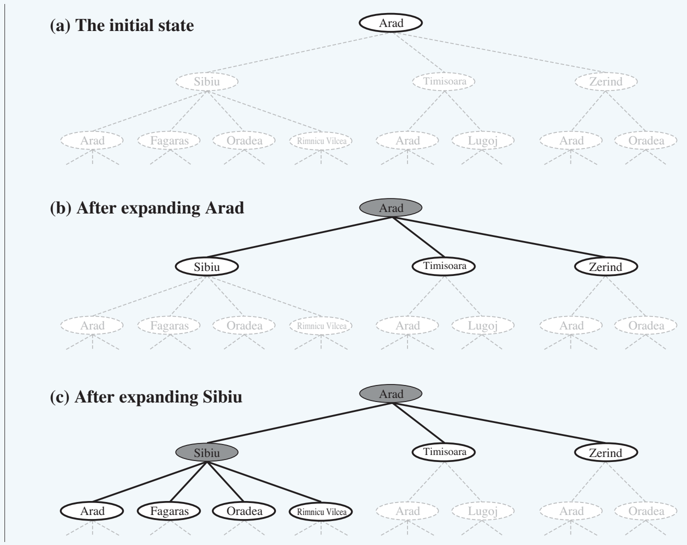


Now we must choose which of these options to consider further

Solving by searching
UFC/DC
AI (CK0031)
2016.2

Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Example



Solving by searching
UFC/DC
AI (CK0031)
2016.2

Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Searching for solutions (cont.)

This is the essence of search

Remark

- Follow up one option now and putting the others aside for later, in case the first choice does not lead to a solution

Searching for solutions (cont.)

Suppose we choose Sibiu first

1. We check to see whether it is a goal state (clearly, it is not)
2. We expand it to get `In(Arad)`, `In(Fagaras)`, `In(Oradea)`, and `In(RimnicuVilcea)`

We can pick any of these 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

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Searching for solutions (cont.)

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

The process of expanding nodes on the frontier continues until either a solution is found or there are no more states to expand

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

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
```

Search algorithms all share this basic structure

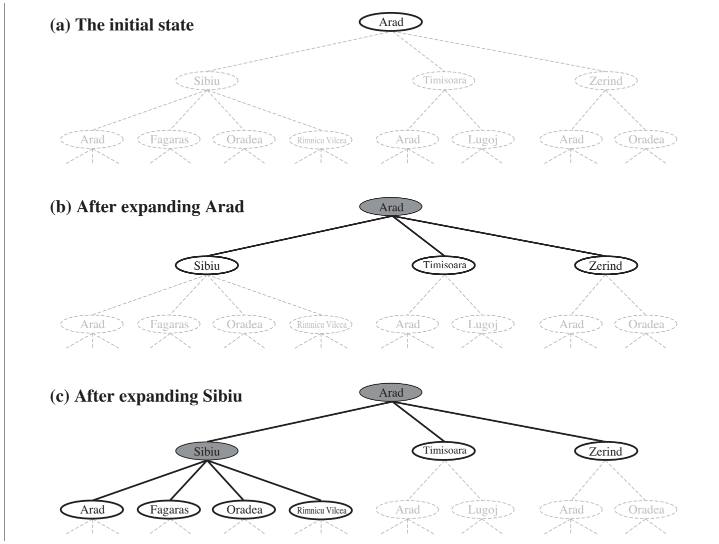
What varies mostly is how they choose which state to expand next

- This is the so-called **search strategy**

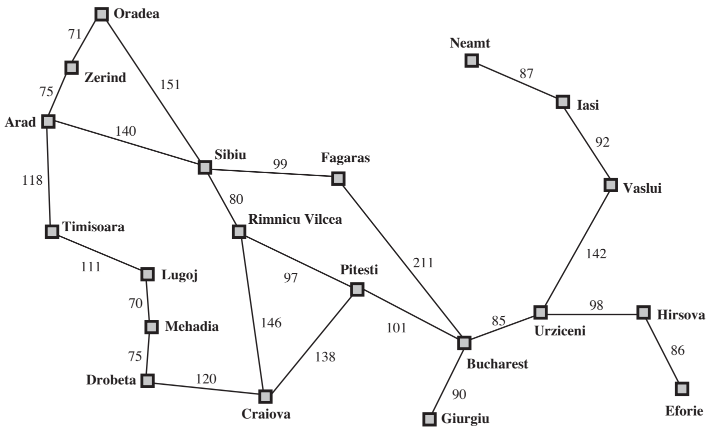
Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Searching for solutions (cont.)

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



- `In(Arad)` is a **repeated state** in the search tree
- In this case, it was generated by a **loopy path**



Considering such loopy paths means that the complete search tree is infinite, there is no limit to how often one can traverse a loop

- On the other hand, the state space has only 20 states

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Searching for solutions (cont.)

Loops can cause certain algorithms to fail

- Otherwise solvable problems can be made unsolvable

No need for loopy paths, more than obvious

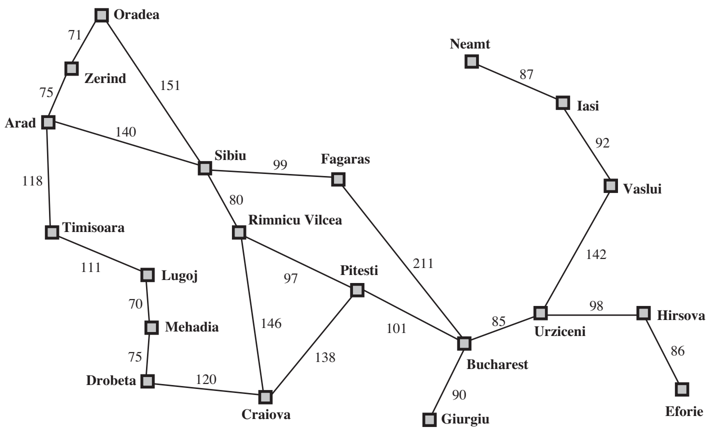
- Path costs are additive and step costs nonnegative
- A loopy path to any state is never better than the same path with the loop removed

Loops are special cases of the general concept of **redundant paths** (there is more than one way to get from one state to another)

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

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

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Searching for solutions (cont.)

Remark

If you are concerned about reaching the goal, there's never any reason to keep more than one path to any given state

- Any goal state that is reachable by extending one path is also reachable by extending the other

In some cases, it is possible to define the problem itself so as to eliminate redundant paths

- If we formulate the 8-queens problem so that a queen can be placed in any column, then each state with n queens can
  be reached by $n!$ different paths
- If we reformulate the problem so that each new queen is placed in the leftmost empty column, then each state can be reached only through one path

Searching for solutions (cont.)

In other cases, redundant paths are unavoidable and this includes all problems where the actions are reversible

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

Route-finding on a rectangular grid (will discuss it soon) is a particularly important example in computer games

- In such grid, each state has four successors, so a search tree of depth d that includes repeated states has 4^d leaves, but there are about $2d^2$ distinct states within d steps of any given state

For $d = 20$, about a trillion nodes but about 800 distinct states

Remark

Redundant paths can cause a tractable problem to turn intractable

- This is true even for algorithms that avoid infinite loops

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

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 called the **explored set** or **closed list**, which remembers every expanded node
- Newly generated nodes that match previously generated nodes, ones in the explored set or the frontier, can be discarded instead of being added to the frontier

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Searching for solutions (cont.)

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

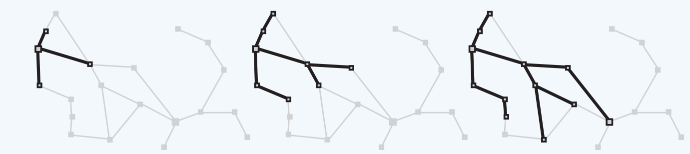
```
function GRAPH-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  initialize the explored set to be empty
  loop do
     if the frontier is empty then return failure
     choose a leaf node and remove it from the frontier
     if the node 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
```

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Searching for solutions (cont.)

The **GRAPH-SEARCH** algorithm contains at most one copy of each state, so 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

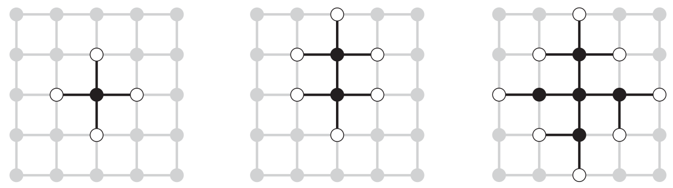
Northernmost city (Oradea) has become a dead end (3rd stage)

- Both of its successors are already explored via other paths

Searching for solutions (cont.)

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

- Every path from the initial state to an unexplored state has to pass through a state in the frontier



(a) (b) (c)

The frontier (white nodes) always separates explored region of the state-space (black nodes) from unexplored region (gray nodes)

- 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
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Search algorithms

Searching for solutions

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Search algorithms

Search algorithms require a data structure to keep track of the search tree that is being constructed

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

- n.STATE: the state in the state space to which the node corresponds;
- n.PARENT: the node in the search tree that generated this node;
- n.ACTION: the action that was applied to the parent to generate the node;
- n.PATH-COST: the cost, $g(n)$, of the path from the initial state to the node, as indicated by the parent pointers

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Search algorithms (cont.)

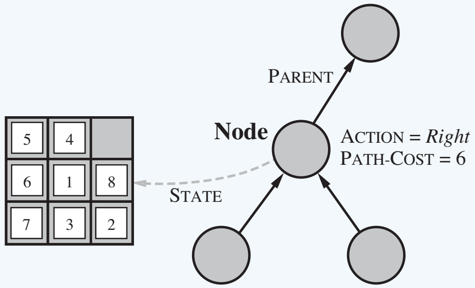
Given the components for a parent node, compute the necessary components for a child node using function **CHILD-NODE**

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 various bookkeeping fields
- Arrows point from child to parent

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
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 bookkeeping data structure (it is 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

- Furthermore, two different nodes can contain the same world state if that state is generated via two different search paths

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Search algorithms (cont.)

The frontier needs to be stored in such 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 are as follows:

- `EMPTY?(queue)` returns true only if there are no more elements in the queue
- `POP(queue)` removes the first element of the queue and returns it
- `INSERT(element,queue)` inserts an element and returns the resulting queue

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Search algorithms (cont.)

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

Three common variants are

- the **first-in, first-out** or **FIFO queue**, which pops the oldest element of the queue;
- the **last-in, first-out** or **LIFO queue** or **stack**, which pops the newest element of the queue;
- the **priority queue**, which pops the element of the queue with the highest priority according to some ordering function

Measuring performance

Searching for solutions

Solving by searching

UFC/DC
AI (CK0031)
2016.2

Problem solving
Problem-solving agents
Well-definedtness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Measuring performance

Before we get into the design of a specific search algorithms, we consider the criteria that might be used to choose among them

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?

Solving by searching

UFC/DC
AI (CK0031)
2016.2

Problem solving
Problem-solving agents
Well-definedtness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Measuring performance (cont.)

Time and space complexity are always considered with respect to some measure of the 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

- The map of Romania is an example of this

Solving by searching

UFC/DC
AI (CK0031)
2016.2

Problem solving
Problem-solving agents
Well-definedtness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Measuring performance (cont.)

Remark

In AI, the graph is often represented implicitly by initial state, actions, and transition model and 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

Measuring performance (cont.)

Time is often measured as number of nodes generated during search, space 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

Solving by searching
UFC/DC
AI (CK0031)
2016.2

Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Measuring performance (cont.)

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

- **search cost**, it typically depends on the time complexity but can also include a term for memory usage
- **total cost**, which combines the search cost and the path cost of the solution found

Example

For the problem of finding a route from Arad to Bucharest, the search cost is the amount of time taken by the search and the solution cost is the total length of the path in kilometres

- To compute the total cost, we add milliseconds and kilometres

Solving by searching
UFC/DC
AI (CK0031)
2016.2

Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Measuring performance (cont.)

No direct link between them: Reasonable to convert kilometres into milliseconds (time is important here, got a flight to take)

- by using an estimate of the car's average speed

This enables the agent to find an optimal tradeoff point at which further work to find a shorter path becomes counterproductive

- A more general problem, tradeoffs between different goods

Solving by searching
UFC/DC
AI (CK0031)
2016.2

Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Uninformed search

Solving by searching

Uninformed search

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

- The term means that the strategies have no additional info about states beyond that provided in the problem definition

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

Search strategies are distinguished by the node expansion order

Strategies that know whether a non-goal state is 'more promising' than another are called **informed**/**heuristic search** strategies

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

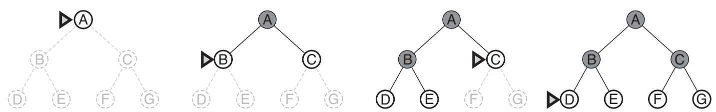
Breadth-first search
Uninformed search

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Breadth-first search

Breadth-first search: Root node is expanded first, all successors of root node are then expanded, then their successors, 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

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Breadth-first search (cont.)

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

Breadth-first search (cont.)

Remark

There is one slight tweak on the general graph-search algo, which is that the goal test is applied to each node when it is generated

- rather than when it is selected for expansion

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Breadth-first search (cont.)

The algorithm, following the template for graph search, discards any new path to a state already in the frontier or explored set

- It is easy to see that any such path must be at least as deep as the one already found
- Breadth-first search always has the shallowest path to every node on the frontier

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

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)
```

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
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 (branching factor b need be finite)
- As a goal node is generated, we know it is the shallowest goal node (all shallower nodes must have been generated already and failed the goal test)

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 nondecreasing function of the depth of the node

Most common such scenario: All actions have the same cost

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
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, for a total of b^2 at level two, each of these generates b more nodes, yielding 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 $b + b^2 + b^3 + \cdots + b^d = \mathcal{O}(b^d)$

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
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, the whole layer of nodes at depth d would be expanded before the goal was detected and the time complexity would be $\mathcal{O}(b^{d+1})$

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Breadth-first search (cont.)

What about space complexity?

For any kind of graph search, which 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 in particular, every node generated remains in memory
- There will be $\mathcal{O}(b^{d-1})$ nodes in the explored set and $\mathcal{O}(b^d)$ 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, and in a state space with redundant paths, switching could cost a lot of time

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

For various values of the solution depth d, the time and memory required for a breadth-first search with branching factor $b = 10$

- The table assumes that 1 million nodes can be generated per second, and that a node requires 1000 bytes of storage
- 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, but 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

Uniform-cost search

Uninformed search

Uniform-cost search

When all step costs are equal, breadth-first search is optimal because it always expands the shallowest unexpanded node

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

Instead of expanding the shallowest node, **uniform-cost search** expands the node n with the lowest path cost $g(n)$

- By storing the frontier as a priority queue ordered by g

Solving by searching
UFC/DC
AI (CK0031)
2016.2

Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

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
```

Solving by searching
UFC/DC
AI (CK0031)
2016.2

Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

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, so it should combine the capabilities of a priority queue and a hash table

Solving by searching
UFC/DC
AI (CK0031)
2016.2

Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Uniform-cost search (cont.)

In addition to the ordering of the queue by path cost, there are two other significant differences from breadth-first search

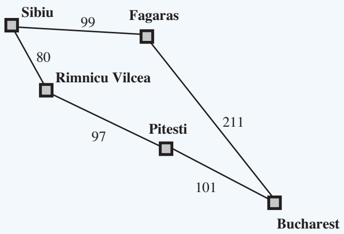
The first is that the goal test is applied to a node when it is selected for expansion rather than when it is first generated

- The reason is that the first goal node that is generated may be on a suboptimal path

The second difference is that a test is added in case a better path is found to a node currently on the frontier

Uniform-cost search (cont.)

Example



From Sibiu to Bucharest

The successors of Sibiu are

- Rimnicu Vilcea and Fagaras
- With 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

- choosing Pitesti for expansion and adding a second path to Bucharest with cost $80 + 97 + 101 = 278$

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
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

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Uniform-cost search (cont.)

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

First, we observe that whenever uniform-cost search selects a node n for expansion, the optimal path to that node has been found

- Were this not the case, there would have to be another frontier node n' on the optimal path from the start node to n, and by definition, n would have lower g-cost than n and would have been selected first

Nonnegative step costs, paths never get shorter as nodes are added

Uniform-cost search expands nodes in order of their optimal path cost

- Hence, the first goal node selected for expansion must be the optimal solution

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Uniform-cost search (cont.)

Uniform-cost search does not care about the number of steps a path has, but only about their total cost

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, so its complexity is not easily characterised in terms of b and d

- Assume that every action costs at least ε
- The algorithm's worst-case time and space complexity is $\mathcal{O}(b^{1+\lfloor C^*/\varepsilon \rfloor})$, which 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}

Solving by searching
UFC/DC
AI (CK0031)
2016.2

Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Uniform-cost search (cont.)

Remark

When all step costs are the same, uniform-cost search is similar to breadth-first search, except that the latter stops as soon as it generates a goal, whereas 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 by expanding nodes at depth d unnecessarily

Solving by searching
UFC/DC
AI (CK0031)
2016.2

Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Depth-first search

Uninformed search

Solving by searching
UFC/DC
AI (CK0031)
2016.2

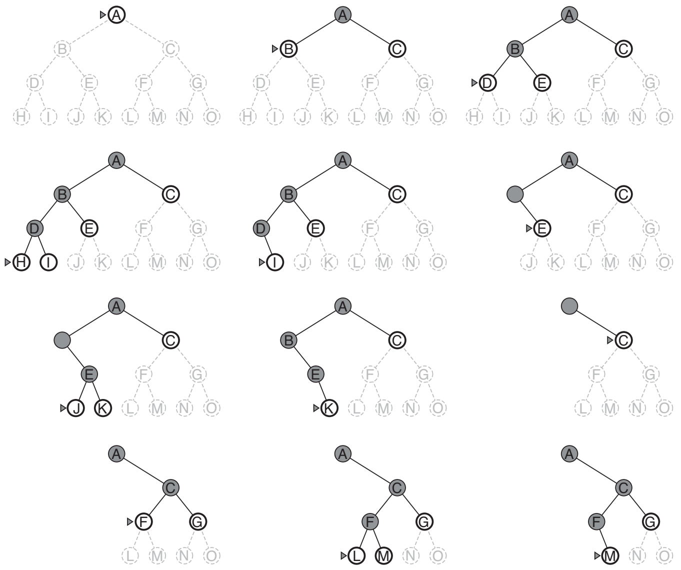
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
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, so then the search 'backs up' to the next deepest node that still has unexplored successors

Depth-first search (cont.)



Solving by searching

UFC/DC
AI (CK0031)
2016.2

Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

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)

Solving by searching

UFC/DC
AI (CK0031)
2016.2

Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

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 that 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

Solving by searching

UFC/DC
AI (CK0031)
2016.2

Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

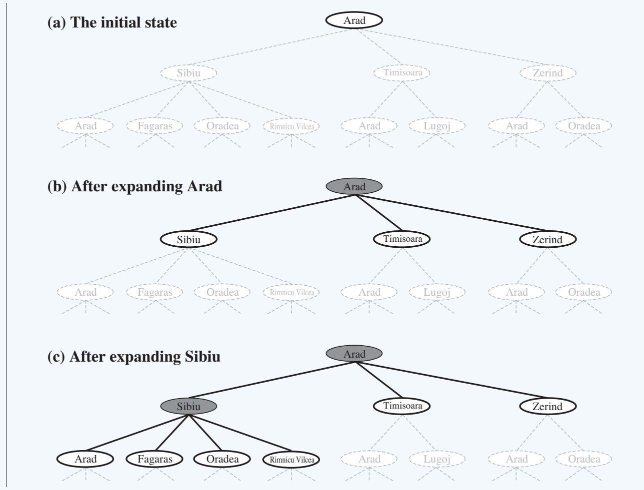
Depth-first search (cont.)

The properties of depth-first search depend strongly 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

Example

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



Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedtness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
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

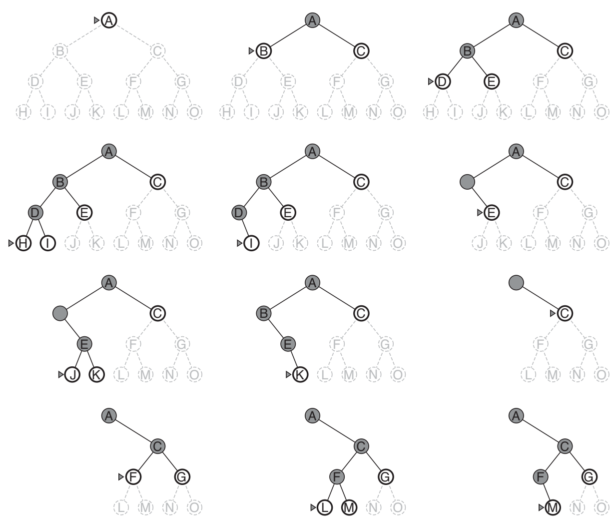
This avoids infinite loops in finite state spaces but does not avoid the proliferation of redundant paths

In infinite state spaces, both versions fail if an infinite non-goal path is encountered

For similar reasons, both versions are non-optimal

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedtness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

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)

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedtness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Depth-first search (cont.)

The time complexity of depth-first graph search is bounded by the size of the state space (which may be infinite)

Depth-first tree search, on the other hand, may generate all of the $\mathcal{O}(b^m)$ nodes in the search tree, where m is the maximum depth of any node; this can be much greater than the size of the state space

Remark

Note that m itself can be much larger than d (the depth of the shallowest solution) and it is infinite if the tree is unbounded

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

For a graph search, there is no advantage, but a depth-first tree search needs to store only a single path from root to leaf node

- along with the remaining unexpanded sibling nodes for each node on the path

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

- For a state space with branching factor b and maximum depth m, depth-first search requires storage of only $\mathcal{O}(b^m)$ nodes

Example

Assuming that nodes at the same depth as the goal node have no successors, we find that depth-first search would require 156Kbytes instead of 10Exabytes at depth $d = 16$, 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

- In backtracking, only one successor is generated at a time rather than all successors; each partially expanded node remembers which successor to generate next
- In this way, only $\mathcal{O}(m)$ memory is needed rather than $\mathcal{O}(b^m)$

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

- For this to work, 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

Solving by searching
UFC/DC
AI (CK0031)
2016.2

Depth-limited search

The failure of depth-first search in infinite state spaces can be alleviated by supplying 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 also introduces an additional source of incompleteness if we choose $l < d$, as the shallowest goal is beyond the depth limit

- (likely when d is unknown)

Solving by searching
UFC/DC
AI (CK0031)
2016.2

Depth-limited search (cont.)

Depth-limited search will also be non-optimal if we choose $l > d$, as its time complexity is $\mathcal{O}(b^l)$ and its space complexity is $\mathcal{O}(bl)$

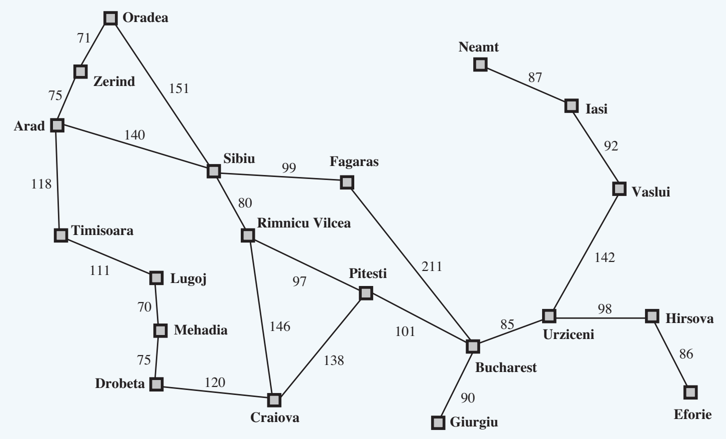
Depth-first search can be viewed as a special case of depth-limited search with $l = \infty$

Solving by searching
UFC/DC
AI (CK0031)
2016.2

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

Example

On 20 cities, therefore we know that if there is a solution, it must be of length 19 at the longest, so $l = 19$ is a possible choice



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

Depth-limited search (cont.)

Definition

This number, **diameter of the state space**, gives us a better depth limit, which leads to a more efficient depth-limited search

Remark

For most problems, however, we will not know a good depth limit until we have solved the problem

Solving by searching
UFC/DC
AI (CK0031)
2016.2

Depth-limited search (cont.)

Depth-limited search can be implemented as a modification to the general tree or graph-search algorithm, 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
```

Remark

Notice that depth-limited search can terminate with two kinds of failure: i) the standard failure value indicates no solution, and ii) the cutoff value indicates no solution within the depth limit

Solving by searching
UFC/DC
AI (CK0031)
2016.2

Iterative deepening depth-first search

Uninformed search

Solving by searching
UFC/DC
AI (CK0031)
2016.2

Iterative deepening depth-first search

Iterative deepening search (or iterative deepening depth-first search) is a general strategy, often used in combination with depth-first tree search, that 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

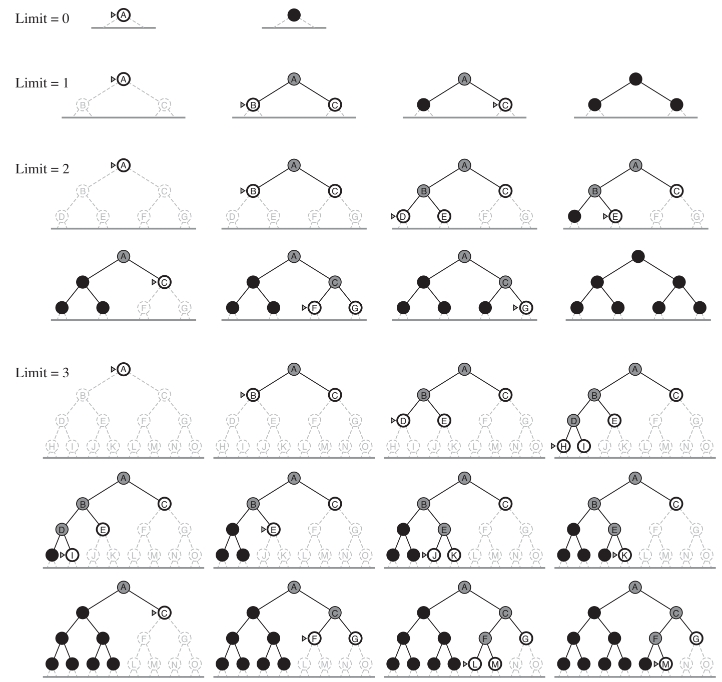
This occurs when depth limit reaches d,
the depth of the shallowest goal node

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
  for depth = 0 to ∞ do
     result ← DEPTH-LIMITED-SEARCH(problem, depth)
     if result ≠ cutoff then return result
```

Iterative deepening depth-first (cont.)

Four iterations of ITERATIVE-DEEPENING-SEARCH on a binary search tree: The solution is found on the fourth iteration

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics



Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Iterative deepening depth-first (cont.)

Iterative deepening combines the benefits of depth-first and breadth-first search

- Like depth-first search, its memory requirements are modest

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

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

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
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

In a search tree with the same (or 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

Iterative deepening depth-first (cont.)

In an iterative deepening search, nodes on bottom level (depth d) are generated once, those on next-to-bottom level are generated twice, and so on, up to the children of the root, generated d times

So 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

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

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedtness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
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 consumed, and then run iterative deepening from all the nodes in the frontier

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedtness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Iterative deepening depth-first (cont.)

Remark

In general, iterative deepening is the preferred uninformed search, when search space is large and the solution depth is unknown

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedtness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Bidirectional search

Uninformed search

Bidirectional search

The idea behind is to run two parallel searches

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

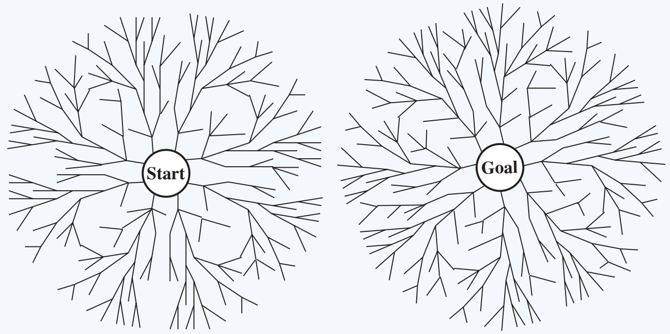
hoping that the two searches meet in the middle

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedtness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
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

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedtness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Bidirectional search (cont.)

Bidirectional search is implemented by replacing the goal test with a check to see whether the frontiers of the two searches intersect

- if they do, a solution has been found

It is important to realise that 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 not another short-cut across the gap

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedtness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Bidirectional search (cont.)

The check can be done when each node is generated or selected for expansion and, with a hash table, will take constant time

Example

If a problem has solution depth $d = 6$, and 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, compared with 1111110 for a standard breadth-first search

Thus, the time complexity of bidirectional search using breadth-first searches in both directions is $\mathcal{O}(b^{d/2})$

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

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Bidirectional search (cont.)

The reduction in time complexity makes bidirectional search attractive, but 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

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Bidirectional search (cont.)

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

Example

For 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

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Comparison

Uninformed search

Uninformed searches, comparison

We compare tree-search strategies using four evaluation criteria:

- The main differences are that depth-first search is complete for finite state spaces and that space and time complexities are bounded by the size of the state space

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

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
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Informed searches
Solving by searching

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Informed searches

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

- 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**, which is an instance of the general **TREE-** or **GRAPH-SEARCH** algo

- 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

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
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

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Informed searches (cont.)

Most best-first algos use as a component of f a **heuristic function**

- $h(n)$: Estimated cost of cheapest path from state at node n to a goal state

Note that $h(n)$ takes a node as input, but unlike $g(n)$, it depends only on the state at that node

Example

In Romania, one might estimate the cost of the cheapest path from Arad to Bucharest via the straight-line distance

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Informed searches (cont.)

Heuristic functions are the most common form in which extra knowledge of the problem is imparted to the search algorithm

- We shall study heuristics in more depth

We begin by considering them to be arbitrary, nonnegative, problem-specific functions, with one single constraint

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

Greedy best-first search

Informed search

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedtness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Greedy best-first search

Greedy best-first search tries to expand the node that is closest to goal, because this is likely to lead to a solution quickly

- Thus, it evaluates nodes by using just the heuristic function

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

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedtness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

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 to know the straight-line distances to Bucharest: For example, $h_{SLD}(\texttt{In}(\texttt{Arad})) = 366$

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

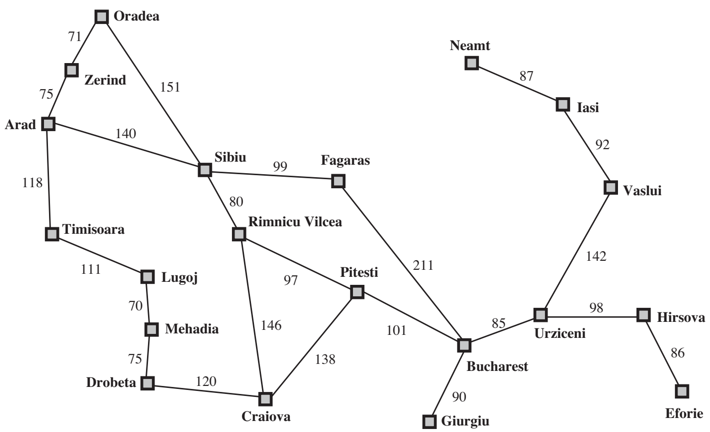
Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedtness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Greedy best-first search (cont.)

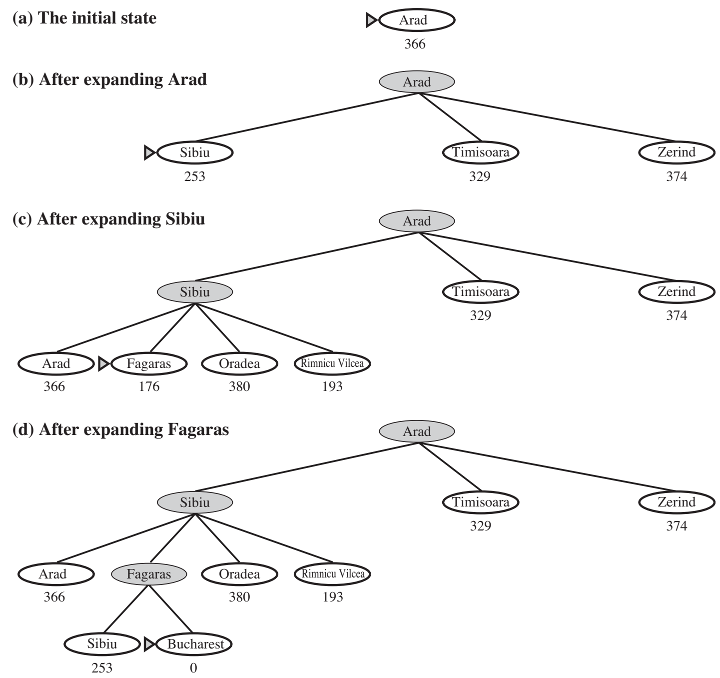
Values of h_{SLD} cannot be computed from the problem description

- Moreover, it takes a certain amount of experience to know that h_{SLD} is correlated with actual road distances
- It is, therefore, a useful heuristic

Greedy best-first search (cont.)



- The first node to be expanded from Arad is Sibiu, as it is 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



Greedy best-first search (cont.)

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

- Hence, 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, while graph search version is complete in finite spaces, but not in infinite ones

Example

Consider the problem of getting from Iasi to Fagaras

- The heuristic suggests that Neamt be expanded first, because it is closest to Fagaras, but it is a dead end.
- The 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, as expanding Neamt puts Iasi back into the frontier, Iasi is closer to Fagaras than Vaslui is, and so 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 $\mathcal{O}(b^m)$, where m is the maximum depth of the search space

With a good heuristic function, complexity can be reduced

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

A^* search
Informed search

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

A^* search

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

- It evaluates nodes by combining $g(n)$, the cost to reach the node, and $h(n)$, the cost to get from the node to the goal

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

Since $g(n)$ gives the path cost from start node to node n, and $h(n)$ is the estimated cost of the cheapest path from n to goal,

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

Thus, if we are trying to find the cheapest solution, a reasonable thing to try first is the node with the lowest value of $g(n) + h(n)$

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

A^* search (cont.)

It turns out that this strategy is more than just reasonable

- Provided that the heuristic function $h(n)$ satisfies certain conditions, A^* search is both complete and optimal

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

Because $g(n)$ is the actual cost to reach n along the current path, and $f(n) = g(n) + h(n)$, we have as an immediate consequence

- $f(n)$ never overestimates the true cost of a solution along the current path through n

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedtness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

A^*: Conditions for optimality (cont.)

Remark

Admissible heuristics are by nature optimistic, as they think that the cost of solving the problem is less than it actually is

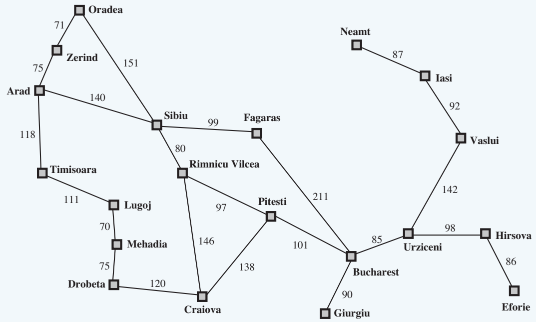
- An obvious example of an admissible heuristic is the straight-line distance h_{SLD} for getting to Bucharest
- Straight-line distance is admissible because the shortest path between any two points is a straight line
- The straight line cannot be an overestimate

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

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedtness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Example

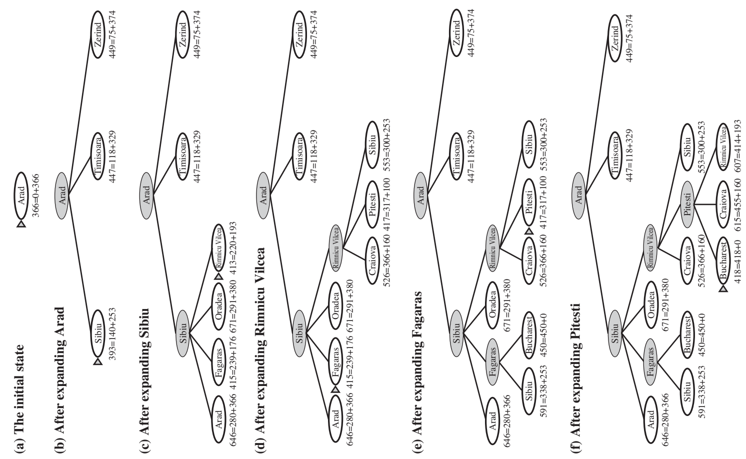
Values of g are computed from step costs



The values of h_{SLD} are calculated as

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

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedtness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics



A^*: Conditions for optimality (cont.)

Bucharest first appears on the frontier at step (e), but 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

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedtness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

A^*: Conditions for optimality (cont.)

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

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

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedtness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

A^*: Conditions for optimality (cont.)

For an admissible heuristic, the inequality makes perfect sense

- if there were a route from n to G_n via n that was cheaper than $h(n)$, that would violate the property that $h(n)$ is a lower bound on the cost to reach G_n

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedtness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

A^*: Optimality

A^* has the following 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

The first step is to establish the following:

- If $h(n)$ is consistent, then values of $f(n)$ along any path are non-decreasing

The proof follows directly from the definition of consistency

Proof

Suppose n' is a successor of n then $g(n') = g(n) + c(n, a, n')$ for some action a, and we have

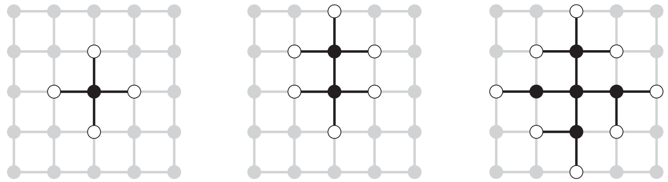
$$\begin{aligned} f(n') &= g(n') + h(n') \\ &= g(n) + c(n, a, n') + h(n') \geq g(n) + h(h) \\ &= f(n) \end{aligned}$$

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

A^*: Optimality (cont.)

The next step is to prove that whenever A^* selects a node n for expansion, the optimal path to that node has been found

- Were this not the case, there would have to be another frontier node n' on the optimal path from start node to n, by the graph separation property



(a) (b) (c)

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

A^*: Optimality (cont.)

It follows that the sequence of nodes expanded by A^* using GRAPH-SEARCH is in non-decreasing order of $f(n)$

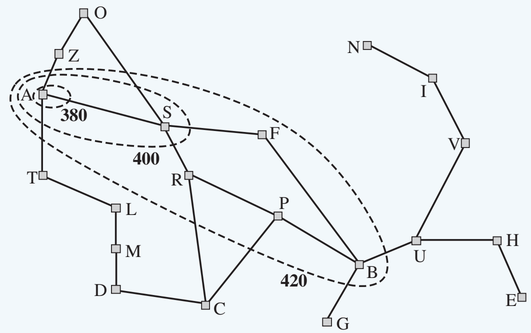
- The first goal node selected for expansion must be an optimal solution because f is the true cost for goal nodes (with $h = 0$)
- All later goal nodes will be at least as expensive

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

A^*: Optimality (cont.)

Because f-costs are non-decreasing along any path, we can draw **contours** in the state space, like in a topographic map

Example



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

A^*: Optimality (cont.)

With uniform-cost search (A^* search using $h(n) = 0$), the bands will be 'circular' around the start state

With accurate heuristics, the bands will stretch toward the goal state and become more narrowly focused around the optimal path

If C^* is the cost of the optimal solution path, then we can say:

- 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 that there be only finitely many nodes with cost less than or equal to C^*, a condition that is true if all step costs exceed some finite ε and if b is finite

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

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**

Because h_{SLD} is admissible, the algorithm can safely ignore this subtree while still guaranteeing optimality

Pruning, or eliminating possibilities from consideration without having to examine them, is an important concept for AI

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

A^*: Optimality (cont.)

Among optimal algorithms of this type (algorithms that extend search paths from root and use the same heuristic information) A^* is optimally efficient for any given consistent heuristic

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

This is because any algorithm that does not expand all nodes with $f(n) < C^*$ runs the risk of missing the optimal solution

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

A^*: Optimality (cont.)

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

- Unfortunately, it does not mean that A^* is the answer to all our searching needs

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

- For problems with constant step costs, the growth in runtime as a function of the optimal solution depth d is analysed in terms of **absolute error** or **relative error** of the heuristic

Definition

The absolute error is defined as $\Delta \equiv (h^* - h)$, where h^* is the actual cost of getting from root to goal, and 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** and is essentially a **tree** with **reversible actions**

Example

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

In this case, the time complexity of A^* is exponential in maximum absolute error, that is $\mathcal{O}(b^{\Delta})$

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: $\mathcal{O}(b^{\varepsilon d}) = \mathcal{O}((b^{\varepsilon})^d)$, so the effective branching factor (defined soon) is b

A^*: Optimality (cont.)

When the state space has many goal states (near-optimal ones, particularly) the search can be led astray from optimal path

- There is an extra cost proportional to the number of goals whose 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 with some subset has been cleaned and all of them are on an optimal solution path

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

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedtness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

A^*: Optimality (cont.)

The complexity of A^* often makes it impractical

Some variants can find suboptimal solutions and it is possible to design heuristics that are more accurate but not strictly admissible

- The use of a good heuristic still provides big savings compared to the use of an uninformed search

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedtness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

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), so A^* usually runs out of space long 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, at cost in execution time

We discuss these next

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedtness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Memory-bounded search

Informed search

Memory-bounded search

The simplest way to reduce memory requirements for A^* is to adapt the idea of iterative deepening to the heuristic search context, resulting in the **iterative-deepening A^*** (**IDA^***) algo

The big difference between IDA^* and standard iterative deepening is that 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

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedtness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Memory-bounded search (cont.)

IDA^* is practical for problems with unit step costs and avoids the overhead associated with keeping a sorted queue of nodes

Unfortunately, IDA^* suffers from the same difficulties with real valued costs as does the iterative version of uniform-cost search

- We briefly examine two other memory-bounded algorithms

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedtness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Memory-bounded search (cont.)

Recursive best-first search (**RBFS**) is a recursive algorithm that attempts to mimic standard best-first search, but using 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 ← [ ]
  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 ← max(s.g + s.h, node.f))
  loop do
      best ← the lowest f-value node in successors
      if best.f > f_limit then return failure, best.f
      alternative ← the second-lowest f-value among successors
      result, best.f ← RBFS(problem, best, min(f_limit, alternative))
      if result ≠ failure then return result
```

By structure, the algo is similar to recursive depth-first search, but rather than continuing 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

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedtness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

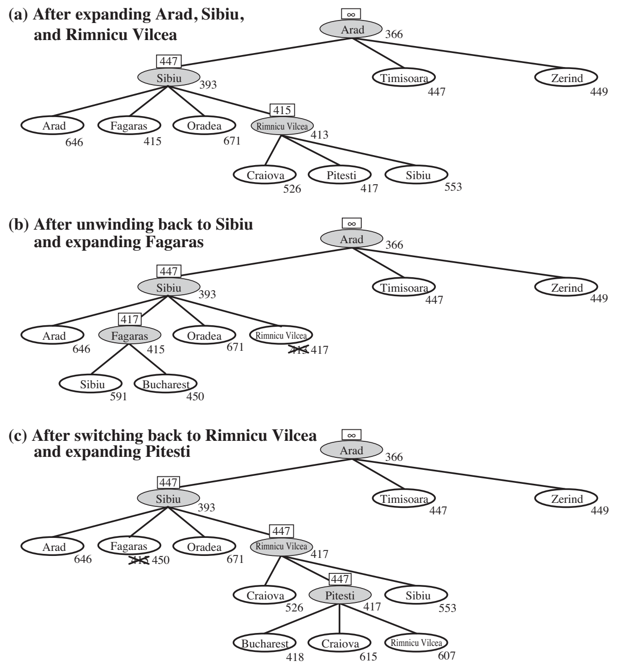
Memory-bounded search (cont.)

If current node exceeds f_limit then the recursion unwinds back to the alternative path

As recursion unwinds, the f-value of each node along the path is replaced with a **backed-up value** (the best f-value of its children)

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

- It can decide whether it is worth re-expanding the subtree at some later time



Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

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

(a) Path via Rimnicu Vilcea is followed until current best leaf (Pitesti) is worse than best alternative path (Fagaras)

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

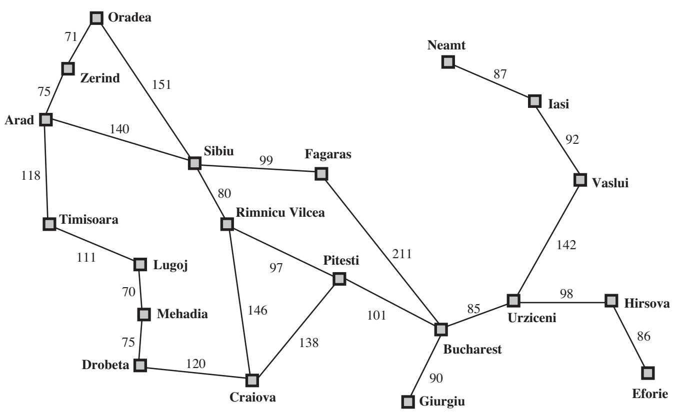
Then Fagaras is expanded, revealing a best leaf value of 450

(c) The recursion unwinds and the best leaf value of the forgotten subtree (450) is backed up to Fagaras
Then Rimnicu Vilcea is expanded

Because the best alternative path (through Timisoara) costs at least 447, the expansion continues to Bucharest

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Memory-bounded search (cont.)



Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
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' and tries Fagaras, and then changes its mind back again

These mind changes occur because every time the 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

Memory-bounded search (cont.)

Remark

Each mind change corresponds to an iteration of IDA^* and could require many re-expansions of forgotten nodes to recreate the best path and extend it one more node

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

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, but its time complexity is rather difficult to characterise

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

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Memory-bounded search (cont.)

IDA^* and RBFS suffer from using too little memory

Between iterations, IDA^* retains only a single number:

- the current f-cost limit

RBFS retains more information, but uses linear space:

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

Remark

Because they forget most of what they have done, both IDA^* and RBFS may end up re-expanding the same states many times over

Also, they suffer the potentially exponential increase in complexity associated with redundant paths in graphs

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Memory-bounded search (cont.)

It seems sensible to use all available memory

Two algorithms that do this are

- **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 the search tree without dropping an old one
- SMA^* always drops the worst leaf node, the one with the highest f-value
- Like RBFS, SMA^* backs up the value of the forgotten node to its parent

In this way, the ancestor of a forgotten subtree knows the quality of best path in that subtree

Memory-bounded search (cont.)

With this info, SMA^* regenerates the subtree only when all other paths have been shown to look worse than the forgotten path

- So, if all descendants of a node n are forgotten, then we will not know which way to go from n, but we will still have an idea of how worthwhile it is to go anywhere from n

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

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

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
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

In practical terms, SMA^* is a robust choice for finding optimal solutions, particularly when the state space is a graph

- step costs are not uniform, and node generation is expensive compared to the overhead of keeping frontier and explored set

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

- That is to say, memory limitations can make a problem intractable from the point of view of computation time

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

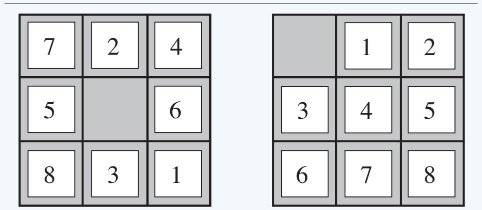
Heuristic functions

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

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
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

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
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

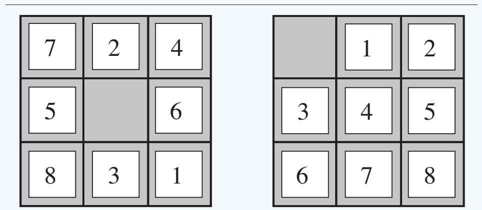
If we want to find the shortest solutions by using A^*, we need a heuristic function that never overestimates the number of steps

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

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Heuristic function (cont.)

Example



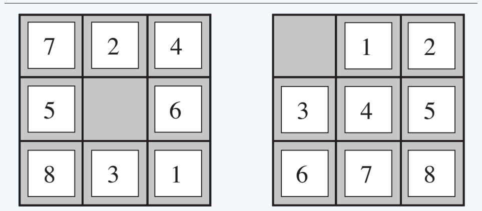
Start State Goal State

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 as it is clear that 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 because all any move can do is move one tile one step closer to the goal
- Tiles 1 to 8 in start state give a distance of $h_2 = 3 + 1 + 2 + 2 + 2 + 3 + 3 + 2 = 18$

Solving by searching
UFC/DC
AI (CK0031)
2016.2

Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Heuristic function (cont.)

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

Solving by searching
UFC/DC
AI (CK0031)
2016.2

Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Accuracy and performance

Heuristic function

Solving by searching
UFC/DC
AI (CK0031)
2016.2

Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
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 $b^* = 1.92$

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

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

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

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

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 $h_2(n) \geq h_1(n)$

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

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

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
AI (CK0031)
2016.2

Admissible heuristics from relaxed problems

Heuristic function

Solving by searching

UFC/DC
AI (CK0031)
2016.2

Admissible heuristics from relaxed problems

Both h_1 (misplaced tiles) and h_2 (Manhattan distance) are fairly good heuristics for the 8-puzzle and 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?

Solving by searching

UFC/DC
AI (CK0031)
2016.2

Admissible heuristics from relaxed problems

For the 8-puzzle, h_1 and h_2 are estimates of the remaining path length, but they are also 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 exact 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 exact number of steps in the shortest solution

Admissible heuristics from relaxed problems (cont.)

Definition

Problems with fewer restrictions on actions: **Relaxed problems**

- 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

- Though the relaxed problem may have better solutions, if the added edges provide short cuts

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

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**

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Admissible heuristics from subproblems

Heuristic function

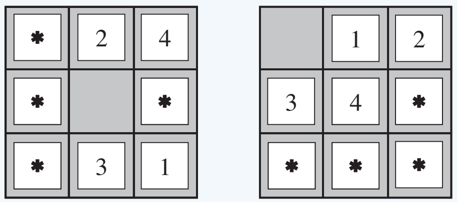
Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Admissible heuristics from subproblems

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

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 Manhattan distance

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

Example

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 simply by looking up the corresponding subproblem configuration in the database

Solving by searching
UFC/DC
AI (CK0031)
2016.2

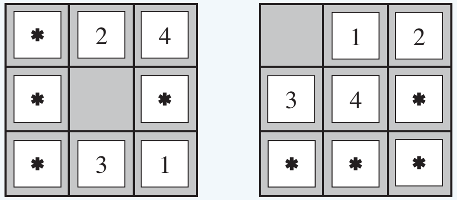
Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
Uninformed search
Breadth-first search
Uniform-cost search
Depth-first search
Depth-limited search
Iterative deepening depth-first search
Bidirectional search
Informed searches
Greedy best-first search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Admissible heuristics from subproblems (cont.)

The database itself is constructed by searching back from the goal and recording the cost of each new pattern encountered

- the expense of this search is amortised over many subsequent problem instances

Example



Start State Goal State

The choice of $1 - 2 - 3 - 4$ is fairly arbitrary, as we can construct databases for $5 - 6 - 7 - 8$, $2 - 4 - 6 - 8$, etc.

Solving by searching
UFC/DC
AI (CK0031)
2016.2

Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Admissible heuristics from subproblems (cont.)

Each database yields an admissible heuristic, and these heuristics can be combined, by taking the maximum value

- A combined heuristic of this kind is more accurate than the Manhattan distance

Remark

The number of nodes generated when solving random 15-puzzles can be reduced by a factor of 1K

Solving by searching
UFC/DC
AI (CK0031)
2016.2

Problem solving
Problem-solving agents
Well-definedness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Admissible heuristics from subproblems (cont.)

Example

Would it be possible to heuristics obtained from the $1 - 2 - 3 - 4$ database and the $5 - 6 - 7 - 8$? The two seem not to overlap ...

- Would this still give an admissible heuristic?

Answer is no, because solutions to $1 - 2 - 3 - 4$ and $5 - 6 - 7 - 8$ subproblem for a given state will almost certainly share moves

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

- But what if we do not count those moves? Like, we record not the total cost of solving the $1 - 2 - 3 - 4$ subproblem, but 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 on the cost of solving the entire problem

Admissible heuristics from subproblems (cont.)

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

Example

With such databases, it is possible to solve random 15-puzzles in a few 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 because the problem can be divided up in such a way that each move affects only one subproblem, because only one tile is moved at a time

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedtness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Learning heuristics

Heuristic function

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedtness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Learning heuristics

A heuristic function $h(n)$ is supposed to estimate the cost of a solution beginning from the state at node n

How could an agent build such a function?

- Devise relaxed problems for which an optimal solution can be found easily

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 from which $h(n)$ can be learned
- Each example consists of a state from the solution path and the actual cost of the solution from that point

Solving by searching
UFC/DC
AI (CK0031)
2016.2
Problem solving
Problem-solving agents
Well-definedtness
Problem formulation
Examples
Searching for solutions
Search algorithms
Measuring performance
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 search
A^* search
Memory-bounded search
Heuristic functions
Accuracy and performance
Admissible heuristics from relaxed problems
Admissible heuristics from subproblems
Learning heuristics

Learning heuristics (cont.)

A learning algorithm can be used to build a function $h(n)$ that can 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** of a state that are relevant to predicting the state's value

- rather than with just the raw state description

Learning heuristics (cont.)

Example

The feature 'number of misplaced tiles' might 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 and gather statistics on their actual solution costs

- We might find that when $x_1(n)$ is 5, the average solution cost is around 14, and so on

Given these data, the value of x_1 can be used to predict $h(n)$

Of course, we can use several features

Example

For example, a second feature $x_2(n)$ might be 'number of pairs of adjacent tiles that are not adjacent in the goal state'

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

$$h(n) = c_1 x_1(n) + c_2 x_2(n)$$

Constants c_1 and c_2 are adjusted to give the best fit to the actual data on solution costs

Example

One expects both c_1 and c_2 to be positive because misplaced tiles and incorrect adjacent pairs make the problem harder to solve

Notice that this heuristic does satisfy the condition that $h(n) = 0$ for goal states, but it is not necessarily admissible or consistent