CS-E4800 Artificial Intelligence

Jussi Rintanen

Department of Computer Science
Aalto University

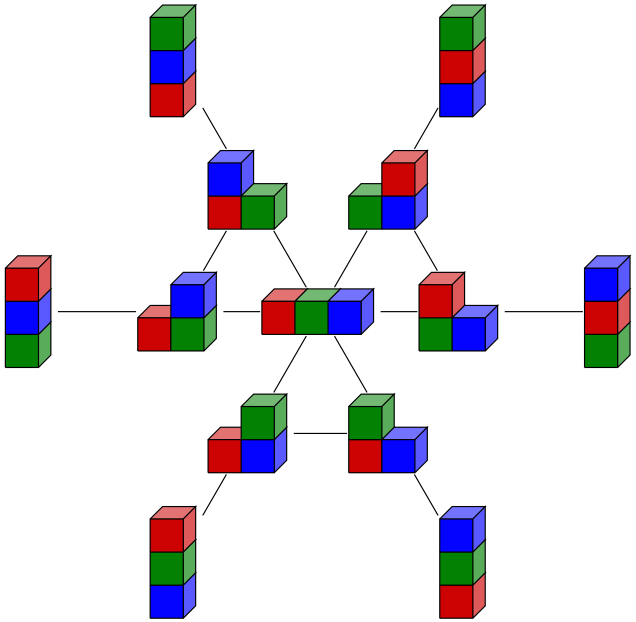
January 12, 2017

Transition System Models

- The decision-making and planning at the top-level of many intelligent systems representable as state transition models
- The world/system is in some state
- The agent can choose an action
- Different actions lead to different successor states

State-space transition graphs

Stacks with three blocks, 1-block moves

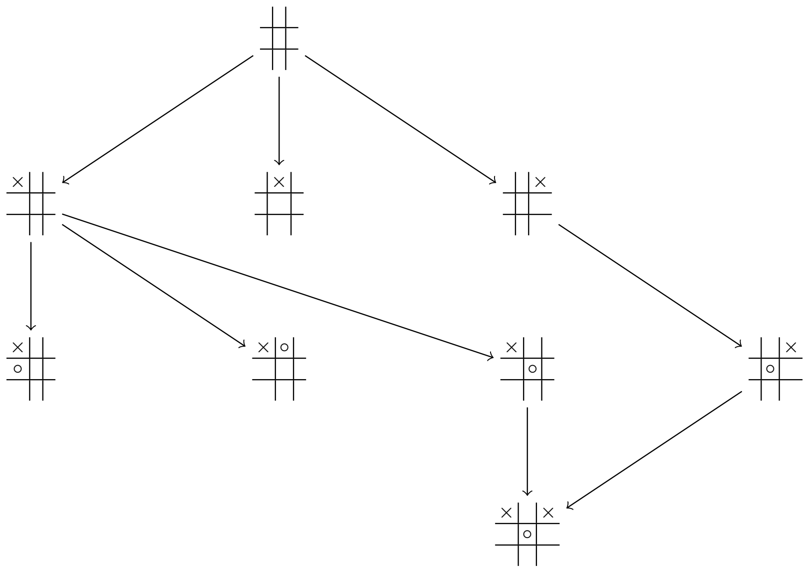


System Models

- Transition systems can be viewed as graphs
- node = state of the world
- arc = transition by some action or event
- labels on arcs can represent
  - probability of a transition from a given state
  - action the transition is associated with

Tic Tac Toe

Small fragment of the graph ($3^9 = 19683$ nodes)



Large Systems, Succinct Representations

Graphs often too large to be described enumeratively.

- node = state = valuation of state variables
- arc = action = changes in values of state variables

Succinct Representation

System with $\mathcal{O}(2^N)$ states has representation of size N

Example: Logistics

State of the system is determined by

1. locations of all vehicles (and other objects)

Location could be alternatively defined as

- coordinates in some system (N60.1869 E24.8223)
- node in a graph (locations, road segments)

Location vector for N vehicles:
$(Otaniemi, Tapiola, Pasila, Westend, \ldots)$

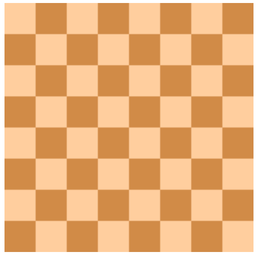
10 objects in 1000 locations $\Longrightarrow 1000^{10} = 10^{30}$ states

Example: Chess

- Number of states estimated to be 10^{43} to 10^{47}.
- Typically 30 to 40 arcs starting in a node.
- Some nodes have no arcs, others more than 200.
- The whole state space for chess can be described compactly.

Example: Chess States

- Letters KRBQNPkrbqnp denote the pieces (WHITE, black) and - denotes an empty square.
- state = listing of the contents of the board



```
--------
-p--kppp
----p---
-PPpPP--
---K--PP
--------
--------
--------
```

+ whose turn (W or B)

- As a vector

```
---------p--kppp----p----PPpPP-----K--PP------------------------W
```

which can be encoded in $4 \times 64 + 1 = 257$ bits.

Example: Chess Moves

- Consider the move of Bishop from (x, y) to (x', y')
- precondition:
  - $x \neq x'$
  - $|x - x'| = |y - y'|$
  - cells between (x, y) and (x', y') are empty (including (x', y'))
- effects:
  - (x, y) is empty
  - (x', y') is the current cell for the Bishop
- Precondition is checked against the state-vector, and the effects are achieved by modifying the state vector.

A Simple Language for State-Models

Definition

An action corresponds to

1. precondition: equalities $x_i = 1$ and $x_i = 0$,
2. effects: $x_i := 0$, $x_i := 1$.

Definition

1. a set X of state variables (with values 0 and 1)
2. a set A of actions
3. an initial state $I : X \to \{0, 1\}$

State Models with an Exponential Size

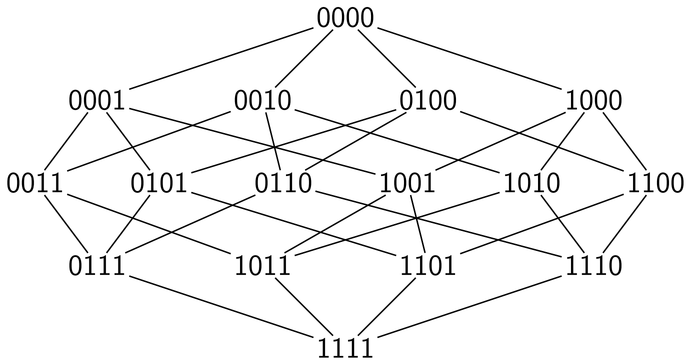
State variables: $x_1, \ldots, x_n$

Actions: Precondition $x_i = 0$ and effect $x_i := 1$ for every $i \in \{1, \ldots, n\}$

Initial state: $x_1 = 0, x_2 = 0, \ldots, x_n = 0$

Remark 1: There are 2^n states. All reachable.

Remark 2: There are $k!$ paths to a state with k 1s.



Complexity of Paths in State Models

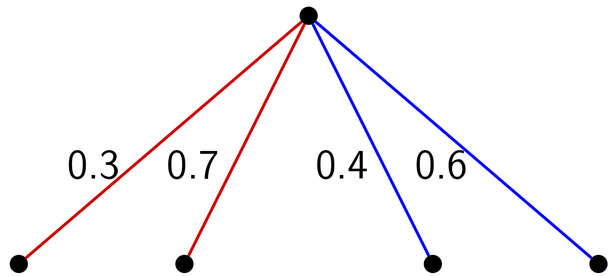
- Finding a path in the graph takes exponential time (worst-case) by all algorithms: some paths have length exponential in the size of the graph description.
- Testing if path exists is PSPACE-hard (Bylander 1994).
- NP-complete for polynomially long ("short") paths
- Other compact representations of large graphs have the same properties (Papadimitriou & Yannakakis 1986,Lozano & Balcazar 1990).

Extensions to the Basic Model

- What we considered above can faithfully model deterministic actions by a single agent in a deterministic and static environment.
- Extension 1: Uncertain effects of an action
- Extension 2: Multiple agents
- Extension 3: Partial observability

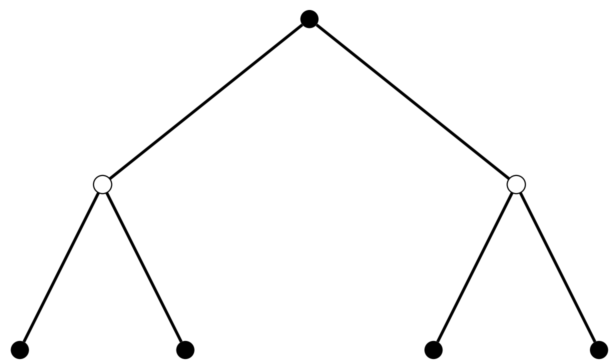
Extension 1: Nondeterminism

- All arcs have a label (depicted as a color below).
- For a given node, only the label can be chosen, not the arc directly.
- All states reached with an arc with the chosen label are possible.
- We could also associate probabilities with the arcs.



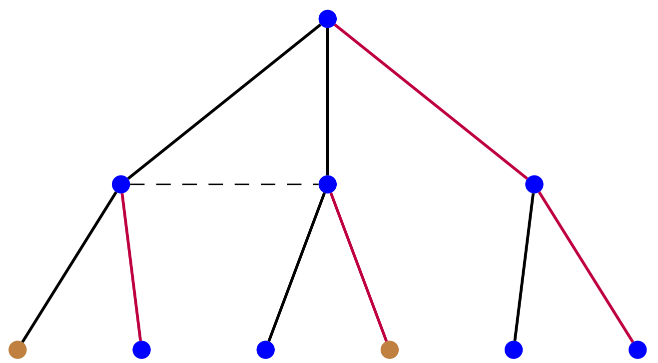
Extension 2: Multiple Agents

- Each node has a label, corresponding to one of n agents (depicted as $\circ$ and $\bullet$ below)
- In every node only the agent corresponding to the node's label can act.



Extension 3: Partial Observability

- Not always possible to distinguish between states
- Each state is labelled with an observation.
- The current action is a function of the history, consisting of actions and labels of visited states.



Decision-Making Problems: Classification

Observability

FO	fully observable
PO	partially observable

Predictability

det	deterministic actions and environment
nondet	non-deterministic actions or environment
prob	non-det. with probabilities, optimal solutions

Representation

enumerative	representation as a (labeled) graph
succinct	representation with state variables

Classification to Complexity Classes

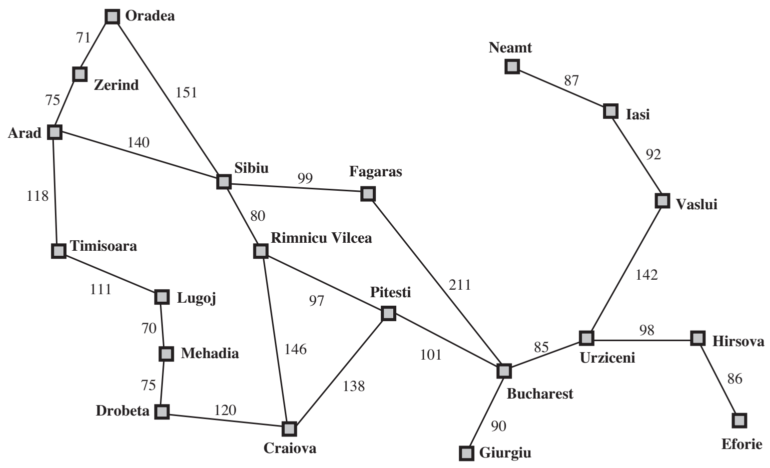
undecidable	prob PO (enumerative, succinct)
↑	
2-EXPTIME	nondet PO, succinct
↑	
EXPSPACE	nondet UO, succinct
↑	
EXPTIME	prob FO, succinct
↑	*provably intractable*
PSPACE	det FO, succinct
↑	
NP	det FO, succinct, polynomial horizon
↑	*presumably intractable*
P	prob FO, enumerative
↑	
NLOGSPACE	det FO, enumerative — *tractable*

State-Space Search

- Solution to many problems:
  - single-agent decision-making in deterministic fully observable environments
  - many types of single-player games and puzzles
  - also other core problems in CS outside AI: software engineering (verification, ...), ...
- Material: Russell&Norvig 3–3.5.2

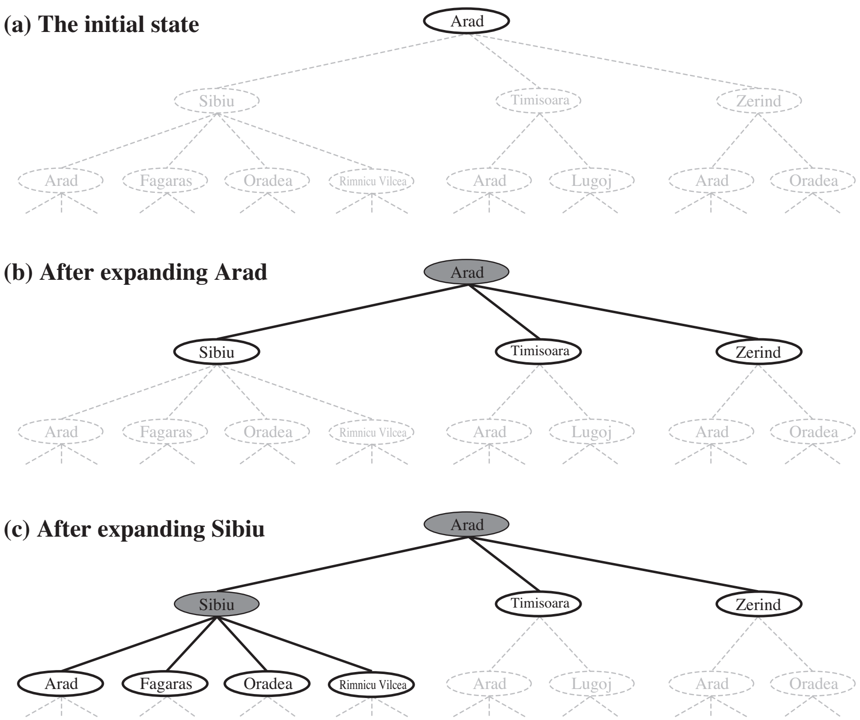
Example: Route-Planning in Romania

Arad to Bucharest



Search algorithms in this lecture

Data: *problem*, *strategy*
Result: solution or failure
root node ← initial state of *problem* (+ supplementary data)
while candidate nodes for expansion do
choose an unexpanded node for expansion according to *strategy*
if chosen node is a goal node then
return path from the root node to the goal node
else
expand the node
add new nodes to the set of unexpanded nodes
end
end
return failure



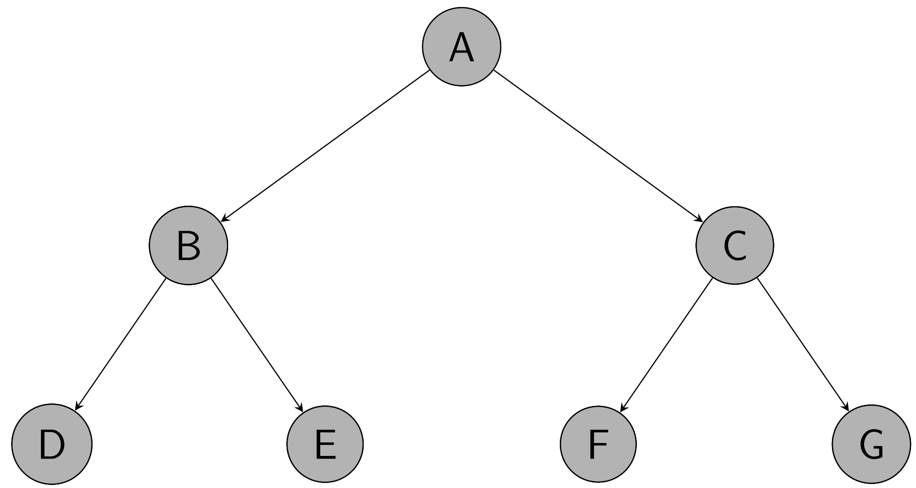
Properties of search algorithms

- Completeness: Guaranteed to find a solution when there is one, and, guaranteed to report failure given a finite but unsolvable problem.
- Optimality: Is the found solution optimal? (smallest cost)
- Time complexity: How much CPU time is used?
- Space complexity: How much memory is used?

Search strategies

- The set of unexpanded nodes is stored
- Relevant data structure is a queue
- Different queue type ⇒ different search algorithm
  - Breadth-first search (first-in first-out queue)
  - Depth-first search (stack = last-in first-out queue)
  - A* (priority queue)

Breadth-first search



the first-in-first-out queue:

A	B	C	D	E	F	G

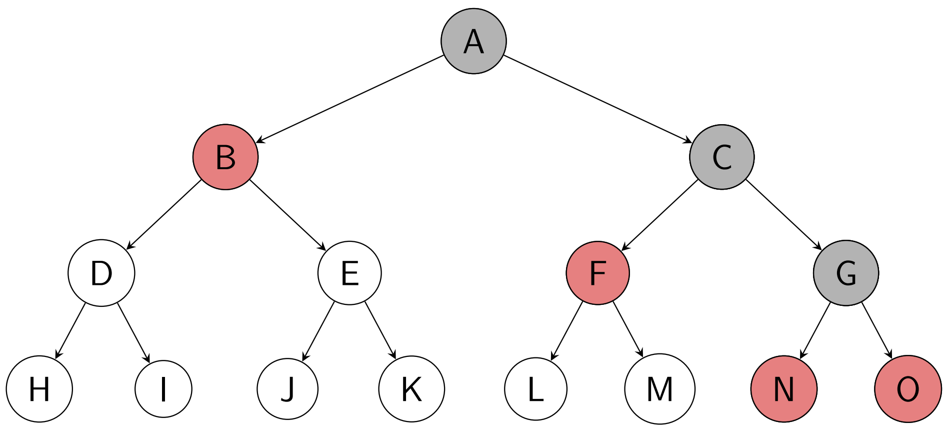
Breadth-first search

- Uses basic first-in first-out (FIFO) queue
- Expands the shallowest node first
- Complete (always finds a solution)
- Optimal when costs constant (uniform costs)
- Both time and space complexities are $O(b^d)$
  b is branching factor
  d is depth of shallowest solution

Depth-first search

- Uses stack or last-in first-out (LIFO) queue
- Expands the deepest node first
- Only stores the current path from starting node
- Complete if cycles in the current path detected
  (will otherwise go infinitely deep in a cycle)
- Not optimal
- Time complexity $O(b^m)$
  m is maximum depth of any node
- Space complexity $O(bm)$

Depth-first search



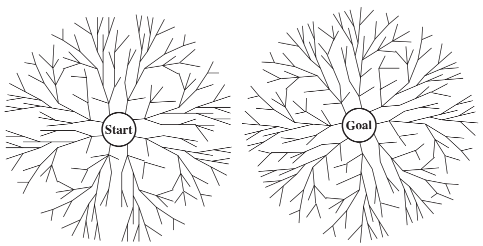
the stack: | A B | C F | G N | O |

Iterative deepening

- Run depth-first search multiple times with depth limit $m = 0, 1, 2, \ldots, d$
- Combines best of breath-first and depth-first
- Complete
- Optimal in the uniform cost case
- Time complexity $O(b^d)$
- Space complexity $O(bd)$

Bidirectional search

- Interleave two breadth-first searches
  - Forward from start towards goals
  - Backward from goals toward start
- Solution is found when the two searches meet
- Time and space complexities $O(b^{d/2})$ (breadth-first)
- Requires a method for running actions backwards



Extraction of Solution Paths

- Extracting a path from the starting state to a goal state requires storing the predecessors of encountered states
- search node = (state,predecessor *search node*)
- If solution paths are not needed, we can use *search node* = *state*
- Solution extraction:
  1. *node* := the encountered goal node
  2. output *node*
  3. *node* := the predecessor node of *node*
  4. if *node* $\neq$ starting node, go to 1

  This produces the path from starting to goal state backwards, so reverse it.

The A* algorithm

- $f(n) = g(n) + h(n)$ approximates solution cost:
  - $g(n)$ is path cost from start to n
  - $h(n)$ is an approximation of cost from n to goal
- Expand n with lowest $f(n)$ (use a priority queue)
- Optimal if $h(n)$ is a lower bound of actual remaining cost (h is admissible)
- A stronger condition is monotonicity (consistency)

$$h(n) \leq (g(n') - g(n)) + h'(n)$$

under which it is sufficient to expand a node with a given state only once. (Discussed later.)

Search nodes for A*

Definition (Search nodes for A*)

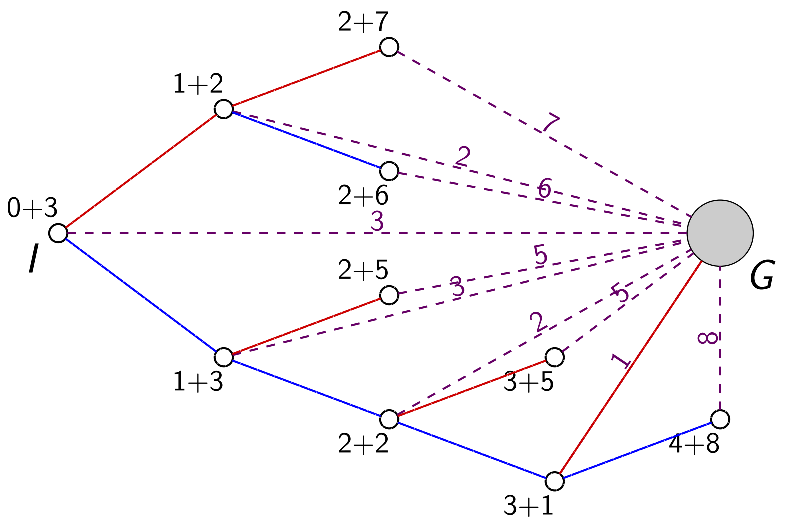
A search node is $n = (s, n_p, c)$ where
- s is a state,
- n_p is the predecessor node, and
- c is the cost of the path from the initial state.

Define $g(n) = c$, $h(n) = h(s)$ and state(n)=s.

Successor set $succ(n)$ consists of all (s', n, c') s. t.
- a is one of the actions possible in s,
- s' is the successor of s w.r.t. a, and
- $c' := c + cost(s, a, s')$.

The A∗ algorithm

Example



The A∗ Algorithm

Algorithm A∗

1: $n_0 := (s_0, n_d, 0)$ for s_0 starting state and "dummy" node n_d;
2: OPEN $:= \{n_0\}$; CLOSED $:= \emptyset$; best $:= \infty$;
3: Take any $n \in$ OPEN with the least $f(n)$;
4: **if** $f(n) \geq best$ **then** go to 12;
5: OPEN $:=$ OPEN$\setminus\{n\}$; CLOSED $:=$ CLOSED$\cup\{n\}$.
6: **for all** $n' \in succ(n)$ **do**
7:   **if** state(n') is a goal state **then** best $:= \min(best, g(n'))$;
8:   **if** $g(n'') \leq g(n')$ and state(n'')=state(n')
9:     for no $n'' \in$OPEN$\cup$CLOSED
10:   **then** OPEN $:=$ OPEN$\cup\{n'\}$;
11: **if** OPEN$\neq \emptyset$ **then** go to 3.
12: **if** best$< \infty$ **then** extract solution **else** no solution;

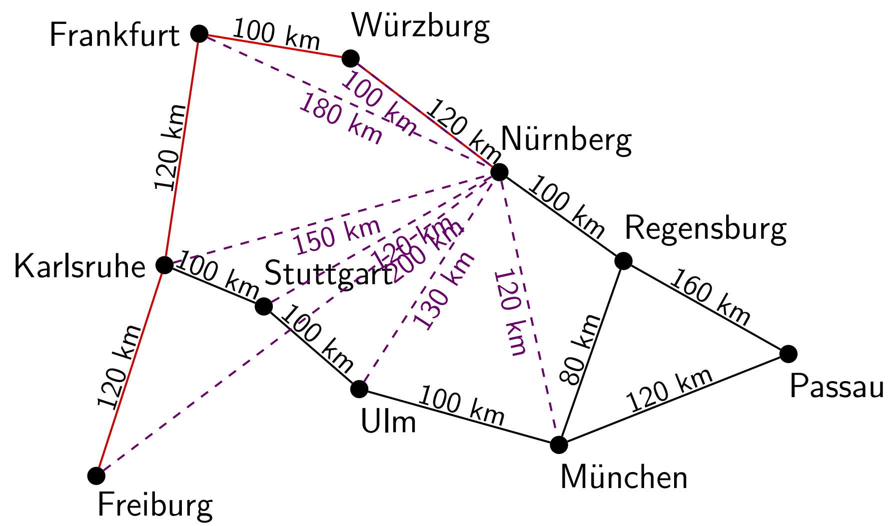
The A* Algorithm

Critical for efficient implementations:

- Data structures for OPEN and CLOSED
  - membership tests: e.g. hash table
  - open state with least $f(n)$: priority queue
- Compact state representation: few cache misses

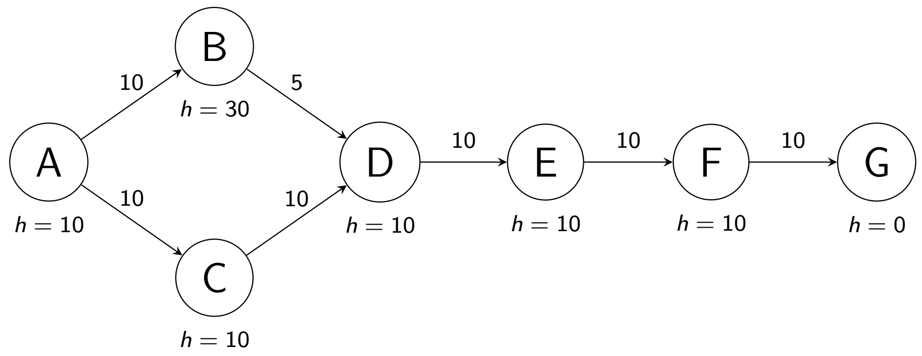
Example: A* for route planning

Shortest path from Freiburg to Nürnberg



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

Non-monotonic h-functions



Note that h above is admissible but not monotonic because $h(B) > +5 + h(D)$.
Expansion of B is delayed because $f(B) = 10 + 30$, and A* will expand nodes with state D twice: $(D, C, 20)$ and $(D, B, 15)$.

Consequences of monotonicity of h-functions

When h is monotonic...

- All nodes with a state s preceding s' are guaranteed to be expanded before any node for s'.
- Instead of keeping track of open/closed nodes, keep track of open/closed states: for a given state s, only one node for s will ever need to be expanded.

Optimality of A*

- Any node n with $f(n) <$best is potentially part of a still better solution
- A* expands all such nodes, to guarantee optimality