Machine Intelligence

Lecture 2: Search

Thomas Dyhre Nielsen

Aalborg University

Tentative course overview

Topics:

- Introduction
- **Search-based methods**
- Constrained satisfaction problems
- Logic-based knowledge representation
- Representing domains endowed with uncertainty.
- Bayesian networks
- Machine learning
- Planning
- Multi-agent systems

Problem Solving as Search

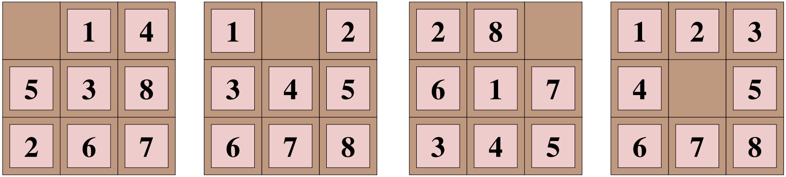
Problem Description

We consider problems where an agent

- has a state-based representation of its environment
- can observe with certainty which state it is in
- has a certain goal it wants to achieve
- can execute actions that have definite effects (no uncertainty)

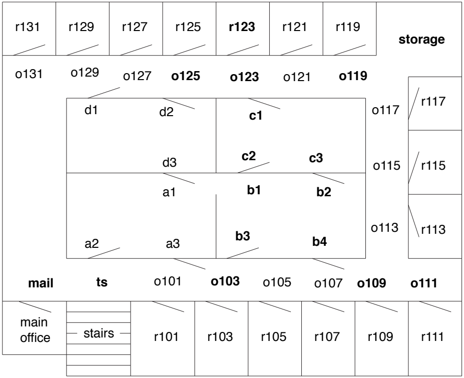
The agent needs to find a sequence of actions that lead it to a **goal state**: a state in which its goal is achieved.

Problem: re-arrange tiles into goal configuration:



Goal Goal

- There are 362880 states
- Actions: *move_up*, *move_down*,*move_left*, *move_right*



- States: locations, e.g. r131, storage, o117, c3,. . .
- Actions: move to neighboring locations, e.g. *move_r131_o131*, *move_o119_storage*, *move_b2_c3*,. . .

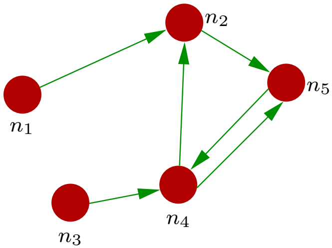
State-Space Problem

A **State-Space Problem** consists of

- A set of states
- A subset of **start states**
- A set of actions (not all actions available at all states)
- An **action function** that for a gives state s and action a returns the state reached when executing a in s
- A **goal test** that for any state s returns the boolean value *goal*(s) (true if s is a goal state)
- (optional) a **cost function** on actions
- (optional) a **value function** on goal states

A **Solution** consists of

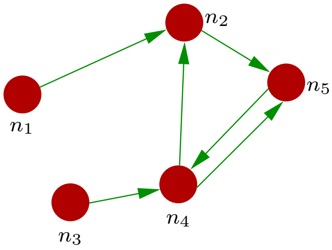
- For any given start state, a sequence of actions that lead to a goal state
- (optional) a sequence of actions with minimal cost
- (optional) a sequence of actions leading to a goal state with maximal value



A **directed graph** consists of

- a set of **nodes**
- a set of **arcs** (ordered pairs of nodes)

Graphs



A **directed graph** consists of

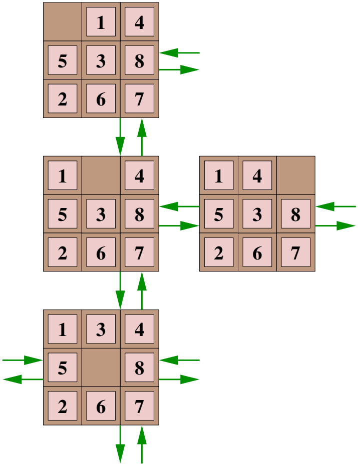
- a set of **nodes**
- a set of **arcs** (ordered pairs of nodes)

Further terminology:

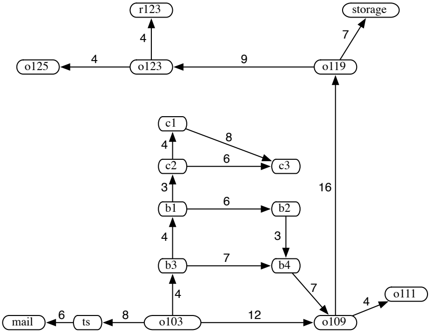
- n_2 is a **neighbor** of n_4 (not the other way round!).
- n_3, n_4, n_2, n_5 is a **path** from n_3 to n_5.
- n_2, n_5, n_4, n_2 is a path that is a **cycle**.
- a graph is **acyclic** if it has no cycles.

- Nodes: states
- Arcs: possible state-transitions from actions (arcs can be labeled with actions)

8 puzzle graph (part)



Delivery Robot Graph



- Arcs labeled with costs (time to travel, fuel costs, ...)
- *Forward branching factor* of a node = number of arcs leaving that node.
- *Backward branching factor* of a node = number of arcs entering that node.

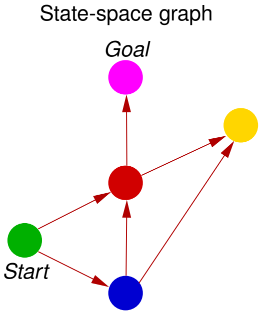
Graph Search

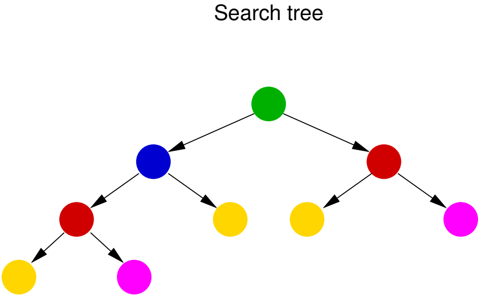
- A state-space problem can be solved by searching in the state-space graph for paths from start states to goal states.

Graph Search

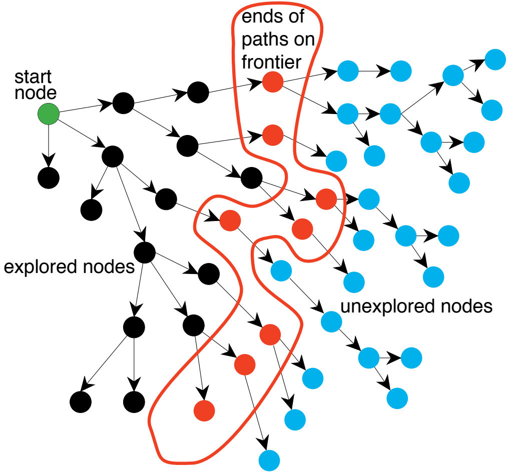
- A state-space problem can be solved by searching in the state-space graph for paths from start states to goal states.
- This does not require the whole graph at once: search may only locally generate neighbors of currently visited node.

From Graph to Search Tree





- Tree: special graph with
  - exactly one node that has no incoming arc (the **root**)
  - all other nodes have exactly one incoming arc (may have 0,1,2,3,. . . outgoing arcs)
- Nodes in the search tree correspond to paths in the graph beginning in the start state
- Nodes in the search tree also are labeled with states: the last state of the path


ends of
paths on
frontier
start
node
explored nodes
unexplored nodes

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

frontier $:= \quad \{\langle s \rangle : s$ is a start node$\}$;

while *frontier* is not empty:
select and **remove** path $\langle n_0, \ldots, n_k \rangle$ from *frontier*;
if $goal(n_k)$
return $\langle n_0, \ldots, n_k \rangle$;
for every neighbor n of n_k
add $\langle n_0, \ldots, n_k, n \rangle$ to *frontier*;
end while

Generic Search Algorithm

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

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

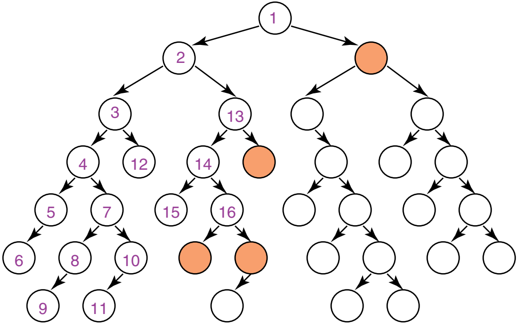
The algorithm does not require the complete graph as input: only needed are:

- List of start nodes (often only one)
- Boolean function *goal*(Node n)
- Function *get_neighbors*(Node n) returning list of neighbors of n.

- Select from the frontier that path that was most recently added to the frontier (frontier implemented as **stack**).

Example

- Explored nodes with order of exploration
- Frontier (colored)
- Unexplored nodes



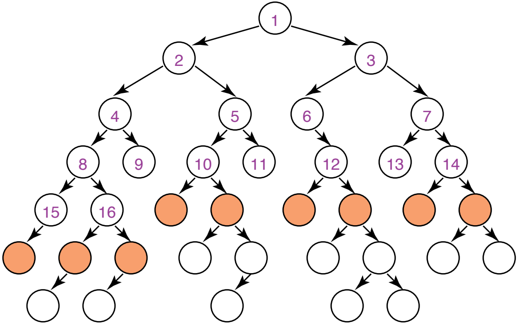
Properties

- Space used is linear in the length of the current path.
- May not terminate if state-space graph has cycles
- With a forward branching factor bounded by b and depth n, the worst-case time complexity of a finite tree is b^n.

- Select from the frontier that path that was earliest added to the frontier (frontier implemented as **queue**).

Example

- Explored nodes with order of exploration
- Frontier (colored)
- Unexplored nodes



Breadth-first search

- Will always find a solution if one exists
- Size of frontier always increases during search up to order of magnitude of total size of search tree.

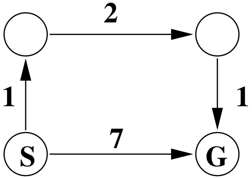
Breadth-first search

- Will always find a solution if one exists
- Size of frontier always increases during search up to order of magnitude of total size of search tree.
- Can be adapted to find a minimum cost path.

Problem

- Assume that for each action at each state we have an associated **cost**
- The cost of a solution is the sum of the costs of all actions on the path from start to goal state.
- A **minimum cost solution** is a solution with minimal cost.

Example



- Breadth first search will find the *shortest*, but not the *cheapest* solution.
- Depth first search may find either solution, depending on order of neighbor enumeration

Lowest-Cost-First Search

Simple modification of generic search algorithm:

- with each path in *frontier* store the cost of the path
- Modify one line of code:
  select and **remove** path $\langle n_0, \ldots, n_k \rangle$ with minimal cost from *frontier*;

Properties

- If all actions have non-zero cost, and solution exists, then a minimal cost solution will be found.
- Space requirement depends on cost structure, but usually similar to breadth-first search.

Goal

- Termination guarantee of breadth-first search
- Space efficiency of depth-first search

Algorithm

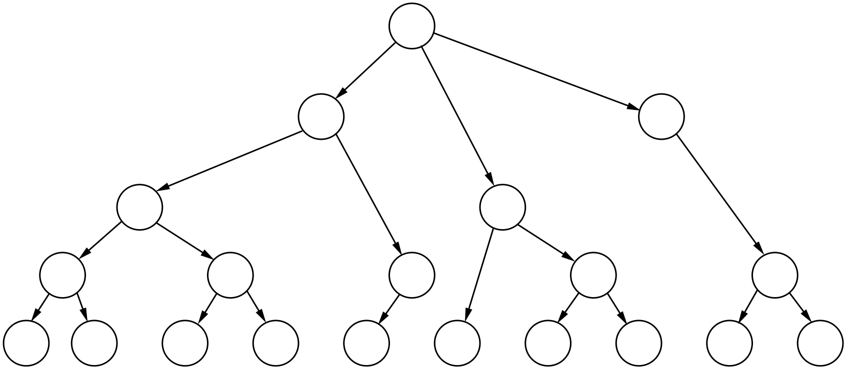
Depth-bounded search k

- As depth-first search, but
- do not add neighbors of selected node to frontier if selected node has depth k.

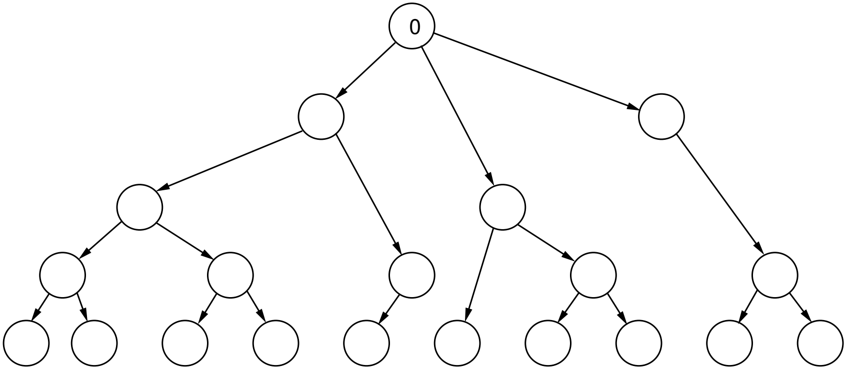
Iterative deepening search

- For $k = 1, 2, 3, \ldots$ perform depth-bounded search k.

Iterative deepening: an example



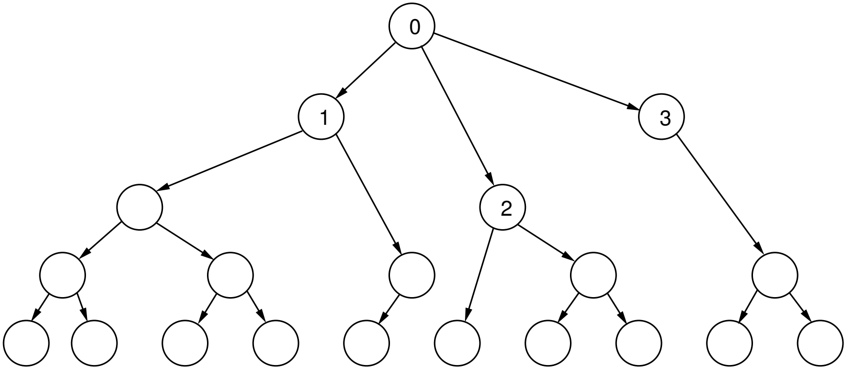
Iterative deepening: an example



Perform depth-bounded search to level $k = 1$.

NB: The node numbering corresponds to when the states are first visited.

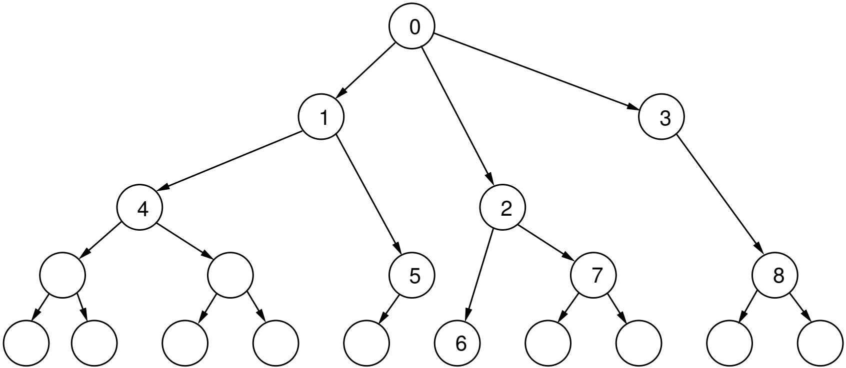
Iterative deepening: an example



Perform depth-bounded search to level $k = 2$.

NB: The node numbering corresponds to when the states are first visited.

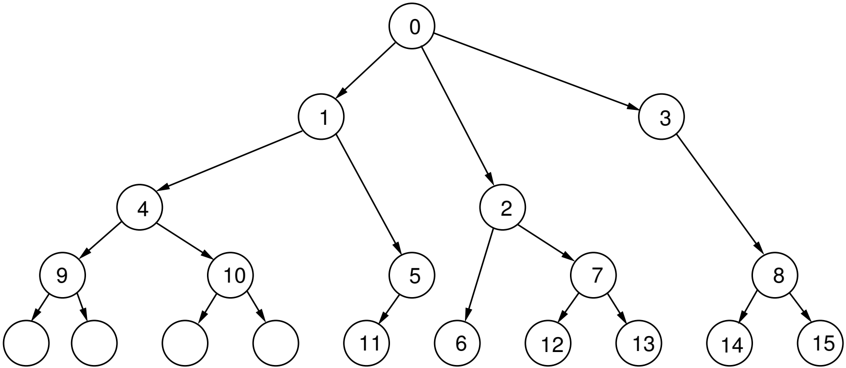
Iterative deepening: an example

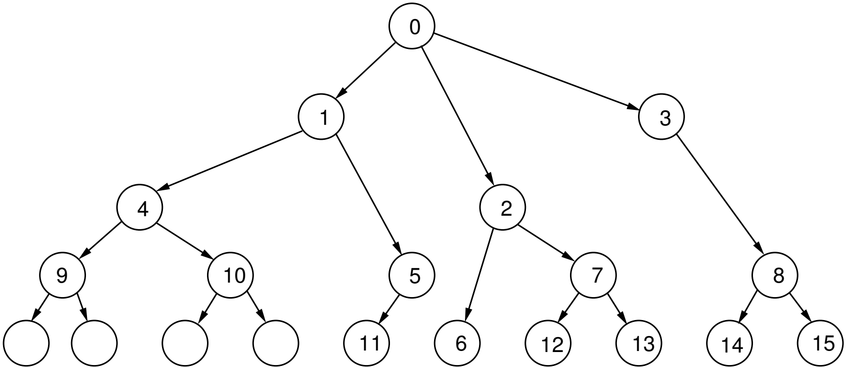


Perform depth-bounded search to level $k = 3$.

NB: The node numbering corresponds to when the states are first visited.

Iterative deepening: an example





Properties

- Has desired termination and space efficiency properties
- Duplicates computations (depth-bounded search k repeats computaions of depth-bounded search $k-1$). Not as problematic as it looks: constant overhead of $(b/(b-1))$.

Uninformed Search

Depth-first, Breadth-first and Iterative deepening are **uninformed** search strategies:
they do not assume/use any knowledge of the search space except the pure graph structure.

Informed Search

Idea

- Lowest-Cost-First Search only considers cost of already constructed partial solution.
- Idea: Try to estimate the cost of optimal path from current state to goal.

Idea

- Lowest-Cost-First Search only considers cost of already constructed partial solution.
- Idea: Try to estimate the cost of optimal path from current state to goal.

Actual Cost

Given a cost function on actions, can define for any node n in the search tree:

> *opt(n)= cost of optimal (minimal cost) path from n to a goal state (infinite if no path to goal exists).*

- The *opt* function can usually not be computed
- *opt*(n) only depends on the state at node n.

Idea

- Lowest-Cost-First Search only considers cost of already constructed partial solution.
- Idea: Try to estimate the cost of optimal path from current state to goal.

Actual Cost

Given a cost function on actions, can define for any node n in the search tree:

opt(n)= cost of optimal (minimal cost) path from n to a goal state (infinite if no path to goal exists).

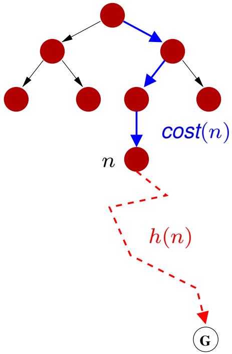
- The *opt* function can usually not be computed
- *opt*(n) only depends on the state at node n.

Heuristic Function

$h(n)$ (non-negative number): estimate of *opt(n)*. $h(n)$ is an **underestimate** if for all nodes n:

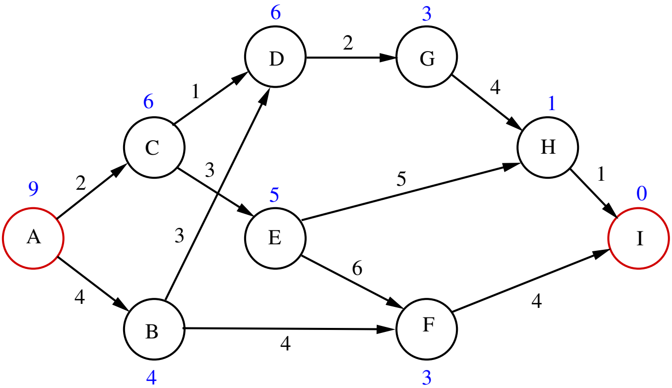
$$h(n) \leq \textit{opt}(n)$$

For any node n in the search tree we have:



- *cost*(n) : (true) cost of reaching n from the root node.
- $h(n)$: a heuristic function
- $f(n) :=$ *cost*$(n) + h(n)$

A^* search: always expand fringe node with minimal f-value.



For node D:

- $\mathit{cost}(D) = 3$
- $h(D) = 6$
- $f(D) := \mathit{cost}(D) + h(D) = 3 + 6 = 9$

Admissibility

For finite state space graphs: if

- all actions have cost > 0
- $h(n)$ is an underestimate
- there exists a solution

then A^* will return an optimal solution.

Admissibility

For finite state space graphs: if

- all actions have cost > 0
- $h(n)$ is an underestimate
- there exists a solution

then A^* will return an optimal solution.

Special Cases

- $h(n) = 0$ for all n: A^* becomes lowest-cost-first search
- $h(n) =$ *opt*(n): A^* directly constructs the optimal path

$\rightsquigarrow$ should find heuristic functions $h(n)$ that are “close underestimates” of *opt*(n).

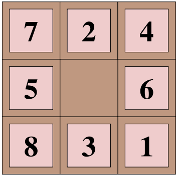
Constructing heuristic functions

General strategy to design underestimates:

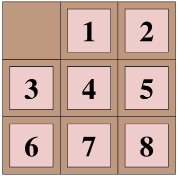
- define a simplified (easier) problem
  - add arcs, states to the state space graph
  - reduce cost of existing arcs
- use the exact function *opt* in the simplified problem as the heuristic function for the original problem
- requires: *opt* in the simplified problem must be easily computable

Simplified Problem 1: Can move any tile to any square (more states: can have several tiles on one square). Then

$h_1(n) = \mathit{opt}_1(n)$:= *Number of tiles that are not in their goal position.*



n



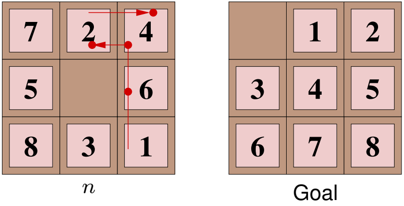
Goal

Simplified Problem 1: Can move any tile to any square (more states: can have several tiles on one square). Then

$h_1(n) =$ *opt*$_1(n)$*:= Number of tiles that are not in their goal position.*

Simplified Problem 2: Can move any tile to an *adjacent* square.

$h_2(n) =$ *opt*$_2(n)$*:= Sum of Manhatten distances of all tiles to their goal position.*



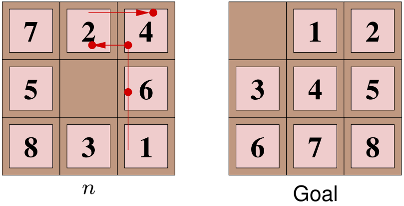
$h_1(n) = 8$

Simplified Problem 1: Can move any tile to any square (more states: can have several tiles on one square). Then

$h_1(n) = opt_1(n)$:= *Number of tiles that are not in their goal position.*

Simplified Problem 2: Can move any tile to an *adjacent* square.

$h_2(n) = opt_2(n)$:= *Sum of Manhatten distances of all tiles to their goal position.*



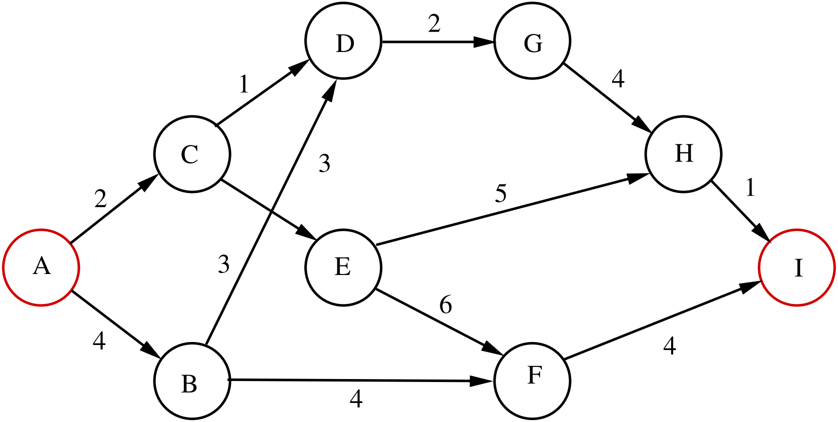
$h_1(n) = 8$
$h_2(n) = 3+1+2+2+2+3+3+2 = 18$

Dynamic programming

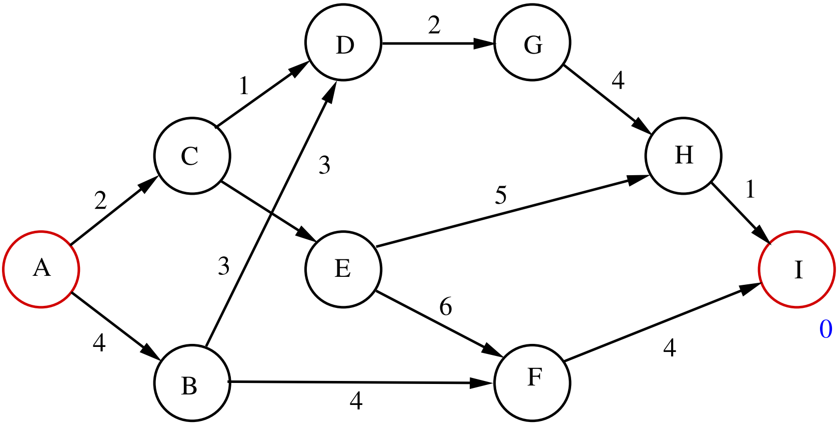
Idea: for statically stored graphs, build a table of $dist(n)$ the actual distance of the shortest path from node n to a goal.
This can be built backwards from the goal:

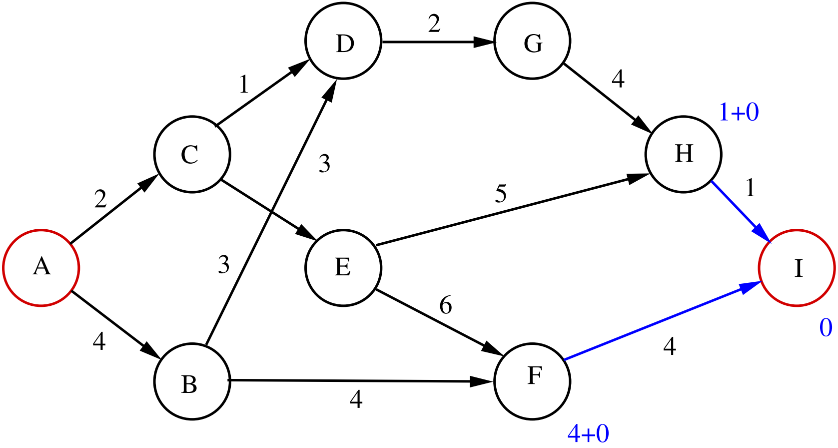
$$dist(n) = \begin{cases} 0 & \text{if } is\_goal(n), \\ \min_{\langle n,m\rangle \in A}(|\langle n,m\rangle| + dist(m)) & \text{otherwise.} \end{cases}$$

This can be used locally to determine what to do.

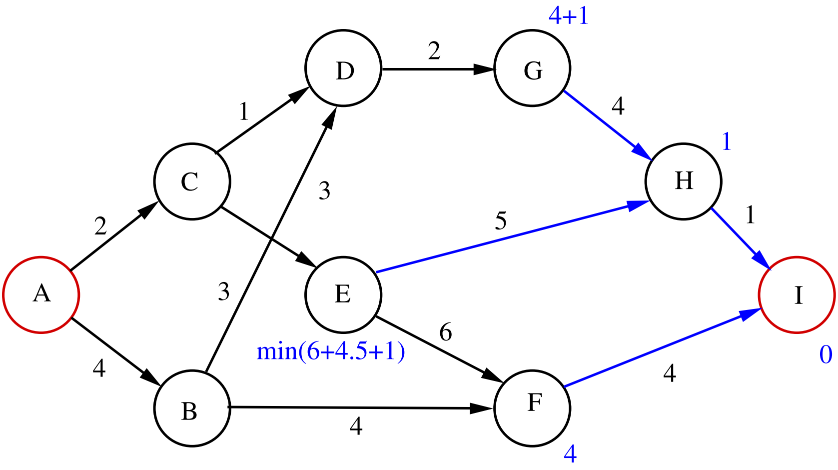

A
B
C
D
E
F
G
H
I
2
4
1
3
3
4
2
5
6
4
1
4

Dynamic programming: Example

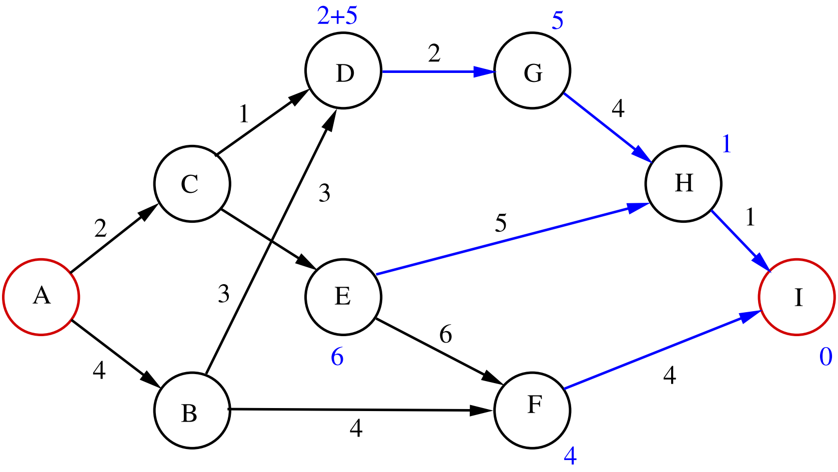



D
G
C
H
A
E
I
B
F
2
1
4
1+0
3
2
5
1
3
6
0
4
4
4
4
4+0

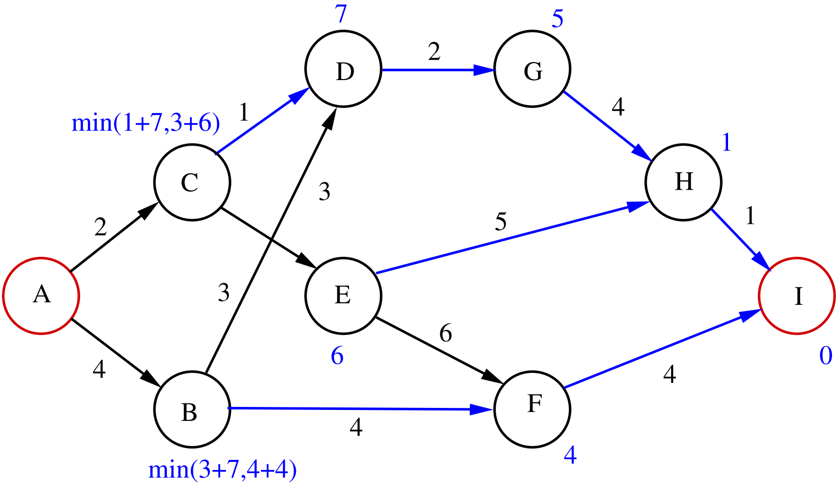
Dynamic programming: Example



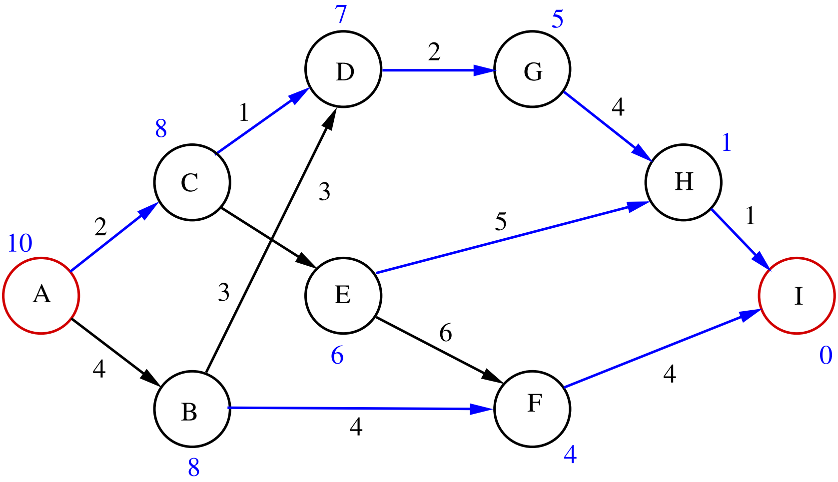
Dynamic programming: Example

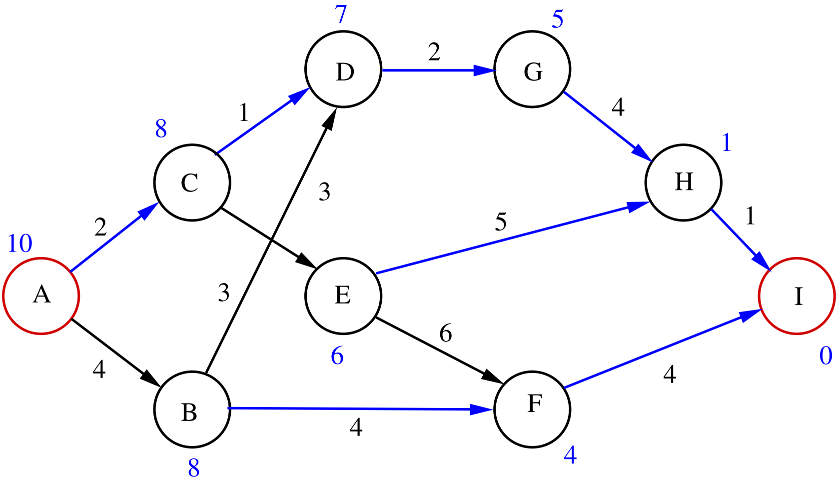


Dynamic programming: Example



Dynamic programming: Example





There are two main problems:

- You need enough space to store the graph.
- The $dist$ function needs to be recomputed for each goal.

Direction of search

- The definition of searching is symmetric: find path from start nodes to goal node or from goal node to start nodes.
- Search complexity is b^n. Should use forward search if forward branching factor is less than backward branching factor, and vice versa.
- Note: sometimes when graph is dynamically constructed, you may not be able to construct the backwards graph.

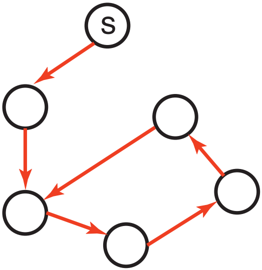
Direction of search

- The definition of searching is symmetric: find path from start nodes to goal node or from goal node to start nodes.
- Search complexity is b^n. Should use forward search if forward branching factor is less than backward branching factor, and vice versa.
- Note: sometimes when graph is dynamically constructed, you may not be able to construct the backwards graph.

Bi-directional search

- You can search backward from the goal and forward from the start simultaneously.
- This wins as $2b^{k/2} \ll b^k$. This can result in an exponential saving in time and space.
- The main problem is making sure the frontiers meet.
- This is often used with one breadth-first method that builds a set of locations that can lead to the goal. In the other direction another method can be used to find a path to these interesting locations.

Cycle checking



- A searcher can prune a path that ends in a node already on the path, without removing an optimal solution.
- Using depth-first methods, with the graph explicitly stored, this can be done in constant time.
- For other methods, the cost is linear in path length.