

Principles of Artificial Intelligence

Problem Solving as Search

Outline

- 1 Problem-Solving Agents
- 2 Example Problems
- 3 Search Generalities
- 4 Uninformed Search Strategies
 - Breadth-First Search
 - Uniform-cost Search
 - Depth-First Search
 - Depth-Limited Search &
 - Iterative Deepening
- 5 Informed Search Strategies
 - Greedy Search
 - A* Search Heuristic Functions

Outline

- 1 Problem-Solving Agents
- 2 Example Problems
- 3 Search Generalities
- 4 Uninformed Search Strategies
 - Breadth-First Search
 - Uniform-cost Search
 - Depth-First Search
 - Depth-Limited Search &
 - Iterative Deepening
- 5 Informed Search Strategies
 - Greedy Search
 - A* Search Heuristic Functions

Problem Solving as Search

One of the dominant approaches to AI problem solving: formulate a problem/task as search in a state space.

- ① Goal formulation: define the successful world states
 - Ex: a set of states, a Boolean test function ...
- ② Problem formulation:
 - define a representation for states
 - define legal actions and transition functions
- ③ Search: find a solution by means of a search process
 - solutions are sequences of actions

Execution: given the solution, perform the actions

==> Problem-solving agents are (a kind of) goal-based agents

Problem Solving as Search

One of the dominant approaches to AI problem solving: formulate a problem/task as search in a state space.

- ① Goal formulation: define the successful world states
 - Ex: a set of states, a Boolean test function ...
- ② Problem formulation:
 - define a representation for states
 - define legal actions and transition functions
- ③ Search: find a solution by means of a search process
 - solutions are sequences of actions

Execution: given the solution, perform the actions

==> Problem-solving agents are (a kind of) goal-based agents

Problem Solving as Search

One of the dominant approaches to AI problem solving: formulate a problem/task as search in a state space.

- ① Goal formulation: define the successful world states
 - Ex: a set of states, a Boolean test function ...
- ② Problem formulation:
 - define a representation for states
 - define legal actions and transition functions
- ③ Search: find a solution by means of a search process
 - solutions are sequences of actions

Execution: given the solution, perform the actions

==> Problem-solving agents are (a kind of) goal-based agents

Problem Solving as Search

One of the dominant approaches to AI problem solving: formulate a problem/task as search in a state space.

- ① Goal formulation: define the successful world states
 - Ex: a set of states, a Boolean test function ...
- ② Problem formulation:
 - define a representation for states
 - define legal actions and transition functions
- ③ Search: find a solution by means of a search process
 - solutions are sequences of actions

Execution: given the solution, perform the actions

==> Problem-solving agents are (a kind of) goal-based agents

Problem Solving as Search

One of the dominant approaches to AI problem solving: formulate a problem/task as search in a state space.

- ① Goal formulation: define the successful world states
 - Ex: a set of states, a Boolean test function ...
- ② Problem formulation:
 - define a representation for states
 - define legal actions and transition functions
- ③ Search: find a solution by means of a search process
 - solutions are sequences of actions

Execution: given the solution, perform the actions

==> Problem-solving agents are (a kind of) goal-based agents

Problem Solving as Search

One of the dominant approaches to AI problem solving: formulate a problem/task as search in a state space.

- ① Goal formulation: define the successful world states
 - Ex: a set of states, a Boolean test function ...
- ② Problem formulation:
 - define a representation for states
 - define legal actions and transition functions
- ③ Search: find a solution by means of a search process
 - solutions are sequences of actions

Execution: given the solution, perform the actions

==> Problem-solving agents are (a kind of) goal-based agents

Problem Solving as Search

One of the dominant approaches to AI problem solving: formulate a problem/task as search in a state space.

- ① Goal formulation: define the successful world states
 - Ex: a set of states, a Boolean test function ...
- ② Problem formulation:
 - define a representation for states
 - define legal actions and transition functions
- ③ Search: find a solution by means of a search process
 - solutions are sequences of actions

Execution: given the solution, perform the actions

==> Problem-solving agents are (a kind of) goal-based agents

Problem Solving as Search: Example

Example: Traveling in Romania

- **Informal description:** On holiday in Romania; currently in Arad. Flight leaves tomorrow from Bucharest
- **Formulate goal:** (Be in) Bucharest
- **Formulate problem:**
 - States: various cities
 - Actions: drive between cities
 - Initial state: Arad
- **Search for a solution:**
sequence of cities from Arad
to Bucharest

e.g. Arad, Sibiu, Fagaras, Bucharest
explore a search tree/graph

Note

The agent is assumed to have no heuristic knowledge about traveling in Romania to exploit

Problem Solving as Search: Example

Example: Traveling in Romania

- **Informal description:** On holiday in Romania; currently in Arad. Flight leaves tomorrow from Bucharest
- **Formulate goal:** (Be in) Bucharest
- **Formulate problem:**
 - States: various cities
 - Actions: drive between cities
 - Initial state: Arad
- **Search for a solution:**
sequence of cities from Arad
to Bucharest

e.g. Arad, Sibiu, Fagaras, Bucharest

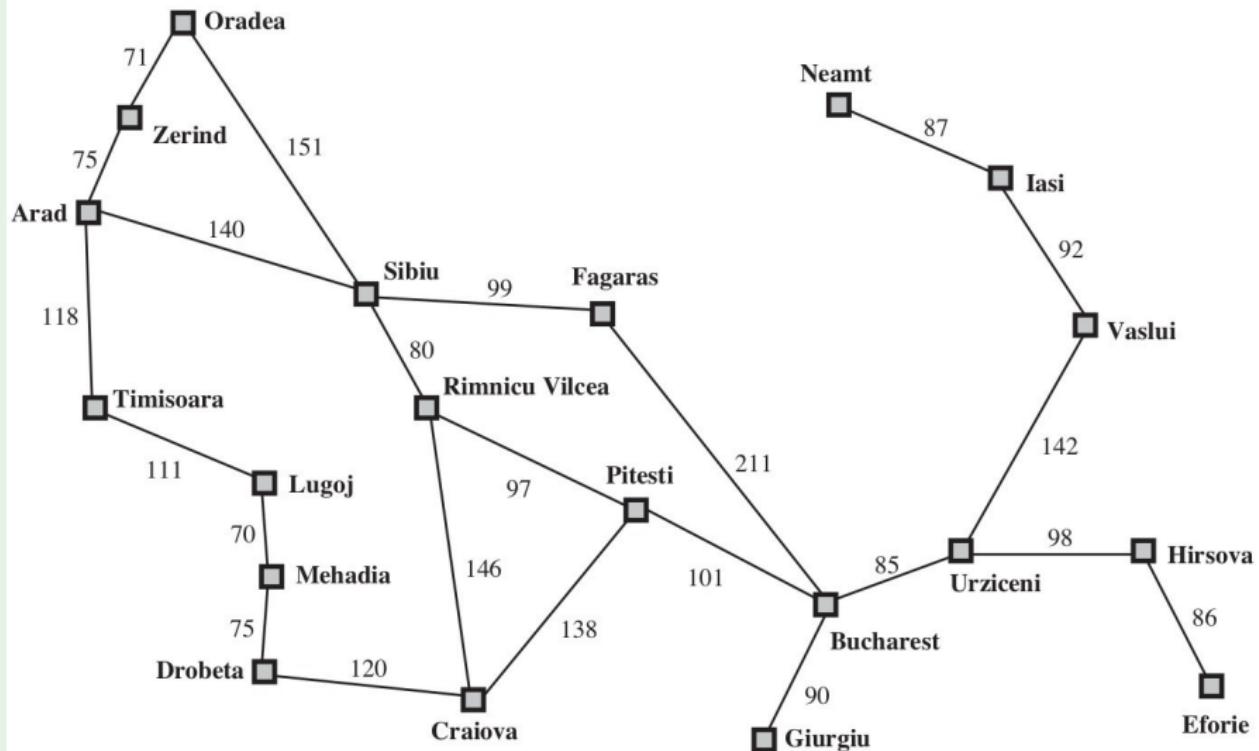
explore a search tree/graph

Note

The agent is assumed to have no heuristic knowledge about traveling in Romania to exploit

Problem Solving as Search: Example [cont.]

A simplified road map of part of Romania.



Problem Solving as Search [cont.]

Assumptions for Problem-solving Agents (this chapter)

- only state representations are **atomic**

==> world states are considered as wholes, with no internal structure

- Ex: Arad, Sibiu, Zerind, Bucharest,...

- the environment is **observable**

==> the agent always knows the current state

- Ex: Romanian cities & roads have signs

- the environment is **discrete**

==> at any state there are only finitely many actions to choose from

- Ex: from Arad, (go to) Sibiu, or Zerind, or Timisoara (see map)

- the environment is **known**

==> the agent knows which states are reached by each action

- ex: the agent has the map

- the environment is **deterministic**

==> each action has exactly one outcome

Ex: from Arad choose go to Sibiu ==> next step in Sibiu

Problem Solving as Search [cont.]

Assumptions for Problem-solving Agents (this chapter)

- only state representations are **atomic**
 - ==> world states are considered as wholes, with no internal structure
 - Ex: Arad, Sibiu, Zerind, Bucharest,...

the environment is **observable**

- ==> the agent always knows the current state
 - Ex: Romanian cities & roads have signs

the environment is **discrete**

- ==> at any state there are only finitely many actions to choose from
 - Ex: from Arad, (go to) Sibiu, or Zerind, or Timisoara (see map)

the environment is **known**

- ==> the agent knows which states are reached by each action
 - ex: the agent has the map

the environment is **deterministic**

==> each action has exactly one outcome

- Ex: from Arad choose go to Sibiu ==> next step in Sibiu

Problem Solving as Search [cont.]

Assumptions for Problem-solving Agents (this chapter)

- only) state representations are **atomic**
 - ==> world states are considered as wholes, with no internal structure
 - Ex: Arad, Sibiu, Zerind, Bucharest,...

the environment is **observable**

- ==> the agent always knows the current state
 - Ex: Romanian cities & roads have signs

the environment is **discrete**

- ==> at any state there are only finitely many actions to choose from
 - Ex: from Arad, (go to) Sibiu, or Zerind, or Timisoara (see map)

the environment is **known**

- ==> the agent knows which states are reached by each action
 - ex: the agent has the map

the environment is **deterministic**

- ==> each action has exactly one outcome

Ex: from Arad choose go to Sibiu ==> next step in Sibiu

Problem Solving as Search [cont.]

Assumptions for Problem-solving Agents (this chapter)

- only) state representations are **atomic**
 - ==> world states are considered as wholes, with no internal structure
 - Ex: Arad, Sibiu, Zerind, Bucharest,...

the environment is **observable**

- ==> the agent always knows the current state
 - Ex: Romanian cities & roads have signs

the environment is **discrete**

- ==> at any state there are only finitely many actions to choose from
 - Ex: from Arad, (go to) Sibiu, or Zerind, or Timisoara (see map)

the environment is **known**

- ==> the agent knows which states are reached by each action
 - ex: the agent has the map

the environment is **deterministic**

- ==> each action has exactly one outcome

Ex: from Arad choose go to Sibiu ==> next step in Sibiu

Problem Solving as Search [cont.]

Assumptions for Problem-solving Agents (this chapter)

- only) state representations are **atomic**
 - ==> world states are considered as wholes, with no internal structure
 - Ex: Arad, Sibiu, Zerind, Bucharest,...

the environment is **observable**

- ==> the agent always knows the current state
 - Ex: Romanian cities & roads have signs

the environment is **discrete**

- ==> at any state there are only finitely many actions to choose from
 - Ex: from Arad, (go to) Sibiu, or Zerind, or Timisoara (see map)

the environment is **known**

- ==> the agent knows which states are reached by each action
 - ex: the agent has the map

the environment is **deterministic**

- ==> each action has exactly one outcome

Ex: from Arad choose go to Sibiu ==> next step in Sibiu

Problem Solving as Search [cont.]

Assumptions for Problem-solving Agents (this chapter)

- only) state representations are **atomic**
 - ==> world states are considered as wholes, with no internal structure
 - Ex: Arad, Sibiu, Zerind, Bucharest,...

the environment is **observable**

- ==> the agent always knows the current state
 - Ex: Romanian cities & roads have signs

the environment is **discrete**

- ==> at any state there are only finitely many actions to choose from
 - Ex: from Arad, (go to) Sibiu, or Zerind, or Timisoara (see map)

the environment is **known**

- ==> the agent knows which states are reached by each action
 - ex: the agent has the map

the environment is **deterministic**

- ==> each action has exactly one outcome

Ex from Arad choose go to Sibiu ==> next step in Sibiu

Problem Solving as Search [cont.]

Notes about search

- Search happens **inside the agent**
 - a **planning** stage before acting different from **searching in the world**
- An agent is given a **description of what to achieve**, not an **algorithm**
 ==> the only possibility is to **search for a solution**

 Searching can be computationally very demanding (NP-hard)
- Can be driven with benefits by knowledge of the problem (**heuristic**)
- ==> **informed/heuristic search**

Problem Solving as Search [cont.]

Notes about search

- Search happens **inside the agent**
 - a **planning** stage before acting different from **searching in the world**
- An agent is given a **description** of what to achieve, not an **algorithm**
==> the only possibility is to **search** for a solution
Searching can be computationally very demanding (NP-hard)
- Can be driven with benefits by knowledge of the problem (**heuristic**)
 - ==> **informed/heuristic search**

Problem Solving as Search [cont.]

Notes about search

- Search happens **inside the agent**
 - a **planning** stage before acting different from **searching in the world**
- An agent is given **a description of what to achieve**, not an **algorithm**
 ==> the only possibility is to **search for a solution**

 Searching can be computationally very demanding (NP-hard)
- Can be driven with benefits by knowledge of the problem (**heuristic**)
 - ==> informed/heuristic search

Problem Solving as Search [cont.]

Notes about search

- Search happens **inside the agent**
 - a **planning** stage before acting different
 - from **searching in the world**
- An agent is given **a description of what to achieve**, not an **algorithm to solve it**
==> the only possibility is to **search for a solution**
- **Searching can be computationally very demanding (NP-hard)**

Can be driven with benefits by knowledge of the problem
(heuristic knowledge) ==> informed/heuristic search

Problem Solving as Search [cont.]

Notes about search

- Search happens **inside the agent**
 - a **planning** stage before acting different
 - from **searching in the world**
- An agent is given **a description of what to achieve**, not an **algorithm to solve it**
==> the only possibility is to **search for a solution**
- Searching can be computationally very demanding (**NP-hard**)
Can be driven with benefits by knowledge of the problem (heuristic knowledge) ==> **informed/heuristic search**

Problem-solving Agent: Schema

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

(Oc S. Russell & P. Norwig, AIMA)

While executing the solution sequence the agent ignores its percepts when choosing an action since it knows in advance what (the environment system")

Problem-solving Agent: Schema

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

(Oc S. Russell & P. Norwig, AIMA)

While executing the solution sequence the agent ignores its percepts when choosing an action since it knows in advance what the system ("the environment")

Well-defined problems and solutions

Problem Formulation: Components

- the **initial state** the agent starts
in •
 - Ex: *In(Arad)*

the set of **applicable actions** available
in a state (**ACTIONS(s)**)

- Ex: if s is *In(Arad)*, then the
 - applicable actions are
{Go(Sibiu), Go(Timisoara),
Go(Zerind)}
 - a description of what each action does (aka **transition model**) **RESULT(s,a)**: state resulting from applying action A in state s
 - Ex: **RESULT(IN(ARAD), Go(ZERIND))** is **IN(ZERIND)**

the **goal test** determining if a given state is a goal state

Explicit (e.g.: *{In(Bucharest)}*) Implicit
(e.g. (Ex: CHECKMATE(x))

the **path cost** function assigns a

Well-defined problems and solutions

Problem Formulation: Components

- the **initial state** the agent starts in •
- Ex: *In(Arad)*

the set[•] of **applicable actions** available in a state (**ACTIONS(s)**)

- Ex: if s is *In(Arad)*, then the
 - applicable actions are
{Go(Sibiu), Go(Timisoara), Go(Zerind)}
- a description of what each action does (aka **transition model**) **RESULT(s,a)**: state resulting from applying action A in state s Ex: **RESULT(IN(ARAD), Go(ZERIND))** is IN(ZERIND)

the **goal test** determining if a given state is a goal state

Explicit (e.g.: *{In(Bucharest)}*) Implicit
(e.g. (Ex: CHECKMATE(x))

the **path cost** function assigns a

Well-defined problems and solutions

Problem Formulation: Components

- the **initial state** the agent starts in •
- Ex: *In(Arad)*

the set[•] of **applicable actions** available in a state (**ACTIONS(s)**)

- Ex: if s is *In(Arad)*, then the
 - applicable actions are
{Go(Sibiu), Go(Timisoara), Go(Zerind)}
 - a description of what each action does (aka **transition model**) **RESULT(s,a)**: state resulting from applying action A in state s
 - Ex: **RESULT(IN(ARAD), Go(ZERIND))** is **IN(ZERIND)**

the **goal test** determining if a given state is a goal state

Explicit (e.g.: *{In(Bucharest)}*) Implicit
(e.g. (Ex: CHECKMATE(x))

the path cost function assigns a

Well-defined problems and solutions

Problem Formulation: Components

- the **initial state** the agent starts
in •
- Ex: *In(Arad)*

the set[•] of **applicable actions** available
in a state (**ACTIONS(s)**)

- Ex: if s is *In(Arad)*, then the
 - applicable actions are
{Go(Sibiu), Go(Timisoara),
Go(Zerind)}
- a description of what each action does (aka **transition
model**) **RESULT(s,a)**: state resulting from applying action a in
state s Ex: **RESULT(IN(ARAD), Go(ZERIND))** is **IN(ZERIND)**

the **goal test** determining if a given state is a goal state

Explicit (e.g.: *{In(Bucharest)}*) Implicit
(e.g. (Ex: CHECKMATE(x))

the **path cost** function assigns a

Well-defined problems and solutions

Problem Formulation: Components

- the **initial state** the agent starts in •
 - Ex: *In(Arad)*

the set of **applicable actions** available in a state (**ACTIONS(s)**)

- Ex: if s is *In(Arad)*, then the
 - applicable actions are *{Go(Sibiu), Go(Timisoara), Go(Zerind)}*
- a description of what each action does (aka **transition model**) **RESULT(s,a)**: state resulting from applying action a in state s Ex: **RESULT(IN(ARAD), GO(ZERIND))** is **IN(ZERIND)**

the **goal test** determining if a given state is a goal state

Explicit (e.g.: *{In(Bucharest)}*) Implicit
(e.g. (Ex: **CHECKMATE(x)**))

the path cost function assigns a

Well-defined problems and solutions

Problem Formulation: Components

- the **initial state** the agent starts
in •
- Ex: *In(Arad)*

the set[•] of **applicable actions** available
in a state (**ACTIONS(s)**)

- Ex: if s is *In(Arad)*, then the
 - applicable actions are
{Go(Sibiu), Go(Timisoara),
Go(Zerind)}
 - a description of what each action does (aka **transition model**) **RESULT(s,a)**: state resulting from applying action A in state s
 - Ex: **RESULT(IN(ARAD), Go(ZERIND))** is **IN(ZERIND)**

the **goal test** determining if a given state is a goal state

Explicit (e.g.: *{In(Bucharest)}*) Implicit
(e.g. (Ex: **CHECKMATE(x)**))

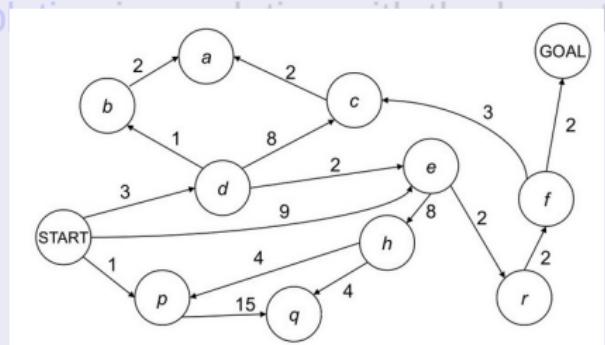
the **path cost** function assigns a

Well-defined problems and solutions [cont.]

State Space, Graphs, Paths, Solutions and Optimal Solutions

Initial state, actions, and transition model implicitly define the state space of the problem

- the state space forms a directed graph (e.g. the Romania map)
 - typically too big to be created explicitly and be stored in full
- in a state space graph, each state occurs only once
- a path is a sequence of states connected by actions
- a solution is a path from the initial state to a goal state
- an optimal solution is a path with minimum total cost

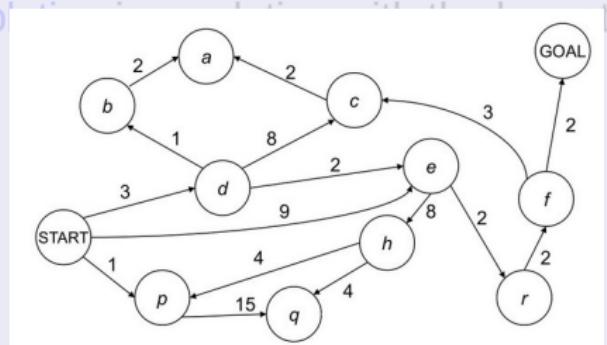


Well-defined problems and solutions [cont.]

State Space, Graphs, Paths, Solutions and Optimal Solutions

Initial state, actions, and transition model implicitly define the state space of the problem

- the state space forms a directed graph (e.g. the Romania map)
 - typically too big to be created explicitly and be stored in full
- in a state space graph, each state occurs only once
- a path is a sequence of states connected by actions
- a solution is a path from the initial state to a goal state
- an optimal solution is a path with minimum total cost

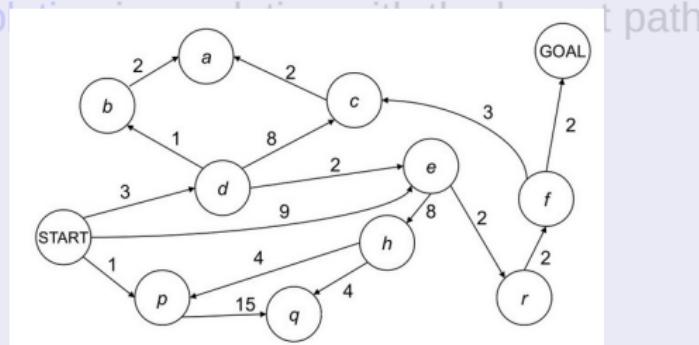


Well-defined problems and solutions [cont.]

State Space, Graphs, Paths, Solutions and Optimal Solutions

Initial state, actions, and transition model implicitly define the state space of the problem

- the state space forms a directed graph (e.g. the Romania map)
 - typically too big to be created explicitly and be stored in full
- in a state space graph, each state occurs only once
- a path is a sequence of states connected by actions
- a solution is a path from the initial state to a goal state
- an optimal solution is a path with minimum total cost



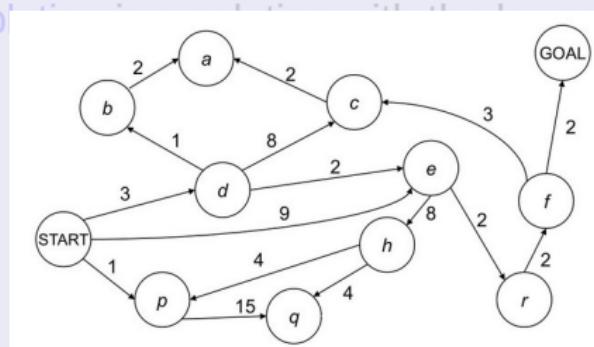
Well-defined problems and solutions [cont.]

State Space, Graphs, Paths, Solutions and Optimal Solutions

Initial state, actions, and transition model implicitly define the state space of the problem

- the state space forms a directed graph (e.g. the Romania map)
 - typically too big to be created explicitly and be stored in full
- in a state space graph, each state occurs only once
- a path is a sequence of states connected by actions
- a solution is a path from the initial state to a goal state

an optimal solution is a path with minimum cost

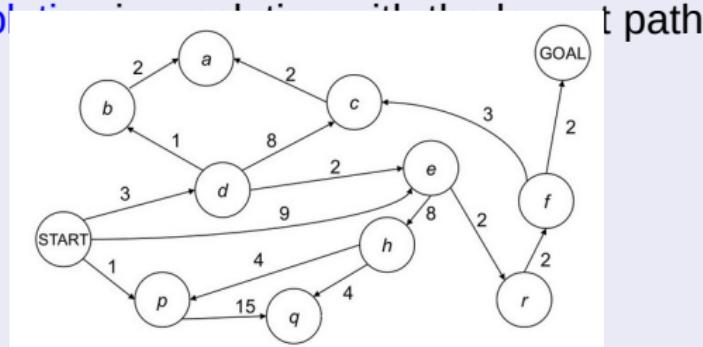


Well-defined problems and solutions [cont.]

State Space, Graphs, Paths, Solutions and Optimal Solutions

Initial state, actions, and transition model implicitly define the state space of the problem

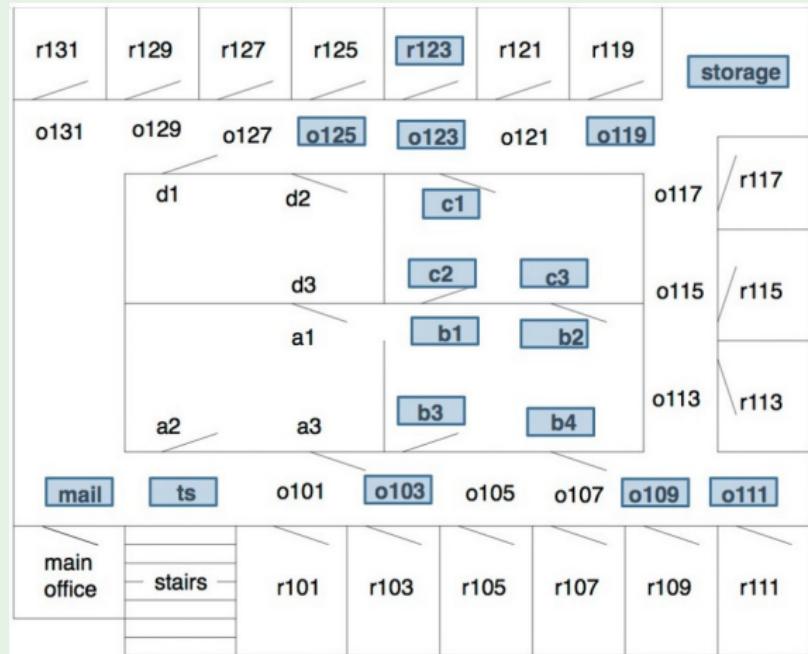
- the state space forms a directed graph (e.g. the Romania map)
 - typically too big to be created explicitly and be stored in full
- in a state space graph, each state occurs only once
- a path is a sequence of states connected by actions
- a solution is a path from the initial state to a goal state
- an optimal solution is a path with minimum total cost



Example: Path finding for a Delivery Robot

Task: move from o103 to r123

- States
 - Initial state
 - Goal state
 - State graph
 - Optimal solution

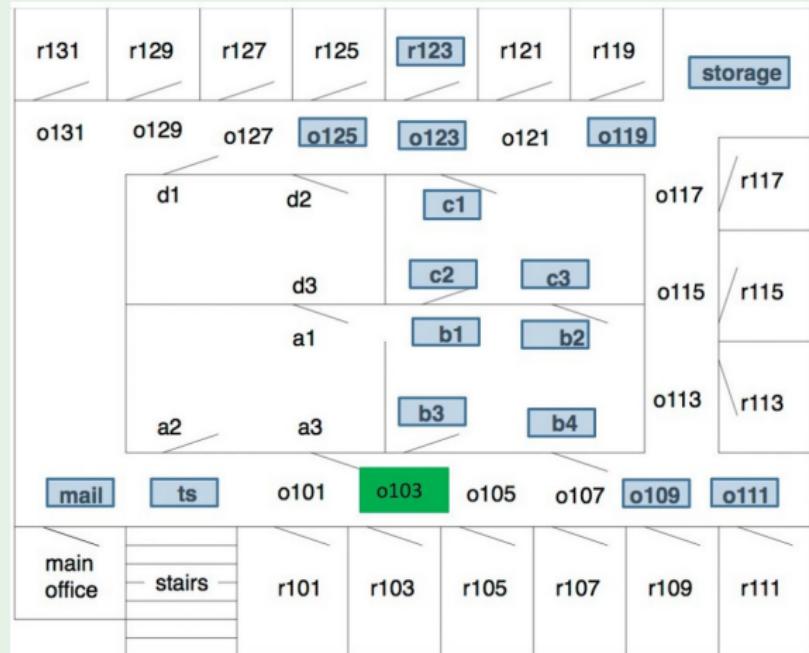


(Courtesy of Maria Simi, UniPi)

Example: Path finding for a Delivery Robot

Task: move from o103 to r123

- States
- Initial state
- Goal state
- State graph
- Optimal solution

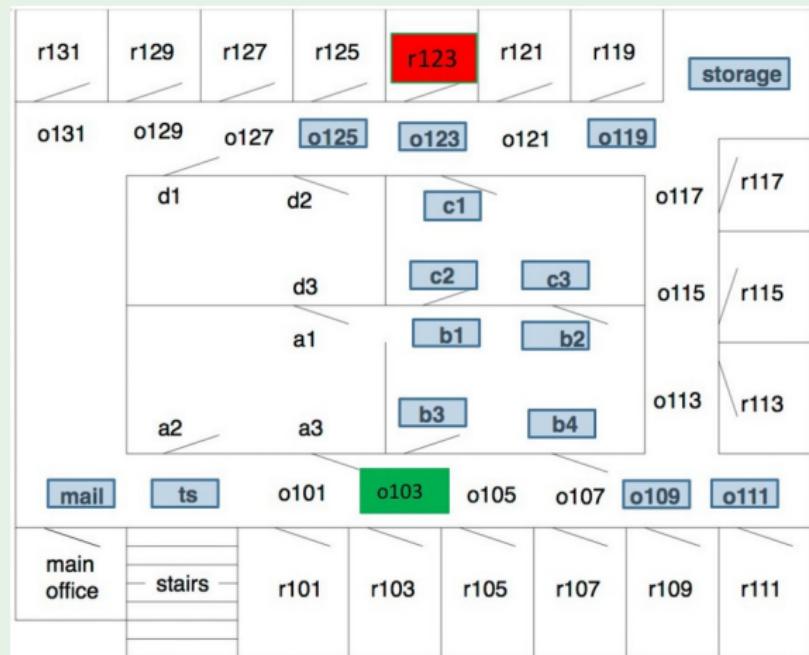


(Courtesy of Maria Simi, UniPi)

Example: Path finding for a Delivery Robot

Task: move from o103 to r123

- States
- Initial state
- Goal state
- State graph
- Optimal solution

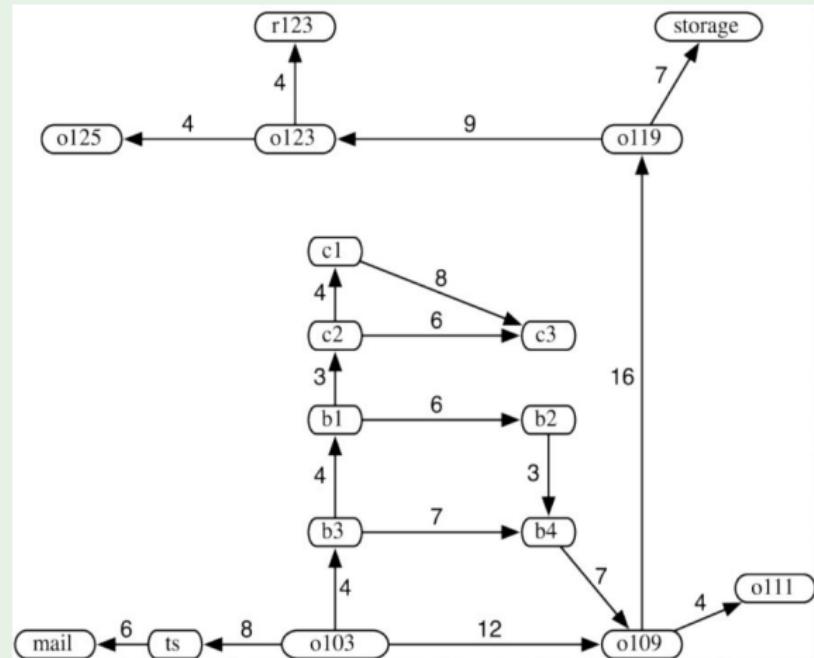


(Courtesy of Maria Simi, UniPi)

Example: Path finding for a Delivery Robot

Task: move from o103 to r123

- States
- Initial state
- Goal state
- State graph
- Optimal solution

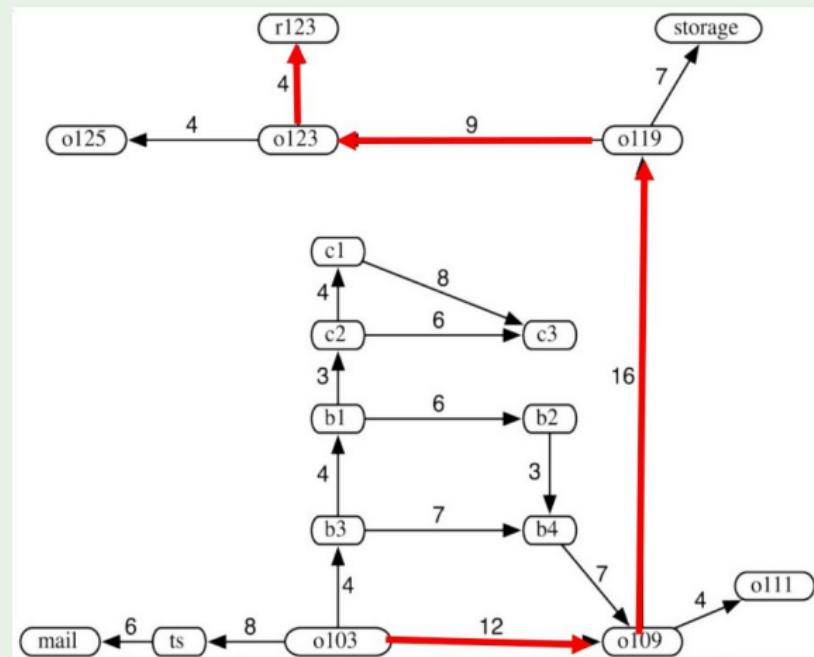


(Courtesy of Maria Simi, UniPi)

Example: Path finding for a Delivery Robot

Task: move from o103 to r123

- States
- Initial state
- Goal state
- State graph
- Optimal solution



(Courtesy of Maria Simi, UniPi)

Well-defined problems and solutions [cont.]

Abstraction

Problem formulations are **models** of reality (i.e. **abstract**)

- **descriptions**) real world is absurdly complex
==> state space must be **abstracted** for problem solving
lots of details removed because irrelevant to the problem
 - Ex: exact position, "turn steering wheel to the left by 20 degree", ...
- abstraction: the process of removing detail from representations
 - abstract state** represents many real states
 - abstract action** represents complex combination of real actions

valid abstraction: can expand any abstract solution into a solution in the detailed world

useful abstraction: if carrying out each of the actions in the solution is easier than in the original problem

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

Well-defined problems and solutions [cont.]

Abstraction

Problem formulations are **models** of reality (i.e. **abstract**

- **descriptions**) real world is absurdly complex

==> state space must be **abstracted** for problem solving

lots of details removed because irrelevant to the problem

- Ex: exact position, "turn steering wheel to the left by 20 degree", ...
- abstraction: the process of removing detail from representations
 - abstract state represents many real states
 - abstract action represents complex combination of real actions

valid abstraction: can expand any abstract solution into a solution in the detailed world

useful abstraction: if carrying out each of the actions in the solution is easier than in the original problem

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

Well-defined problems and solutions [cont.]

Abstraction

Problem formulations are **models** of reality (i.e. **abstract**)

- **descriptions**) real world is absurdly complex

==> state space must be **abstracted** for problem solving

~~lots~~ of details removed because irrelevant to the problem

- Ex: **exact position**, “turn steering wheel to the left by 20 degree”, ...

- abstraction: the process of removing detail from representations

abstract state represents many real states

abstract action represents complex combination of real actions

valid abstraction: can expand any abstract solution into a solution in the detailed world

useful abstraction: if carrying out each of the actions in the solution is easier than in the original problem

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

Well-defined problems and solutions [cont.]

Abstraction

Problem formulations are **models** of reality (i.e. **abstract**)

- **descriptions**) real world is absurdly complex

==> state space must be **abstracted** for problem solving

~~lots~~ of details removed because irrelevant to the problem

- Ex: **exact position**, “turn steering wheel to the left by 20 degree”, ...
- **abstraction**: the process of removing detail from representations
 - abstract state** represents many real states
 - abstract action** represents complex combination of real actions

~~valid abstraction~~: can expand any abstract solution into a solution in the detailed world

~~useful abstraction~~: if carrying out each of the actions in the solution is easier than in the original problem

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

Well-defined problems and solutions [cont.]

Abstraction

Problem formulations are **models** of reality (i.e. **abstract**)

- **descriptions**) real world is absurdly complex

==> state space must be **abstracted** for problem solving

~~lots~~ of details removed because irrelevant to the problem

- Ex: **exact position**, “turn steering wheel to the left by 20 degree”, ...
- **abstraction**: the process of removing detail from representations
 - abstract state** represents many real states
 - abstract action** represents complex combination of real actions

valid abstraction: can expand any abstract solution into a solution in the detailed world

useful abstraction: if carrying out each of the actions in the solution is easier than in the original problem

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

Well-defined problems and solutions [cont.]

Abstraction

Problem formulations are **models** of reality (i.e. **abstract**)

- **descriptions**) real world is absurdly complex

==> state space must be **abstracted** for problem solving

~~lots~~ of details removed because irrelevant to the problem

- Ex: **exact position**, “turn steering wheel to the left by 20 degree”, ...
- **abstraction**: the process of removing detail from representations
 - abstract state** represents many real states
 - abstract action** represents complex combination of real actions

valid abstraction: can expand any abstract solution into a solution in the detailed world

useful abstraction: if carrying out each of the actions in the solution is easier than in the original problem

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

Well-defined problems and solutions [cont.]

Abstraction

Problem formulations are **models** of reality (i.e. **abstract**)

- **descriptions**) real world is absurdly complex

==> state space must be **abstracted** for problem solving

~~lots~~ of details removed because irrelevant to the problem

- Ex: **exact position**, “turn steering wheel to the left by 20 degree”, ...
- **abstraction**: the process of removing detail from representations
 - abstract state** represents many real states
 - abstract action** represents complex combination of real actions

valid abstraction: can expand any abstract solution into a solution in the detailed world

useful abstraction: if carrying out each of the actions in the solution is easier than in the original problem

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

Outline

1 Problem-Solving Agents

2 Example Problems

3 Search Generalities

4 Uninformed Search Strategies

- Breadth-First Search
- Uniform-cost Search

Depth-First Search

Depth-Limited Search &

Iterative Deepening

5

Informed Search Strategies

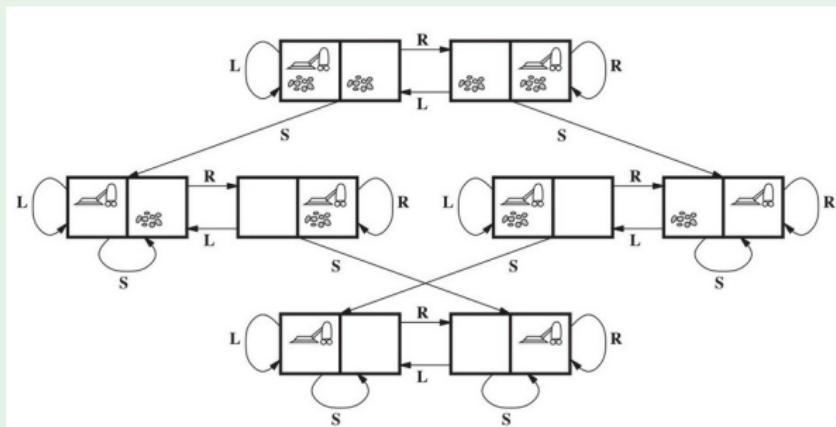
- Greedy Search

A* Search Heuristic

Functions

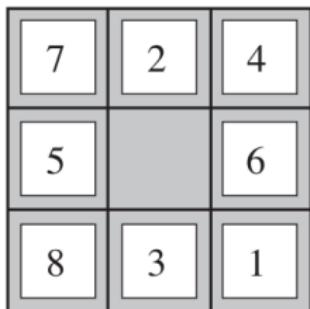
Toy Example: Simple Vacuum Cleaner

- **States:** 2 locations, each {clean, dirty}: $2 \cdot 2^2 = 8$
- **Initial State:** any
- **Actions:** {Left, Right, Suck}
- **Transition Model:** Left [Right] if A [B], Suck if clean \Rightarrow no effect
- **Goal Test:** check if squares are clean
- **Path Cost:** each step costs 1 \Rightarrow path cost is # of steps in path

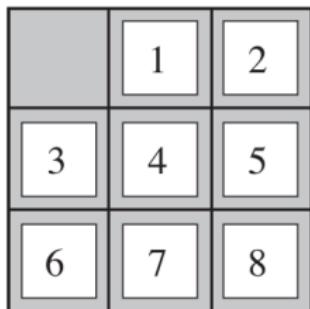


Toy Example: The 8-Puzzle

- **States:** Integer location of each tile $\Rightarrow 9!/2$ reachable
- **Initial State:** any
- **Actions:** moving *{Left, Right, Up, Down}* the empty space
- **Transition Model:** empty switched with the tile in target location
- **Goal Test:** checks state corresponds with goal configuration
- **Path Cost:** each step costs 1 \Rightarrow path cost is # of steps in path

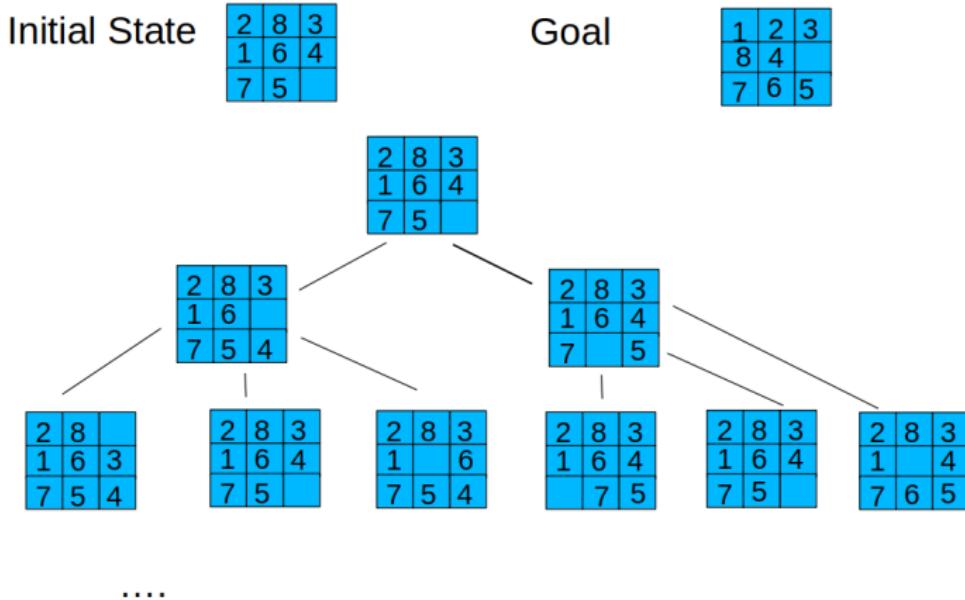


Start State



Goal State

Toy Example: The 8-Puzzle [cont.]



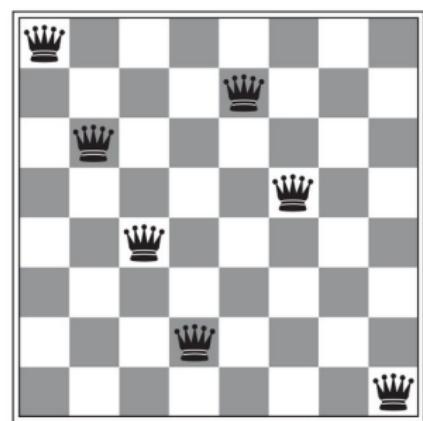
(Courtesy of Michela Milano, UniBO)

NP-complete: N-Puzzle ($N = k^2 - 1$): $N!/2$ reachable states

Toy Example: 8-Queens Problem

- **States:** any arrangement of 0 to 8 queens on the board
==> $64 \cdot 63 \cdot \dots \cdot 57 \approx 1.8 \cdot 10^{14}$ possible sequences
- **Initial State:** no queens on the board
- **Actions:** add a queen to any empty square
- **Transition Model:** returns the board with a queen added

Goal Test: 8 queen
queen Path Cost:

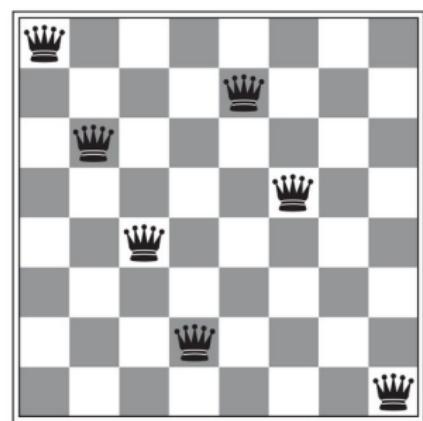


tacked by other

Toy Example: 8-Queens Problem

- **States:** any arrangement of 0 to 8 queens on the board
==> $64 \cdot 63 \cdot \dots \cdot 57 \approx 1.8 \cdot 10^{14}$ possible sequences
- **Initial State:** no queens on the board
- **Actions:** add a queen to any empty square
- **Transition Model:** returns the board with a queen added

Goal Test: 8 queen
queen Path Cost:

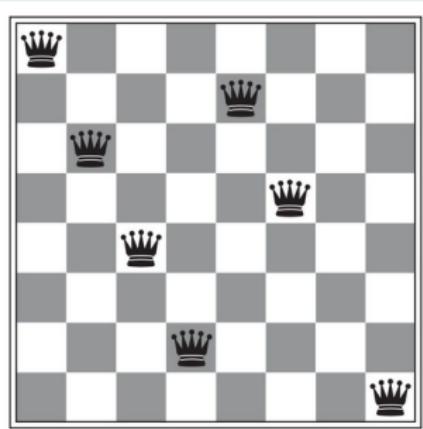


tacked by other

Toy Example: 8-Queens Problem (incremental)

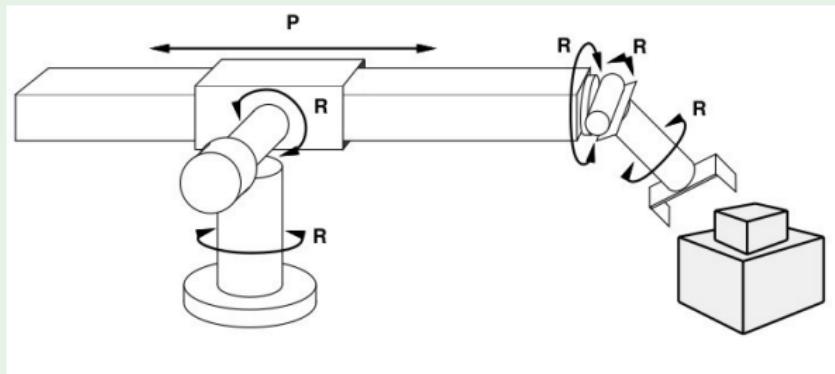
- **States:** $n \leq 8$ queens on board, one per column in the n leftmost columns, no queen attacking another
==> 2057 possible sequences
- **Actions:** Add a queen to any square in the leftmost empty column such that it is not attacked by any other queen.

...



Real-World Example: Robotic Assembly

- **States:** real-valued coordinates of robot joint angles, and of parts of the object to be assembled
- **Initial State:** any arm position and object configuration
- **Actions:** continuous motions of robot joints
- **Transition Model:** position resulting from motion
- **Goal Test:** complete assembly (without robot)
- **Path Cost:** time to execute



Other Real-World Examples

- Airline travel planning problems
- Touring problems
- VLSI layout problem
- Robot navigation
- Automatic assembly sequencing
- Protein design
- ...

Outline

- 1 Problem-Solving Agents
- 2 Example Problems
- 3 Search Generalities
- 4 Uninformed Search Strategies
 - Breadth-First Search
 - Uniform-cost SearchDepth-First Search
Depth-Limited Search &
Iterative Deepening
- 5 Informed Search Strategies
 - Greedy SearchA* Search

Searching for Solutions

Search: Generate sequences of actions.

- **Expansion:** one starts from a state, and applying the operators (or successor function) will generate new states
- **Search strategy:** at each step, choose which state to expand.
- **Search Tree:** It represents the expansion of all states starting from the initial state (the **root** of the tree)
- The **leaves** of the tree represent either:
 - states to expand
 - solutions
 - dead-ends

Tree Search Algorithms

Tree Search: Basic idea

- Off-line, simulated exploration of state space
 - start from initial state
 - pick one leaf node, and generate its successors (a.k.a. **expanding** a node) states
 - set of current leaves called **frontier** (a.k.a. **fringe**, **open list**) strategy for picking leaves critical (**search strategy**)
 - ends when either **a goal state is reached**, or **no more candidates to expand are available** (or time-out/memory-out occur)

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

 initialize the frontier using the initial state of *problem*

loop do

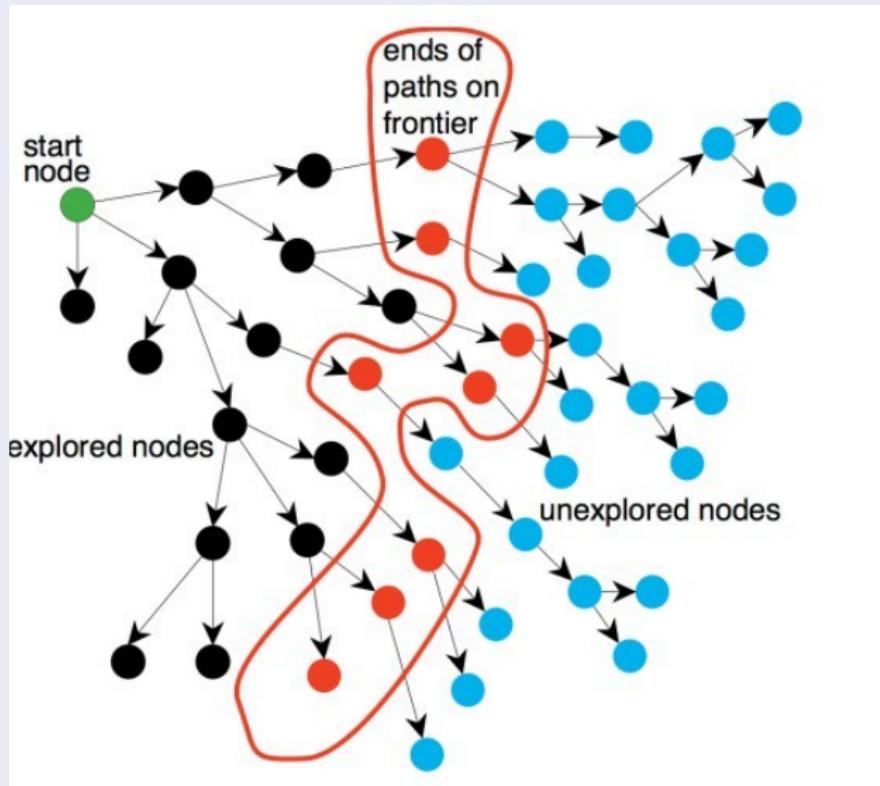
if the frontier is empty **then return** failure

 choose a leaf node and remove it from the frontier

if the node contains a goal state **then return** the corresponding solution

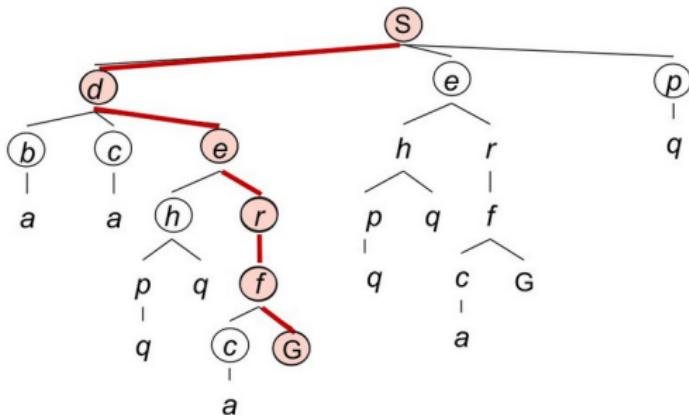
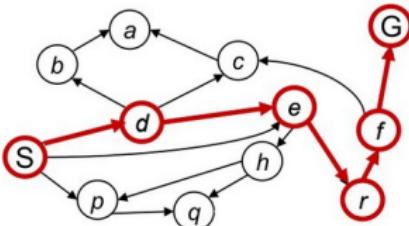
 expand the chosen node, adding the resulting nodes to the frontier

Tree Search Algorithms [cont.]



(Courtesy of Maria Simi, UniPi)

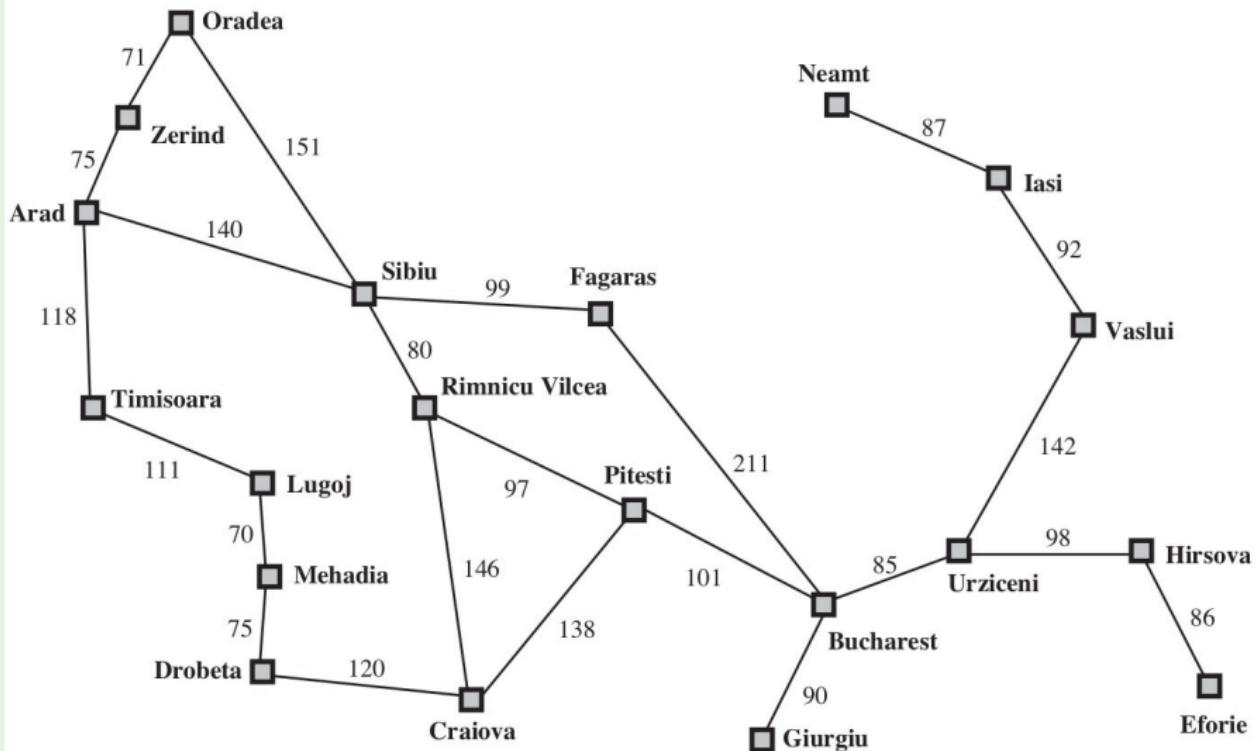
Tree-Search Example



~~s~~
~~s → d~~
s → e
s → p
s → d → b
s → d → c
~~s → d → e~~
s → d → e → h
~~s → d → e → r~~
~~s → d → e → r → f~~
s → d → e → r → f → c
~~s → d → e → r → f → G~~

Tree-Search Example: Trip from Arad to Bucharest

A simplified road map of part of Romania.



Tree-Search Example: Trip from Arad to Bucharest

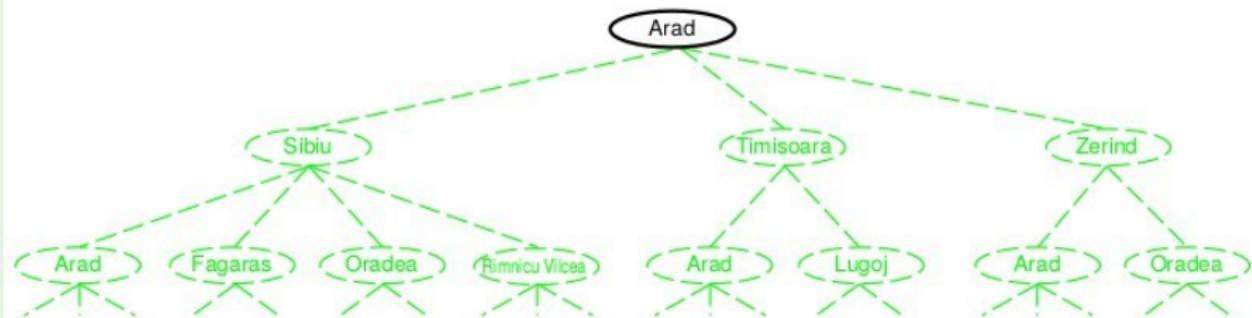
Expanding the search tree

- Initial state: {Arad}

Expand initial state \Rightarrow

- {Sibiu, Timisoara, Zerind}

Pick&expand Sibiu \Rightarrow



(Oc S. Russell & P. Norwig, AIMA)

Beware: Arad $\xrightarrow{1}$ Sibiu $\xrightarrow{1}$ Arad (repeated state \Rightarrow loopy)

Tree-Search Example: Trip from Arad to Bucharest

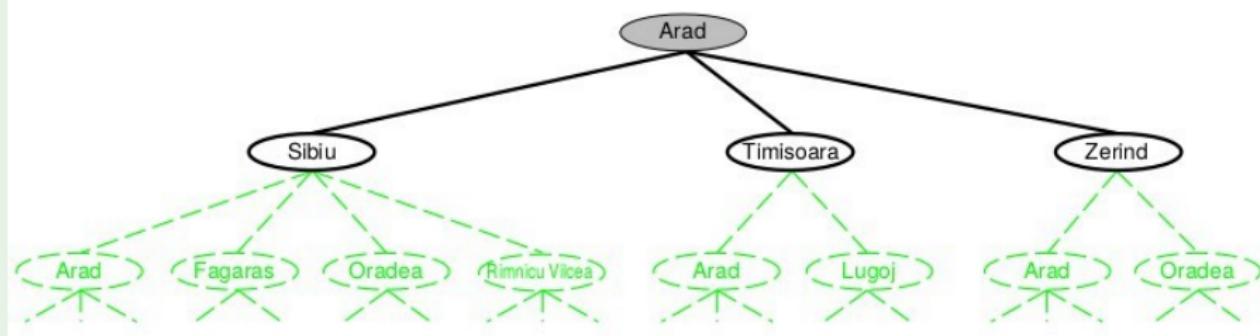
Expanding the search tree

- Initial state: {Arad}

Expand initial state \Rightarrow

- {Sibiu, Timisoara, Zerind}

Pick&expand Sibiu \Rightarrow



(Oc S. Russell & P. Norwig, AIMA)

Beware: Arad $\xrightarrow{1}$ Sibiu $\xrightarrow{1}$ Arad (repeated state \Rightarrow loop ~~path~~)
hi)

Tree-Search Example: Trip from Arad to Bucharest

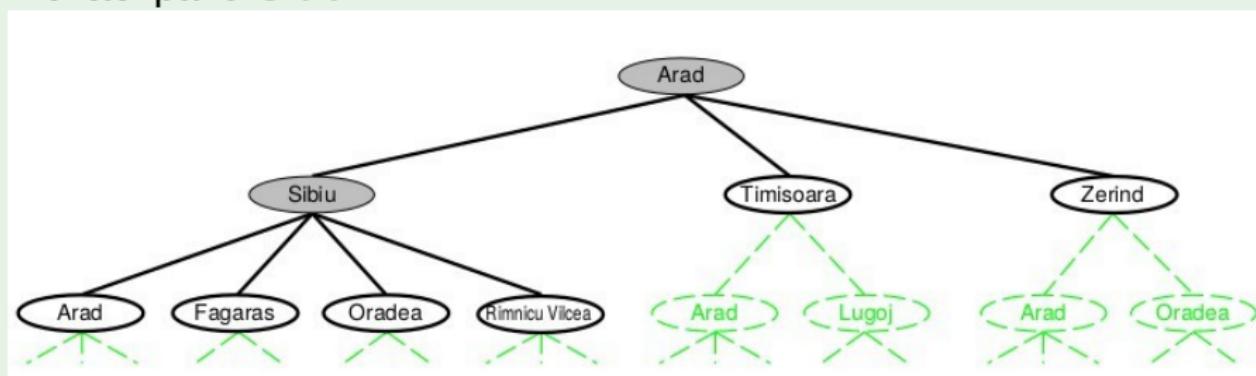
Expanding the search tree

- Initial state: {Arad}

Expand initial state ==>

- {Sibiu, Timisoara, Zerind}

Pick&expand Sibiu ==>



(Oc S. Russell & P. Norwig, AIMA)

Beware: Arad $\xrightarrow{1}$ Sibiu $\xrightarrow{1}$ Arad (repeated state ==> loopy path!)

Tree-Search Example: Trip from Arad to Bucharest

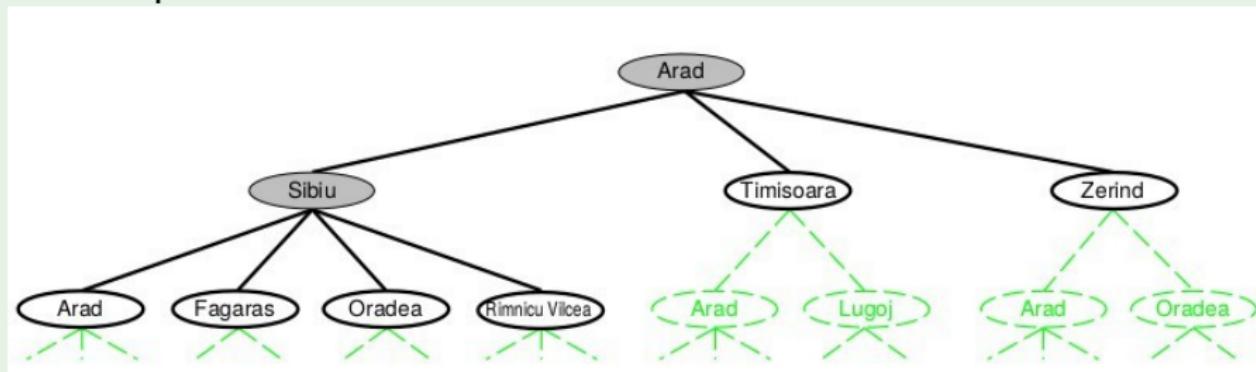
Expanding the search tree

- Initial state: {Arad}

Expand initial state ==>

- {Sibiu, Timisoara, Zerind}
-
-

Pick&expand Sibiu ==>



(Oc S. Russell & P. Norwig, AIMA)

Beware: Arad $\xrightarrow{1}$ Sibiu $\xrightarrow{1}$ Arad (repeated state ==> loopy path!)

Tree-Search Example: Trip from Arad to Bucharest

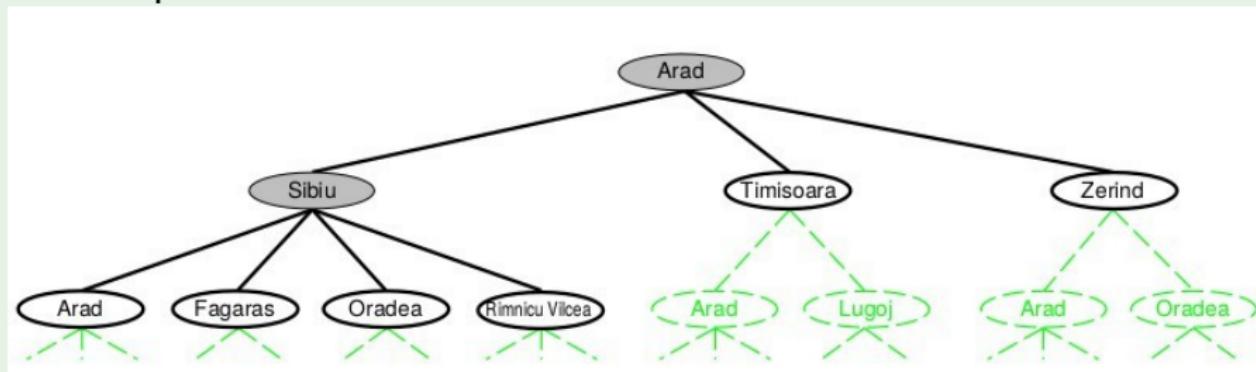
Expanding the search tree

- Initial state: {Arad}

Expand initial state ==>

- {Sibiu, Timisoara,
Zerind}
-

Pick&expand Sibiu ==>



(Oc S. Russell & P. Norwig, AIMA)

Beware: Arad $\xrightarrow{1}$ Sibiu $\xrightarrow{1}$ Arad (repeated state ==> loopy

path!)

Repeated states & Redundant Paths

- redundant paths occur when there is more than one way to get from one state to another
 - same state explored more than once
- Failure to detect repeated states can:
 - cause infinite loops
 - turn linear problem into exponential

(Oc S. Russell & P. Norwig, AIMA)

Moral: Algorithms that forget their history are doomed to repeat it!

Repeated states & Redundant Paths

- redundant paths occur when there is more than one way to get from one state to another
==> same state explored more than once
- Failure to detect repeated states can:
 - cause infinite loops
 - turn linear problem into exponential

(Oc S. Russell & P. Norwig, AIMA)

Moral: Algorithms that forget their history are doomed to repeat it!

Repeated states & Redundant Paths

- redundant paths occur when there is more than one way to get from one state to another
 ==> same state explored more than once
- Failure to detect repeated states can:
 - cause infinite loops
 - turn linear problem into exponential

(Oc S. Russell & P. Norwig, AIMA)

Moral: Algorithms that forget their history are doomed to repeat it!

Repeated states & Redundant Paths

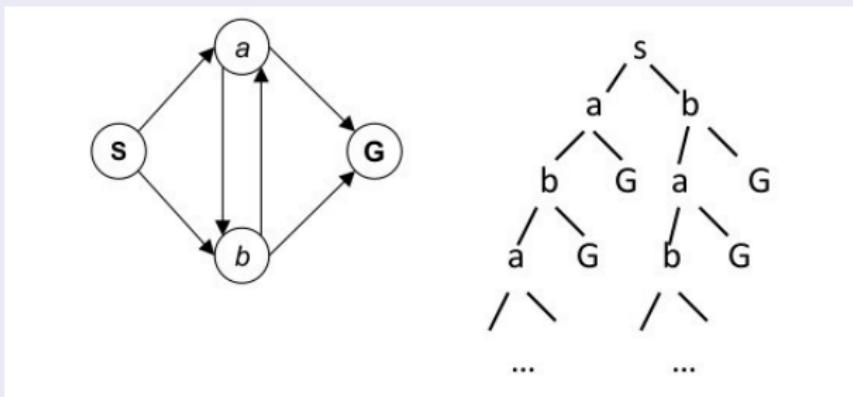
- redundant paths occur when there is more than one way to get from one state to another
==> same state explored more than once
- Failure to detect repeated states can:
 - cause infinite loops
 - turn linear problem into exponential

(Oc S. Russell & P. Norwig, AIMA)

Moral: Algorithms that forget their history are doomed to repeat it!

Repeated states & Redundant Paths

- redundant paths occur when there is more than one way to get from one state to another
==> same state explored more than once
- Failure to detect repeated states can:
 - cause infinite loops
 - turn linear problem into exponential

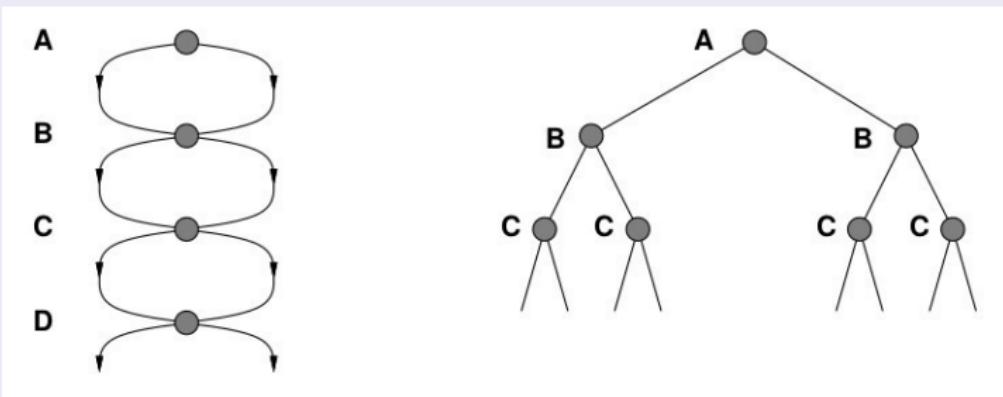


(Oc S. Russell & P. Norwig, AIMA)

Moral: Algorithms that forget their history are doomed to repeat it!

Repeated states & Redundant Paths

- redundant paths occur when there is more than one way to get from one state to another
==> same state explored more than once
- Failure to detect repeated states can:
 - cause infinite loops
 - turn linear problem into exponential

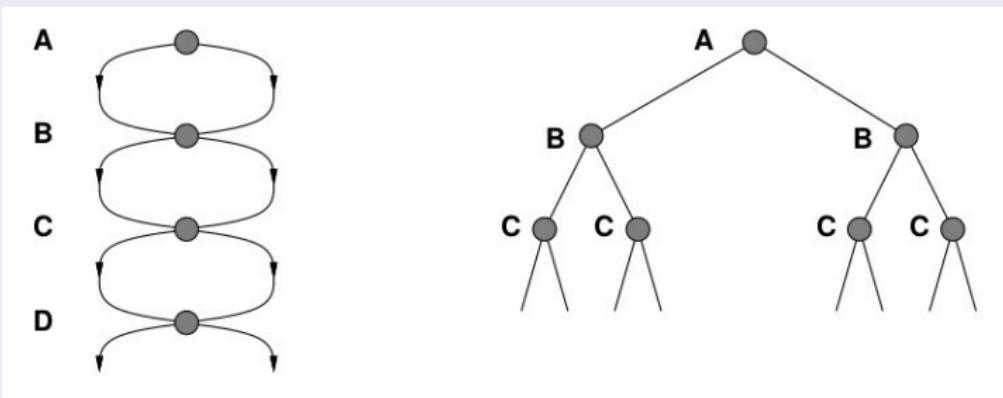


(Oc S. Russell & P. Norwig, AIMA)

Moral: Algorithms that forget their history are doomed to repeat if

Repeated states & Redundant Paths

- redundant paths occur when there is more than one way to get from one state to another
==> same state explored more than once
- Failure to detect repeated states can:
 - cause infinite loops
 - turn linear problem into exponential



(Oc S. Russell & P. Norwig, AIMA)

Moral: Algorithms that forget their history are doomed to repeat it!

Graph Search Algorithms

Graph Search: Basic idea

- add a data structure which remembers every expanded node
 - a.k.a. explored set or closed list
 - typically a hash table (access $O(1)$)
- do not expand a node if it already

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

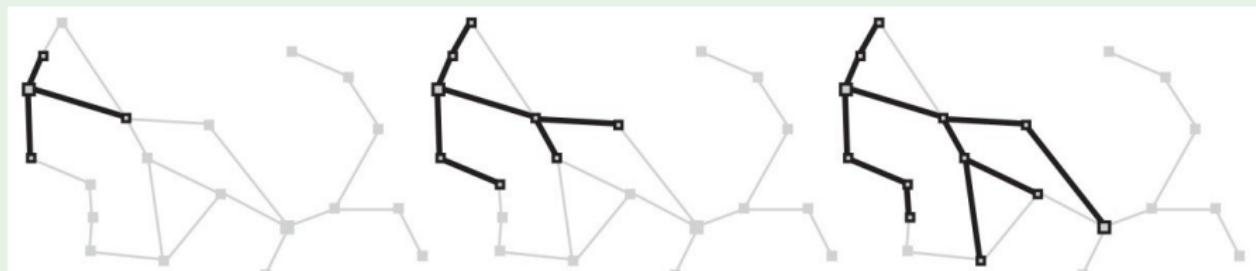
 expand the chosen node, adding the resulting nodes to the frontier

only if not in the frontier or explored set

Graph Search Algorithms: Example

Graph search on the Romania trip problem

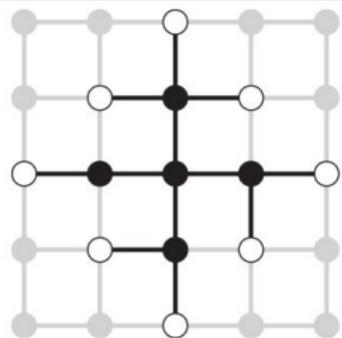
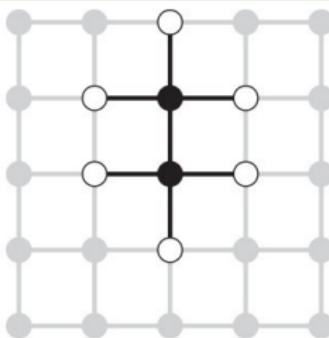
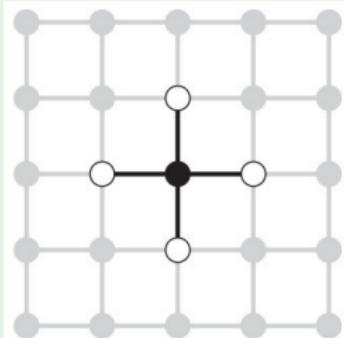
- (at each stage each path extended by one step) two states become dead-end



(Oc S. Russell & P. Norwig, AIMA)

Graph Search Algorithms: Example

Separation Property of graph search: the frontier separates the state-space graph into the explored region and the unexplored region



(Oc S. Russell & P. Norwig, AIMA)

Implementation: States vs. Nodes

- A state is a representation of a physical configuration
- A node is a data structure constituting part of a search tree

includes fields: state, parent, action, path cost $g(x)$

=⇒ node \neq state

It is easy to compute a child node from its parent

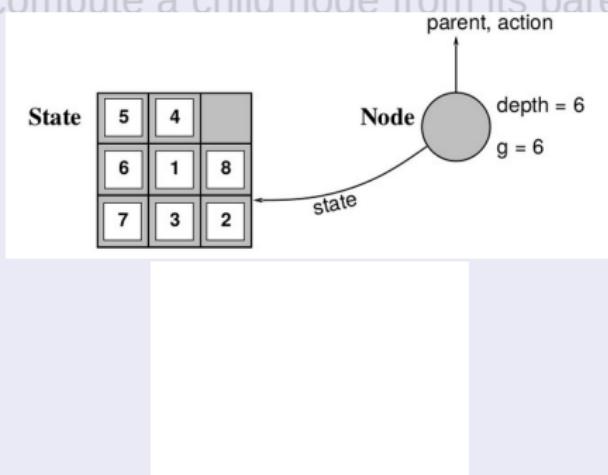
Implementation: States vs. Nodes

- A state is a representation of a physical configuration
- A node is a data structure constituting part of a search tree

includes fields: state, parent, action, path cost $g(x)$

\Rightarrow node \neq state

It is easy to compute a child node from its parent

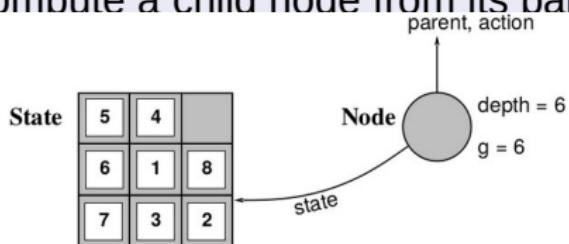


Implementation: States vs. Nodes

- A state is a representation of a physical configuration
- A node is a data structure constituting part of a search tree
 - includes fields: state, parent, action, path cost $g(x)$

=⇒ node ≠ state

It is easy to compute a child node from its parent



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)*

Implementation: Frontier and Explored

Frontier/Fringe

- Implemented as a Queue:
 - First-in-First-Out, FIFO (aka “queue”): $O(1)$ access
 - Last-in-First-Out, LIFO (aka “stack”): $O(1)$ access
 - Best-First-out (aka “priority queue”): $O(\log(n))$ access
- Two primitives:
 - **IsEmpty(QUEUE)**: returns true iff there are no more elements
 - **Pop(QUEUE)**: removes and returns the first element of the queue
 - **Insert(ELEMENT,QUEUE)**: inserts an element into queue

Explored

- Implemented as a Hash Table: $O(1)$ access
- Three primitives:
 - **IsThere(ELEMENT,HASH)**: returns true iff element is in the hash
 - **Insert(ELEMENT,HASH)**: inserts element into hash
- Choice of hash function critical for efficiency

Implementation: Frontier and Explored

Frontier/Fringe

- Implemented as a Queue:
 - First-in-First-Out, FIFO (aka “queue”): $O(1)$ access
 - Last-in-First-Out, LIFO (aka “stack”): $O(1)$ access
 - Best-First-out (aka “priority queue”): $O(\log(n))$ access
- Two primitives:
 - `IsEmpty(QUEUE)`: returns true iff there are no more elements
 - `Pop(QUEUE)`: removes and returns the first element of the queue
 - `Insert(ELEMENT,QUEUE)`: inserts an element into queue

Explored

- Implemented as a Hash Table: $O(1)$ access
- Three primitives:
 - `IsThere(ELEMENT,HASH)`: returns true iff element is in the hash
 - `Insert(ELEMENT,HASH)`: inserts element into hash
- Choice of hash function critical for efficiency

Implementation: Frontier and Explored

Frontier/Fringe

- Implemented as a Queue:
 - First-in-First-Out, FIFO (aka “queue”): $O(1)$ access
 - Last-in-First-Out, LIFO (aka “stack”): $O(1)$ access
 - Best-First-out (aka “priority queue”): $O(\log(n))$ access
- Two primitives:
 - `IsEmpty(QUEUE)`: returns true iff there are no more elements
 - `Pop(QUEUE)`: removes and returns the first element of the queue
 - `Insert(ELEMENT,QUEUE)`: inserts an element into queue

Explored

- Implemented as a Hash Table: $O(1)$ access
- Three primitives:
 - `IsThere(ELEMENT,HASH)`: returns true iff element is in the hash
 - `Insert(ELEMENT,HASH)`: inserts element into hash
- Choice of hash function critical for efficiency

Implementation: Frontier and Explored

Frontier/Fringe

- Implemented as a Queue:
 - First-in-First-Out, FIFO (aka “queue”): $O(1)$ access
 - Last-in-First-Out, LIFO (aka “stack”): $O(1)$ access
 - Best-First-out (aka “priority queue”): $O(\log(n))$ access
- Two primitives:
 - `IsEmpty(QUEUE)`: returns true iff there are no more elements
 - `Pop(QUEUE)`: removes and returns the first element of the queue
 - `Insert(ELEMENT,QUEUE)`: inserts an element into queue

Explored

- Implemented as a Hash Table: $O(1)$
- access Three primitives:
 - `IsThere(ELEMENT,HASH)`: returns true iff element is in the hash
 - `Insert(ELEMENT,HASH)`: inserts element into hash
- Choice of hash function critical for efficiency

Implementation: Frontier and Explored

Frontier/Fringe

- Implemented as a Queue:
 - First-in-First-Out, FIFO (aka “queue”): $O(1)$ access
 - Last-in-First-Out, LIFO (aka “stack”): $O(1)$ access
 - Best-First-out (aka “priority queue”): $O(\log(n))$ access
- Two primitives:
 - `IsEmpty(QUEUE)`: returns true iff there are no more elements
 - `Pop(QUEUE)`: removes and returns the first element of the queue
 - `Insert(ELEMENT,QUEUE)`: inserts an element into queue

Explored

- Implemented as a Hash Table: $O(1)$
- access Three primitives:
 - `IsThere(ELEMENT,HASH)`: returns true iff element is in the hash
 - `Insert(ELEMENT,HASH)`: inserts element into hash
- Choice of hash function critical for efficiency

Implementation: Frontier and Explored

Frontier/Fringe

- Implemented as a Queue:
 - First-in-First-Out, FIFO (aka “queue”): $O(1)$ access
 - Last-in-First-Out, LIFO (aka “stack”): $O(1)$ access
 - Best-First-out (aka “priority queue”): $O(\log(n))$ access
- Two primitives:
 - `IsEmpty(QUEUE)`: returns true iff there are no more elements
 - `Pop(QUEUE)`: removes and returns the first element of the queue
 - `Insert(ELEMENT,QUEUE)`: inserts an element into queue

Explored

- Implemented as a Hash Table: $O(1)$
- access Three primitives:
 - `IsThere(ELEMENT,HASH)`: returns true iff element is in the hash
 - `Insert(ELEMENT,HASH)`: inserts element into hash
- Choice of hash function critical for efficiency

Implementation: Frontier and Explored

Frontier/Fringe

- Implemented as a Queue:
 - First-in-First-Out, FIFO (aka “queue”): $O(1)$ access
 - Last-in-First-Out, LIFO (aka “stack”): $O(1)$ access
 - Best-First-out (aka “priority queue”): $O(\log(n))$ access
- Two primitives:
 - `IsEmpty(QUEUE)`: returns true iff there are no more elements
 - `Pop(QUEUE)`: removes and returns the first element of the queue
 - `Insert(ELEMENT,QUEUE)`: inserts an element into queue

Explored

- Implemented as a Hash Table: $O(1)$
- access Three primitives:
 - `IsThere(ELEMENT,HASH)`: returns true iff element is in the hash
 - `Insert(ELEMENT,HASH)`: inserts element into hash
- Choice of hash function critical for efficiency

Uninformed vs. Informed Search Strategies

Strategies: Two possibilities

- Uninformed strategies (a.k.a. blind strategies)
 - do not use any domain knowledge
 - apply rules arbitrarily and do an exhaustive search strategy
==> impractical for some complex problems.
- Informed strategies
 - use domain knowledge
 - apply rules following heuristics (driven by domain knowledge)
==> practical for many complex problems.

Uninformed vs. Informed Search Strategies

Strategies: Two possibilities

- **Uninformed** strategies (a.k.a. **blind** strategies)
 - do not use any domain knowledge
 - apply rules arbitrarily and do an exhaustive search strategy
==> impractical for some complex problems.
- **Informed** strategies
 - use domain knowledge
 - apply rules following heuristics (driven by domain knowledge)
==> practical for many complex problems.

Uninformed vs. Informed Search Strategies

Strategies: Two possibilities

- **Uninformed** strategies (a.k.a. **blind** strategies)
 - do not use any domain knowledge
 - apply rules arbitrarily and do an exhaustive search strategy
==> impractical for some complex problems.
- **Informed** strategies
 - use domain knowledge
 - apply rules following heuristics (driven by domain knowledge)
==> practical for many complex problems.

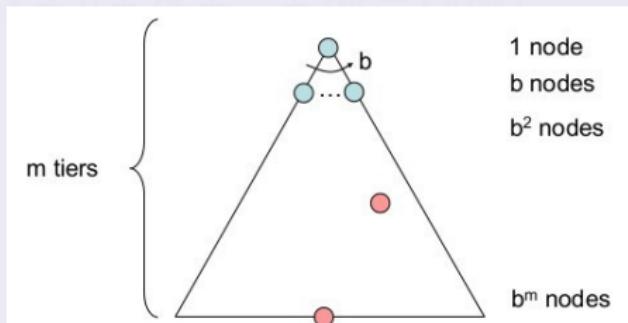
Outline

- 1 [Problem-Solving Agents](#)
- 2 [Example Problems](#)
- 3 [Search Generalities](#)
- 4 [Uninformed Search Strategies](#)
 - Breadth-First [Search](#)
 - Uniform-cost [Search](#)
 - [Depth-First Search](#)
 - [Depth-Limited Search & Iterative Deepening](#)
- 5 [Informed Search Strategies](#)
 - Greedy [Search](#)
 - [A* Search Heuristic Functions](#)

Evaluating Search Strategies

- Strategies are evaluated along the following dimensions:
 - completeness: does it always find a solution if one exists?
 - time complexity: how many steps to find a solution?
 - space complexity: how much memory is needed?
 - optimality: does it always find a least-cost solution?
- Time and space complexity are measured in terms of
 - b : maximum branching factor of the search tree
 - d : depth of the least-cost solution
 - m : maximum depth of the state space (may be $+\infty$)

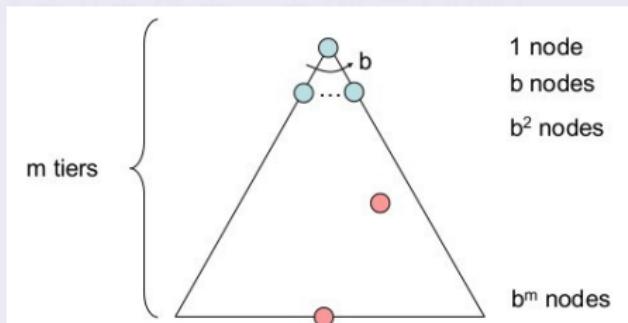
\Rightarrow # nodes = b^d (for finite b, d)



Evaluating Search Strategies

- Strategies are evaluated along the following dimensions:
 - completeness: does it always find a solution if one exists?
 - time complexity: how many steps to find a solution?
 - space complexity: how much memory is needed?
 - optimality: does it always find a least-cost solution?
- Time and space complexity are measured in terms of
 - b : maximum branching factor of the search tree
 - d : depth of the least-cost solution
 - m : maximum depth of the state space (may be $+\infty$)

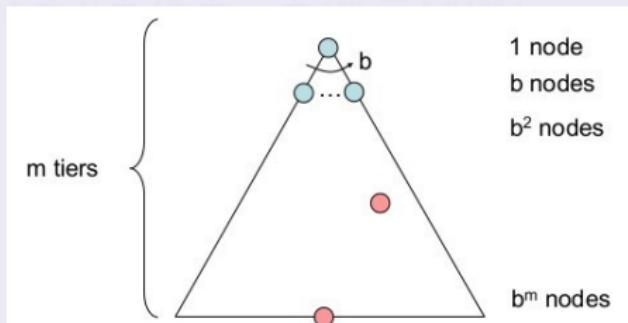
\Rightarrow # nodes: b^d (for finite b, d)



Evaluating Search Strategies

- Strategies are evaluated along the following dimensions:
 - completeness:** does it always find a solution if one exists?
 - time complexity:** how many steps to find a solution?
 - space complexity:** how much memory is needed?
 - optimality:** does it always find a least-cost solution?
- Time and space complexity are measured in terms of
 - b : maximum branching factor of the search tree
 - d : depth of the least-cost solution
 - m : maximum depth of the state space (may be $+\infty$)

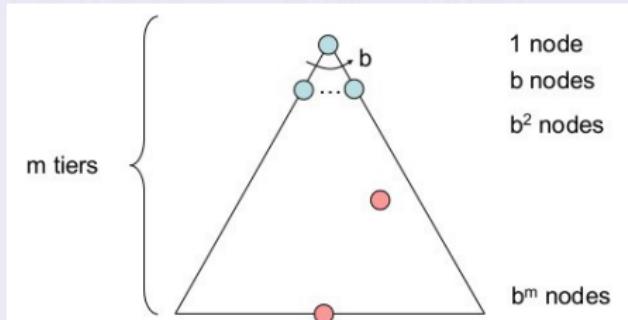
$\Rightarrow \# \text{nodes}$:



Evaluating Search Strategies

- Strategies are evaluated along the following dimensions:
 - completeness: does it always find a solution if one exists?
 - time complexity: how many steps to find a solution?
 - space complexity: how much memory is needed?
 - optimality: does it always find a least-cost solution?
- Time and space complexity are measured in terms of
 - b : maximum branching factor of the search tree
 - d : depth of the least-cost solution
 - m : maximum depth of the state space (may be $+\infty$)

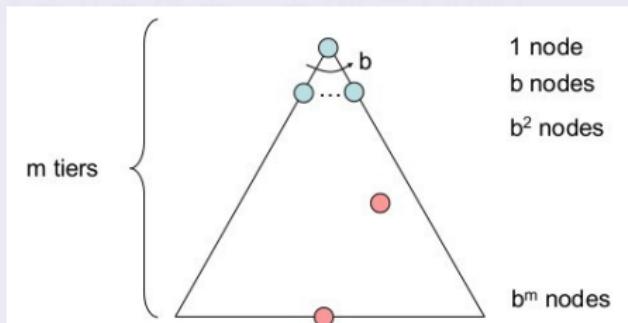
\Rightarrow # nodes:



Evaluating Search Strategies

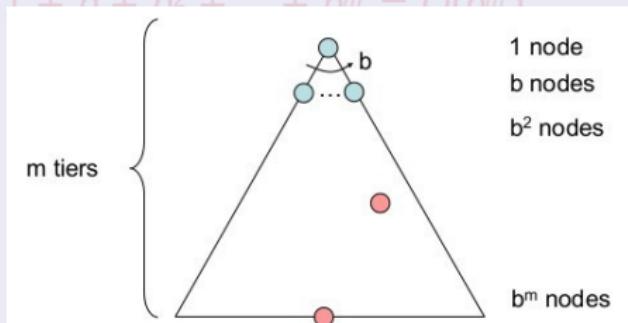
- Strategies are evaluated along the following dimensions:
 - completeness: does it always find a solution if one exists?
 - time complexity: how many steps to find a solution?
 - space complexity: how much memory is needed?
 - optimality: does it always find a least-cost solution?
- Time and space complexity are measured in terms of
 - b : maximum branching factor of the search tree
 - d : depth of the least-cost solution
 - m : maximum depth of the state space (may be $+\infty$)

\Rightarrow # nodes:



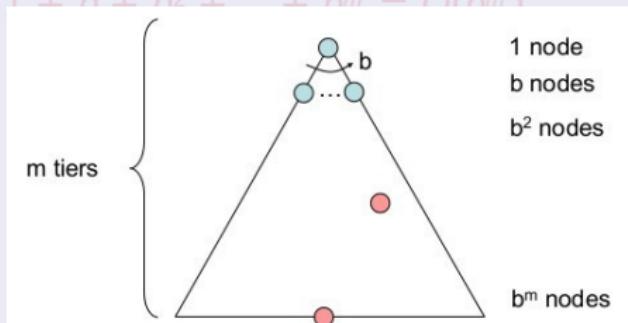
Evaluating Search Strategies

- Strategies are evaluated along the following dimensions:
 - completeness: does it always find a solution if one exists?
 - time complexity: how many steps to find a solution?
 - space complexity: how much memory is needed?
 - optimality: does it always find a least-cost solution?
- Time and space complexity are measured in terms of
 - b: maximum branching factor of the search tree d:
 - m: depth of the least-cost solution
- m : maximum depth of the state space (may be $+\infty$)
 $\Rightarrow \# \text{ nodes: } 1 + b + b^2 + \dots + b^m = O(b^m)$



Evaluating Search Strategies

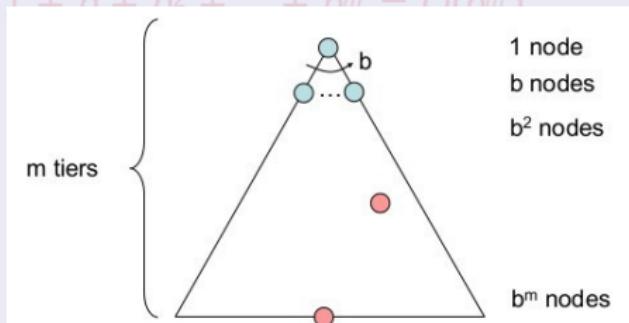
- Strategies are evaluated along the following dimensions:
 - completeness: does it always find a solution if one exists?
 - time complexity: how many steps to find a solution?
 - space complexity: how much memory is needed?
 - optimality: does it always find a least-cost solution?
- Time and space complexity are measured in terms of
 - b: maximum branching factor of the search tree d:
 - depth of the least-cost solution
 - m: maximum depth of the state space (may be $+\infty$)
 $\Rightarrow \# \text{ nodes: } 1 + b + b^2 + \dots + b^m = O(b^m)$



Evaluating Search Strategies

- Strategies are evaluated along the following dimensions:
 - completeness: does it always find a solution if one exists?
 - time complexity: how many steps to find a solution?
 - space complexity: how much memory is needed?
 - optimality: does it always find a least-cost solution?
- Time and space complexity are measured in terms of
 - b: maximum branching factor of the search tree d:
 - depth of the least-cost solution
 - m: maximum depth of the state space (may be $+\infty$)

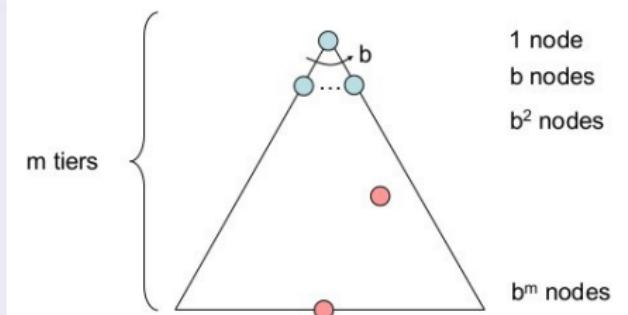
$\Rightarrow \# \text{ nodes: } 1 + b + b^2 + \dots + b^m = O(b^m)$



Evaluating Search Strategies

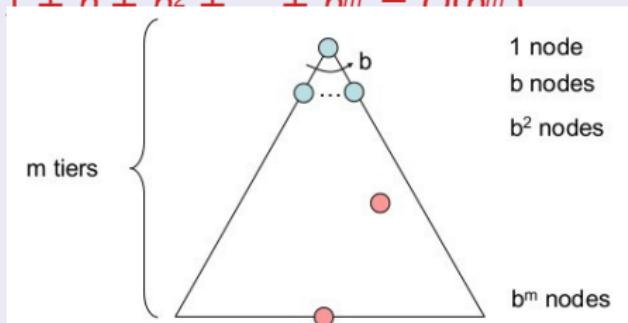
- Strategies are evaluated along the following dimensions:
 - completeness: does it always find a solution if one exists?
 - time complexity: how many steps to find a solution?
 - space complexity: how much memory is needed?
 - optimality: does it always find a least-cost solution?
- Time and space complexity are measured in terms of
 - b : maximum branching factor of the search tree d :
 - depth of the least-cost solution
- m : maximum depth of the state space (may be $+\infty$)

$\Rightarrow \# \text{ nodes: } 1 + b + b^2 + \dots + b^m = O(b^m)$



Evaluating Search Strategies

- Strategies are evaluated along the following dimensions:
 - completeness: does it always find a solution if one exists?
 - time complexity: how many steps to find a solution?
 - space complexity: how much memory is needed?
 - optimality: does it always find a least-cost solution?
 - Time and space complexity are measured in terms of
 - b : maximum branching factor of the search tree d :
 - depth of the least-cost solution
 - m : maximum depth of the state space (may be $+\infty$)
- $\Rightarrow \# \text{ nodes: } 1 + b + b^2 + \dots + b^m = O(b^m)$



Uninformed Search Strategies

Uninformed strategies

Use only the information available in the problem

- definition Different uninformed search strategies

Breadth-first search

- Uniform-cost search
- Depth-first search

Depth-limited search &

- Iterative-deepening search

Defined by **the access strategy of the frontier/fringe**

(i.e. the order of node expansion)

Outline

- 1 [Problem-Solving Agents](#)
- 2 [Example Problems](#)
- 3 [Search Generalities](#)
- 4 [Uninformed Search Strategies](#)
 - Breadth-First [Search](#)
 - Uniform-cost [Search](#)
 - [Depth-First Search](#)
 - [Depth-Limited Search & Iterative Deepening](#)
- 5 [Informed Search Strategies](#)
 - Greedy [Search](#)
 - [A* Search Heuristic Functions](#)

Breadth-First Search Strategy (BFS)

Breadth-First Search

- Idea: Expand first the shallowest unexpanded nodes
- Implementation: frontier/fringe implemented as a FIFO queue
 - novel successors pushed to the end of the queue

Breadth-First Search Strategy (BFS)

Breadth-First Search

- Idea: Expand first the shallowest unexpanded nodes
- Implementation: frontier/fringe implemented as a FIFO queue
 - novel successors pushed to the end of the queue

Breadth-First Search Strategy (BFS)

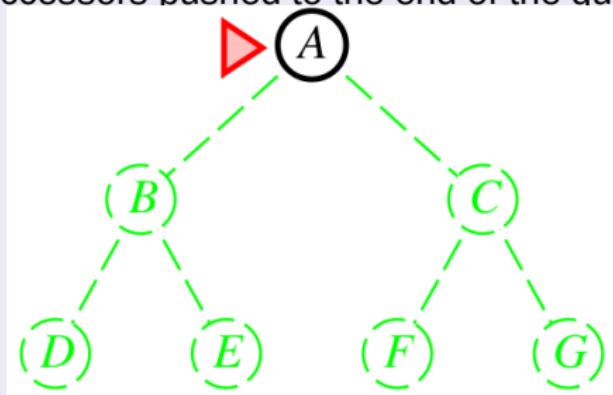
Breadth-First Search

- Idea: Expand first the shallowest unexpanded nodes
- Implementation: frontier/fringe implemented as a FIFO queue
==> novel successors pushed to the end of the queue

Breadth-First Search Strategy (BFS)

Breadth-First Search

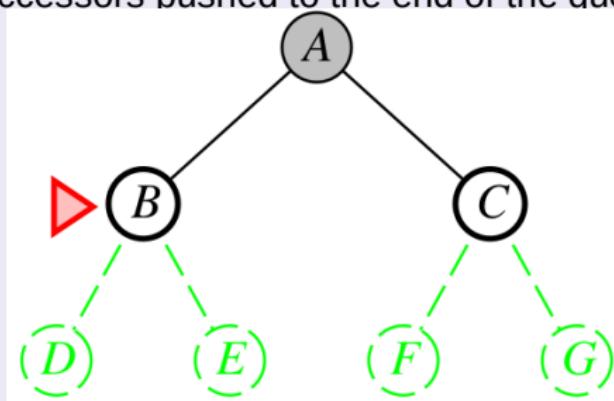
- Idea: Expand first the shallowest unexpanded nodes
- Implementation: frontier/fringe implemented as a FIFO queue
==> novel successors pushed to the end of the queue



Breadth-First Search Strategy (BFS)

Breadth-First Search

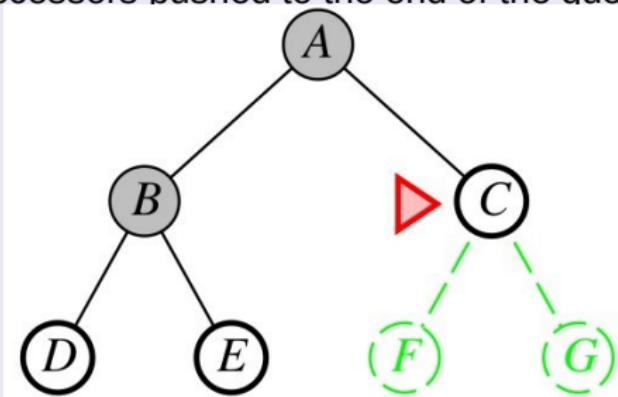
- Idea: Expand first the shallowest unexpanded nodes
- Implementation: frontier/fringe implemented as a FIFO queue
==> novel successors pushed to the end of the queue



Breadth-First Search Strategy (BFS)

Breadth-First Search

- Idea: Expand first the shallowest unexpanded nodes
- Implementation: frontier/fringe implemented as a FIFO queue
==> novel successors pushed to the end of the queue

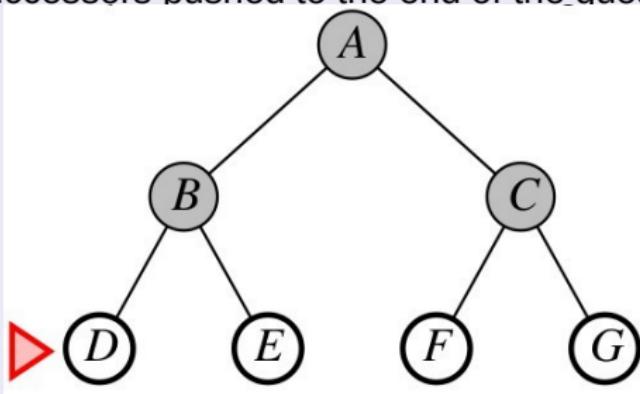


(Oc D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

Breadth-First Search Strategy (BFS)

Breadth-First Search

- Idea: Expand first the shallowest unexpanded nodes
- Implementation: frontier/fringe implemented as a FIFO queue
==> novel successors pushed to the end of the queue



Breadth-First Search Strategy (BFS) [cont.]

BFS, Graph version (Tree version without

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node  $\leftarrow$  a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier  $\leftarrow$  a FIFO queue with node as the only element
  explored  $\leftarrow$  an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node  $\leftarrow$  POP(frontier) /* chooses the shallowest node in frontier */
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child  $\leftarrow$  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  $\leftarrow$  INSERT(child, frontier)
```

(Oc S. Russell & P. Norwig, AIMA)

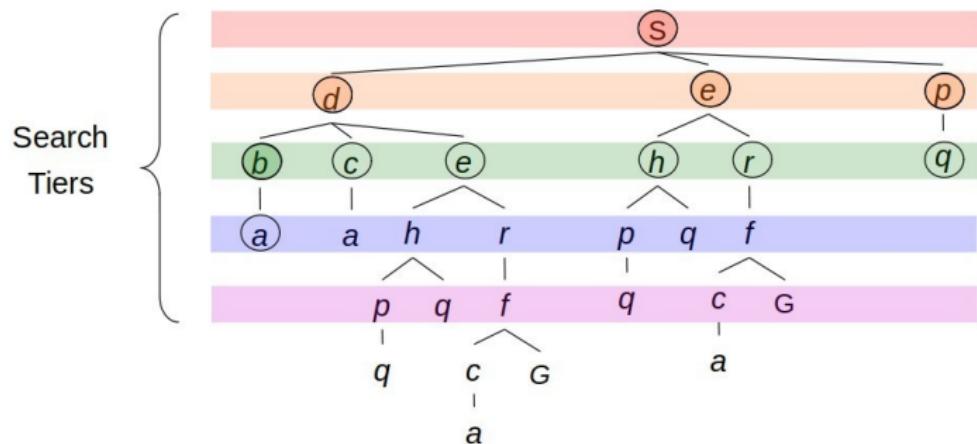
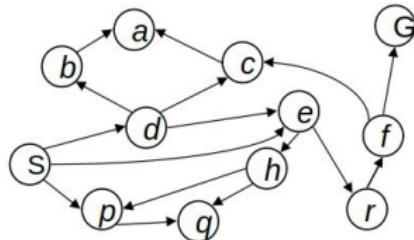
Note: the goal test is applied to each node **when it is generated**, rather than when **it is selected for expansion**
==> solution detected 1 layer earlier

Breadth-First Search: Tiers

State space is explored by tiers (tree)

Strategy: expand a shallowest node first

Implementation: Fringe is a FIFO queue



Breadth-First Search (BFS): Properties

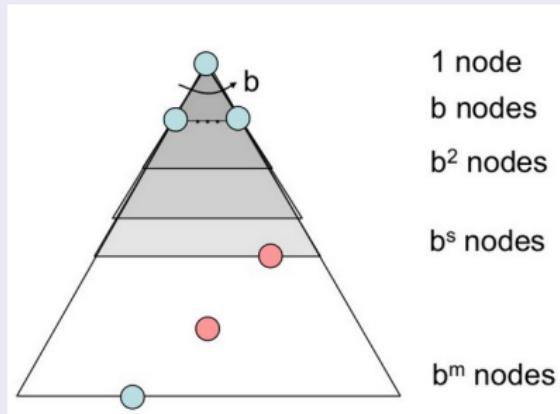
d : depth of shallowest

- solution How many steps?

How much memory?

Is it complete?

Is it optimal?



(Oc D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

Breadth-First Search (BFS): Properties

d : depth of shallowest

- solution How many steps?
processes all nodes above shallowest solution
- \Rightarrow takes $O(b^d)$ time

How much memory?

- max frontier size: b^d nodes
- $\Rightarrow O(b^d)$ memory size

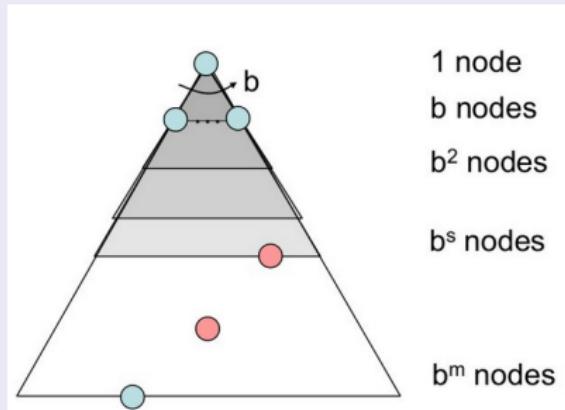
Is it complete?

- if solution exists, b^d finite
- \Rightarrow Yes

Is it optimal?

- if and only if all costs are 1
- \Rightarrow shallowest solution

Memory requirement is a major problem for breadth-first search



(Oc D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

Breadth-First Search (BFS): Properties

d : depth of shallowest

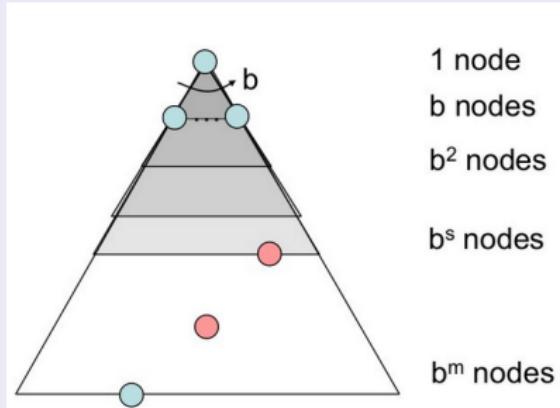
- solution How many steps?
processes all nodes above shallowest solution
 - \Rightarrow takes $O(b^d)$ time

How much memory?

max frontier size: b^d nodes

Is it complete?

Is it optimal?



(Oc D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

Breadth-First Search (BFS): Properties

d : depth of shallowest

- solution How many steps?
processes all nodes above shallowest solution
 - \Rightarrow takes $O(b^d)$ time

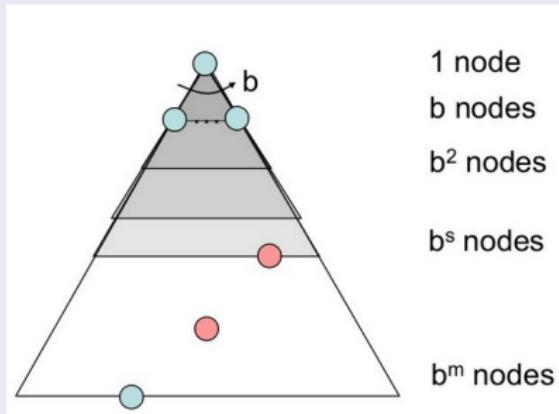
How much memory?

max frontier size: b^d nodes

- $\implies O(b^d)$ memory size

Is it complete?

Is it optimal?



(Oc D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

Breadth-First Search (BFS): Properties

d : depth of shallowest

- solution How many steps?
processes all nodes above shallowest solution
- $\Rightarrow O(b^d)$ time

How much memory?

- max frontier size: b^d nodes
- $\Rightarrow O(b^d)$ memory size

Is it complete?

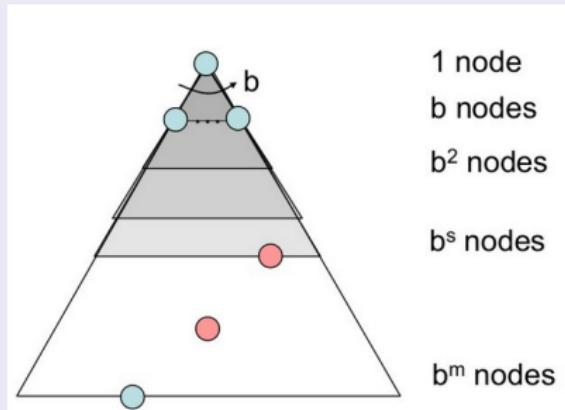
- if solution exists, b^d finite
- \Rightarrow Yes

Is it optimal?

if and only if all costs are 1

\Rightarrow shallowest solution

Memory requirement is a major problem for breadth-first search



(Oc D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

Breadth-First Search (BFS): Properties

d : depth of shallowest

- solution How many steps?
processes all nodes above shallowest solution
- $\Rightarrow O(b^d)$ time

How much memory?

- max frontier size: b^d nodes
- $\Rightarrow O(b^d)$ memory size

Is it complete?

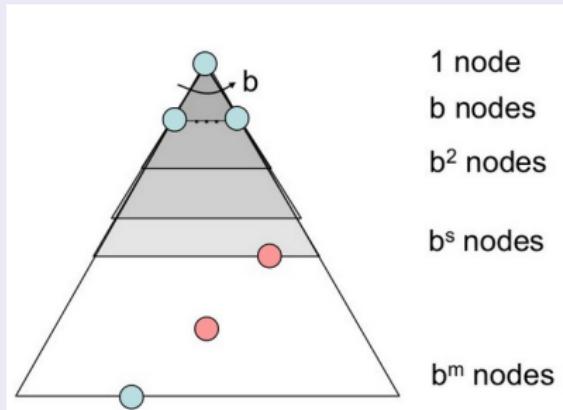
- if solution exists, b^d finite
- \Rightarrow Yes

Is it optimal?

if and only if all costs are 1

\Rightarrow shallowest solution

Memory requirement is a major problem for breadth-first search



(Oc D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

Breadth-First Search (BFS): Properties

d : depth of shallowest

- solution How many steps?
processes all nodes above shallowest solution
- $\Rightarrow O(b^d)$ time

How much memory?

- max frontier size: b^d nodes
- $\Rightarrow O(b^d)$ memory size

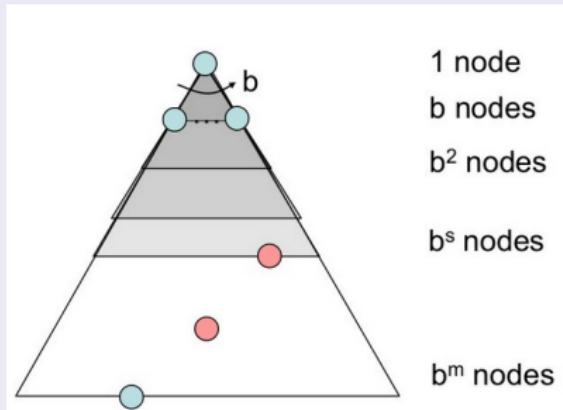
Is it complete?

- if solution exists, b^d finite
- \Rightarrow Yes

Is it optimal?

- if and only if all costs are 1
- \Rightarrow shallowest solution

Memory requirement is a major problem for breadth-first search



(Oc D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

Breadth-First Search (BFS): Properties

d : depth of shallowest

- solution How many steps?
processes all nodes above shallowest solution
- $\Rightarrow O(b^d)$ time

How much memory?

- max frontier size: b^d nodes
- $\Rightarrow O(b^d)$ memory size

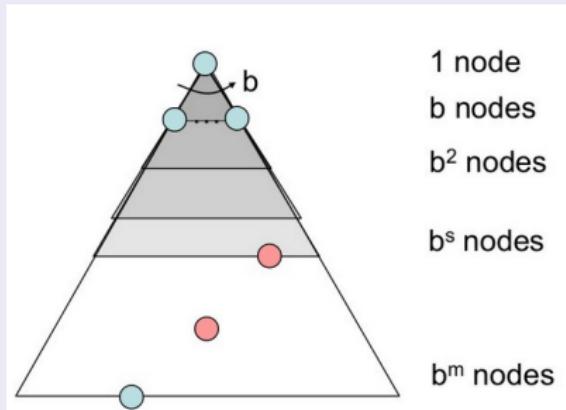
Is it complete?

- if solution exists, b^d finite
- \Rightarrow Yes

Is it optimal?

- if and only if all costs are 1
- \Rightarrow shallowest solution

Memory requirement is a major problem for breadth-first search



(Oc D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

Breadth-First Search (BFS): Properties

d : depth of shallowest

- solution How many steps?
processes all nodes above shallowest solution
- $\Rightarrow O(b^d)$ time

How much memory?

- max frontier size: b^d nodes
- $\Rightarrow O(b^d)$ memory size

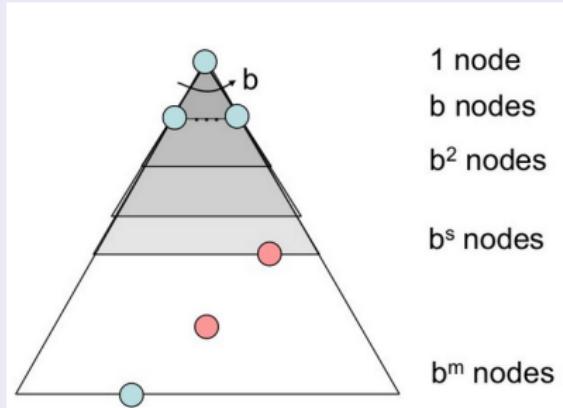
Is it complete?

- if solution exists, b^d finite
- \Rightarrow Yes

Is it optimal?

- if and only if all costs are 1
- \Rightarrow shallowest solution

Memory requirement is a major problem for breadth-first search



(Oc D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

Breadth-First Search (BFS): Time and Memory

- Assume:

- 1 million nodes generated per second
- 1 node requires 1000 bytes of storage
- branching factor $b = 10$

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

(Oc S. Russell & P. Norwig, AIMA)

Memory requirements bigger problem for BFS than execution time

Outline

- 1 [Problem-Solving Agents](#)
- 2 [Example Problems](#)
- 3 [Search Generalities](#)
- 4 [Uninformed Search Strategies](#)
 - Breadth-First [Search](#)
 - Uniform-cost [Search](#)
 - [Depth-First Search](#)
 - [Depth-Limited Search & Iterative Deepening](#)
- 5 [Informed Search Strategies](#)
 - Greedy [Search](#)
 - [A* Search Heuristic Functions](#)

Uniform-Cost Search Strategy (UCS)

Uniform-Cost Search

- Idea: Expand first the node with lowest path cost $g(n)$
- Implementation: frontier/fringe implemented as a priority queue ordered by $g()$
 - novel nearest successors pushed to the top of the queue
- similar to BFS if step costs are all equal

(Courtesy of Michela Milano, UniBO)

Uniform-Cost Search Strategy (UCS)

Uniform-Cost Search

- Idea: Expand first the node with lowest path cost $g(n)$
- Implementation: frontier/fringe implemented as a priority queue ordered by $g()$
 - novel nearest successors pushed to the top of the queue
- similar to BFS if step costs are all equal

(Courtesy of Michela Milano, UniBO)

Uniform-Cost Search Strategy (UCS)

Uniform-Cost Search

- Idea: Expand first the node with lowest path cost $g(n)$
- Implementation: frontier/fringe implemented as a priority queue ordered by $g()$
 - ⇒ novel nearest successors pushed to the top of the queue
- similar to BFS if step costs are all equal

Uniform-Cost Search Strategy (UCS)

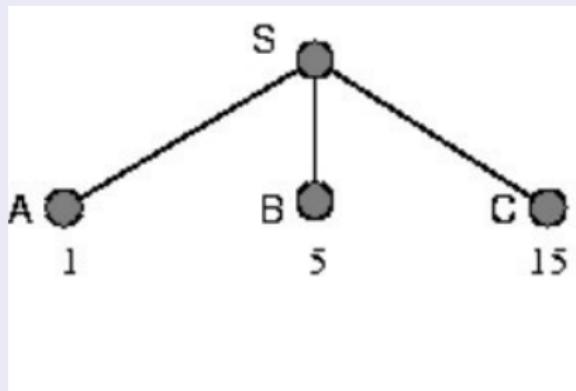
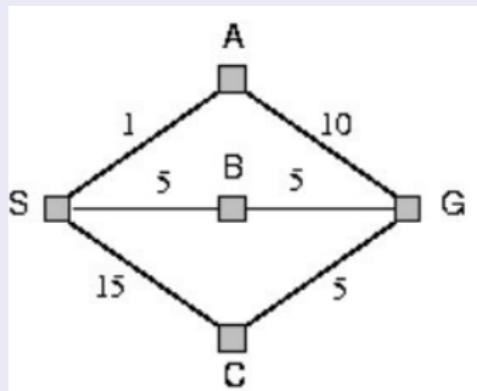
Uniform-Cost Search

- Idea: Expand first the node with lowest path cost $g(n)$
- Implementation: frontier/fringe implemented as a priority queue ordered by $g()$
 - ⇒ novel nearest successors pushed to the top of the queue
- similar to BFS if step costs are all equal

Uniform-Cost Search Strategy (UCS)

Uniform-Cost Search

- Idea: Expand first the node with lowest path cost $g(n)$
- Implementation: frontier/fringe implemented as a priority queue ordered by $g()$
 - novel nearest successors pushed to the top of the queue
- similar to BFS if step costs are all equal

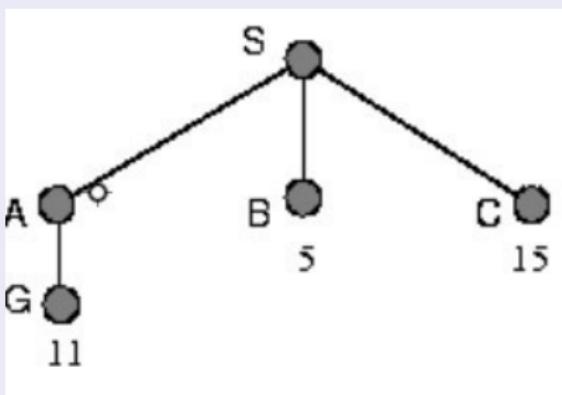
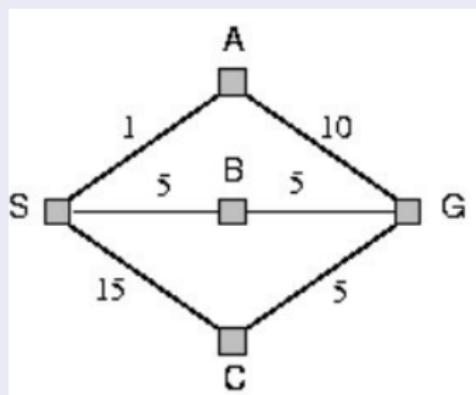


(Courtesy of Michela Milano, UniBO)

Uniform-Cost Search Strategy (UCS)

Uniform-Cost Search

- Idea: Expand first the node with lowest path cost $g(n)$
- Implementation: frontier/fringe implemented as a priority queue ordered by $g()$
 - novel nearest successors pushed to the top of the queue
- similar to BFS if step costs are all equal

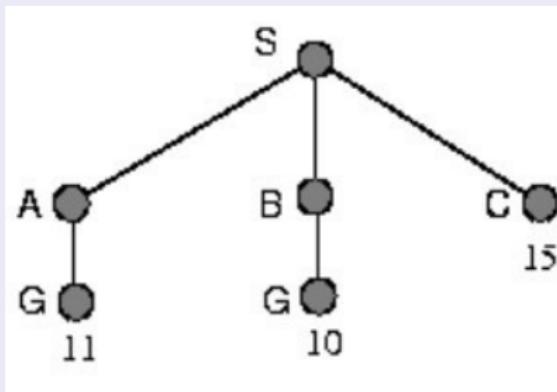
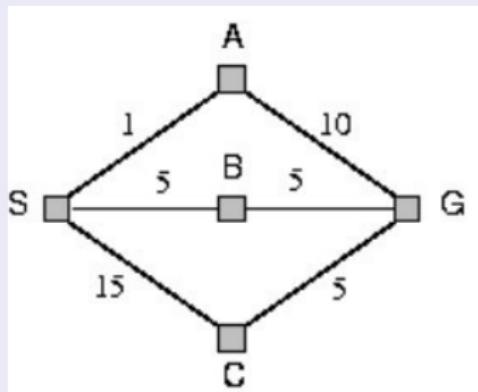


(Courtesy of Michela Milano, UniBO)

Uniform-Cost Search Strategy (UCS)

Uniform-Cost Search

- Idea: Expand first the node with lowest path cost $g(n)$
- Implementation: frontier/fringe implemented as a priority queue ordered by $g()$
==> novel nearest successors pushed to the top of the queue
- similar to BFS if step costs are all equal



(Courtesy of Michela Milano, UniBO)

Uniform-Cost Search Strategy (UCS) [cont.]

UCS, Graph version (Tree version without

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
    node  $\leftarrow$  a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
    frontier  $\leftarrow$  a priority queue ordered by PATH-COST, with node as the only element
    explored  $\leftarrow$  an empty set
    loop do
        if EMPTY?(frontier) then return failure
        node  $\leftarrow$  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  $\leftarrow$  CHILD-NODE(problem, node, action)
            if child.STATE is not in explored or frontier then
                frontier  $\leftarrow$  INSERT(child, frontier)
            else if child.STATE is in frontier with higher PATH-COST then
                replace that frontier node with child
```

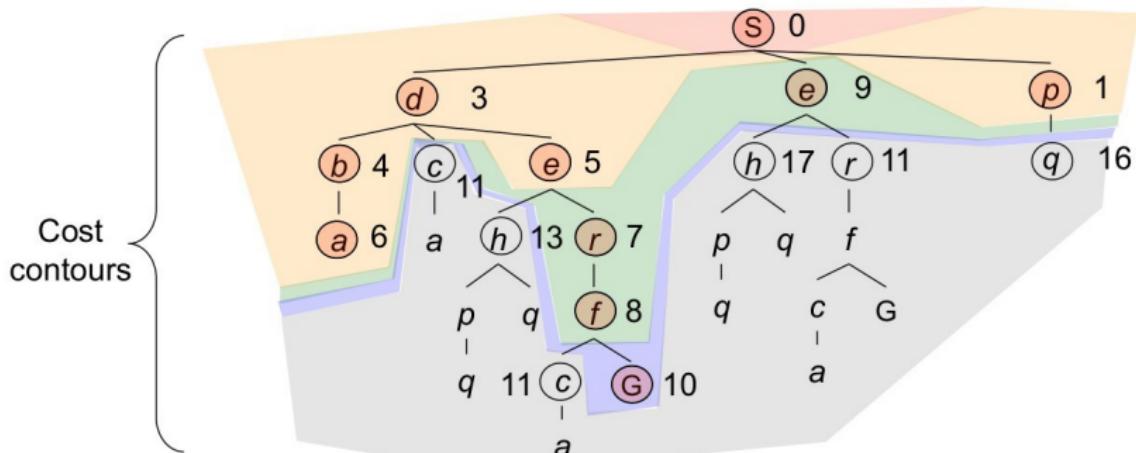
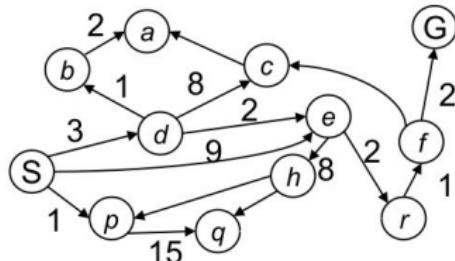
(Oc S. Russell & P. Norwig, AIMA)

- apply the goal test to a node when it is selected for expansion rather than when it is first generated
- replace in the frontier a node with same state but worse path cost
- avoid generating suboptimal paths

Uniform-Cost Search

Strategy: expand a cheapest node first:

Fringe is a priority queue
(priority: cumulative cost)



Uniform-Cost Search (UCS): Properties

C^* : cost of cheapest solution; c : minimum arc cost

==> $1 + \lceil C^*/c \rceil$ "effective depth" How many steps?

processes all nodes costing less than cheapest solution

•=> takes $O(b^{1+LC^*/c})$ time

How much memory?

max frontier size: $b^{1+LC^*/c}$

==> $O(b^{1+L^2/c})$ memory size

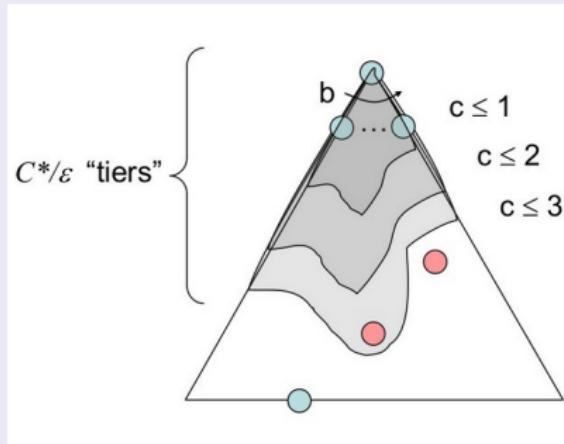
Is it complete?

if solution exists, finite cost

==> Yes

• Is it optimal?

• Yes



(Oc D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

Memory requirement is a major problem also for uniform-cost search

Uniform-Cost Search (UCS): Properties

C^* : cost of cheapest solution; c : minimum arc cost

- $1 + \lceil C^*/c \rceil$ “effective depth” How many steps? processes all nodes costing less than cheapest solution
- takes $O(b^{1+LC^*/c})$ time

How much memory?

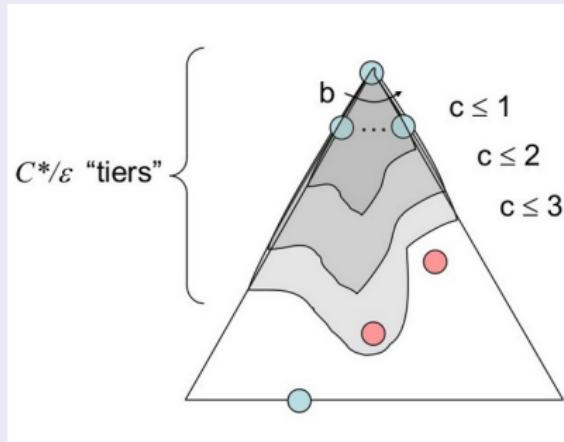
max frontier size: $b^{1+LC^*/c}$
⇒ $O(b^{1+LC^*/c})$ memory size

Is it complete?

if solution exists, finite cost

⇒ Yes

- Is it optimal?
 - Yes



(Oc D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

Memory requirement is a major problem also for uniform-cost search

Uniform-Cost Search (UCS): Properties

C^* : cost of cheapest solution; c : minimum arc cost

- $1 + \lceil C^*/c \rceil$ “effective depth” How many steps? processes all nodes costing less than cheapest solution
- \Rightarrow takes $O(b^{1+LC^*/c})$ time

How much memory?

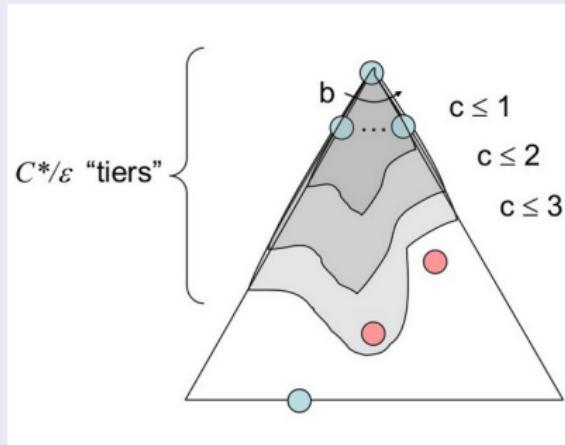
max frontier size: $b^{1+LC^*/c}$
 $\Rightarrow O(b^{1+LC^*/c})$ memory size

Is it complete?

if solution exists, finite cost

\Rightarrow Yes

- Is it optimal?
 - Yes



(Oc D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

Memory requirement is a major problem also for uniform-cost search

Uniform-Cost Search (UCS): Properties

C^* : cost of cheapest solution; c : minimum arc cost

- $1 + \lceil C^*/c \rceil$ “effective depth” How many steps? processes all nodes costing less than cheapest solution
- \Rightarrow takes $O(b^{1+LC^*/c})$ time

How much memory?

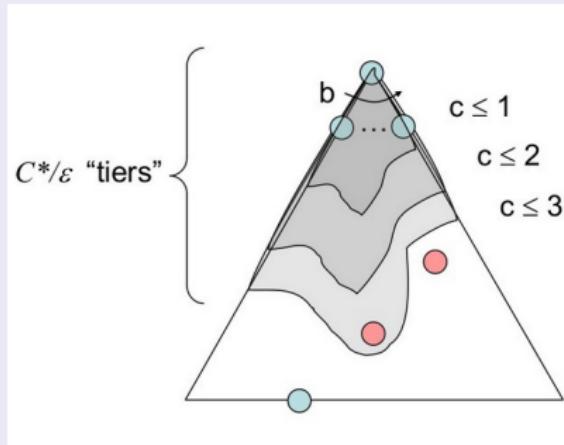
- max frontier size: $b^{1+LC^*/c}$
- $\Rightarrow O(b^{1+LC^*/c})$ memory size

Is it complete?

if solution exists, finite cost

\Rightarrow Yes

- Is it optimal?
 - Yes



(Oc D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

Memory requirement is a major problem also for uniform-cost search

Uniform-Cost Search (UCS): Properties

C^* : cost of cheapest solution; c : minimum arc cost

- $1 + \lceil C^*/c \rceil$ “effective depth” How many steps? processes all nodes costing less than cheapest solution
- \Rightarrow takes $O(b^{1+LC^*/c})$ time

How much memory?

- max frontier size: $b^{1+LC^*/c}$
- $\Rightarrow O(b^{1+LC^*/c})$ memory size

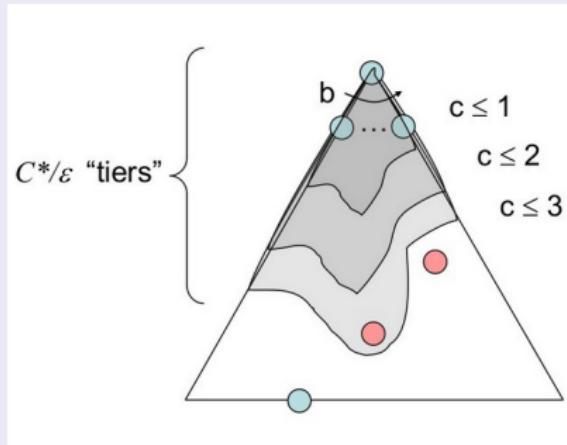
Is it complete?

if solution exists, finite cost

\Rightarrow Yes

- Is it optimal?

Yes



(Oc D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

Memory requirement is a major problem also for uniform-cost search

Uniform-Cost Search (UCS): Properties

C^* : cost of cheapest solution; c : minimum arc cost

- $1 + \lceil C^*/c \rceil$ “effective depth” How many steps? processes all nodes costing less than cheapest solution
- \Rightarrow takes $O(b^{1+LC^*/c})$ time

How much memory?

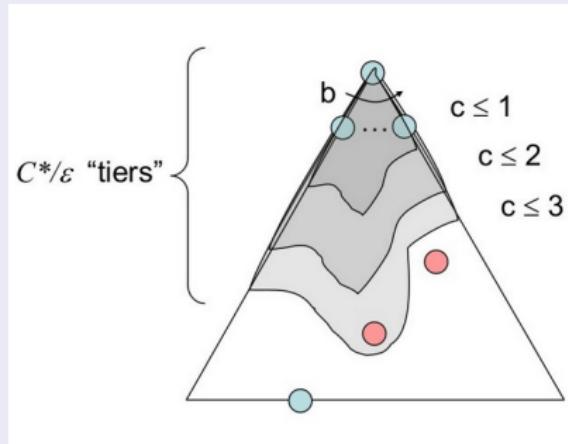
- max frontier size: $b^{1+LC^*/c}$
- $\Rightarrow O(b^{1+LC^*/c})$ memory size

Is it complete?

- if solution exists, finite cost
- \Rightarrow Yes

- Is it optimal?

- Yes



(Oc D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

Memory requirement is a major problem also for uniform-cost search

Uniform-Cost Search (UCS): Properties

C^* : cost of cheapest solution; c : minimum arc cost

- $1 + \lceil C^*/c \rceil$ “effective depth” How many steps? processes all nodes costing less than cheapest solution
- \Rightarrow takes $O(b^{1+LC^*/c})$ time

How much memory?

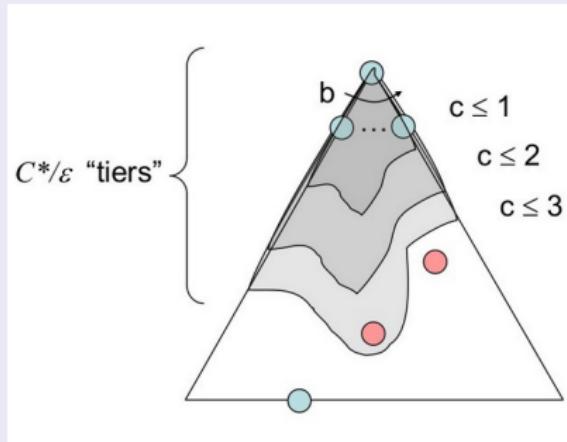
- max frontier size: $b^{1+LC^*/c}$
- $\Rightarrow O(b^{1+LC^*/c})$ memory size

Is it complete?

- if solution exists, finite cost
- \Rightarrow Yes

- Is it optimal?

- Yes



(Oc D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

Memory requirement is a major problem also for uniform-cost search

Uniform-Cost Search (UCS): Properties

C^* : cost of cheapest solution; c : minimum arc cost

- ⇒ 1 + $\lceil C^*/c \rceil$ “effective depth” How many steps?
processes all nodes costing less than cheapest solution
- ⇒ takes $O(b^{1+LC^*/c})$ time

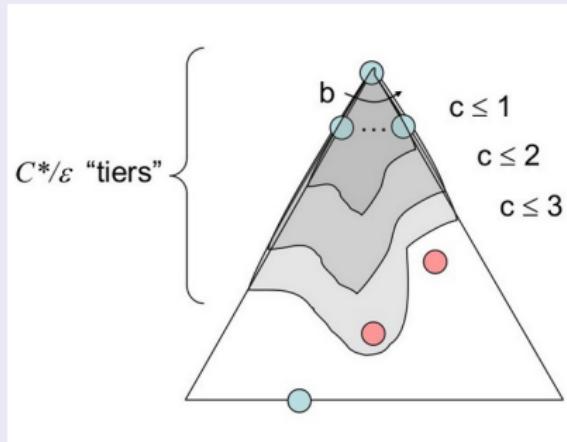
How much memory?

- max frontier size: $b^{1+LC^*/c}$
- ⇒ $O(b^{1+LC^*/c})$ memory size

Is it complete?

- if solution exists, finite cost
- ⇒ Yes

- Is it optimal?
- Yes



(Oc D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

Memory requirement is a major problem also for uniform-cost search

Uniform-Cost Search (UCS): Properties

C^* : cost of cheapest solution; c : minimum arc cost

- $1 + \lceil C^*/c \rceil$ “effective depth” How many steps? processes all nodes costing less than cheapest solution
- \Rightarrow takes $O(b^{1+LC^*/c})$ time

How much memory?

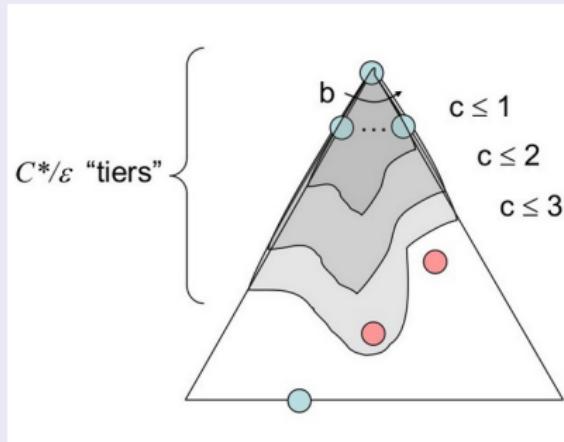
- max frontier size: $b^{1+LC^*/c}$
- $\Rightarrow O(b^{1+LC^*/c})$ memory size

Is it complete?

if solution exists, finite cost

\Rightarrow Yes

- Is it optimal?
 - Yes



(Oc D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

Memory requirement is a major problem also for uniform-cost search

Outline

- 1 [Problem-Solving Agents](#)
- 2 [Example Problems](#)
- 3 [Search Generalities](#)
- 4 [Uninformed Search Strategies](#)
 - Breadth-First [Search](#)
 - Uniform-cost [Search](#)
 - [Depth-First Search](#)
 - [Depth-Limited Search & Iterative Deepening](#)
- 5 [Informed Search Strategies](#)
 - Greedy [Search](#)
 - [A* Search Heuristic Functions](#)

Depth-First Search Strategy (DFS)

Depth-First Search

- Idea: Expand first the deepest unexpanded nodes
- Implementation: frontier/fringe implemented as a LIFO queue (aka stack)
 - novel successors pushed to the top of the stack

Depth-First Search Strategy (DFS)

Depth-First Search

- Idea: **Expand first the deepest unexpanded nodes**
- Implementation: frontier/fringe implemented as a LIFO queue (aka stack)
 - novel successors pushed to the top of the stack

Depth-First Search Strategy (DFS)

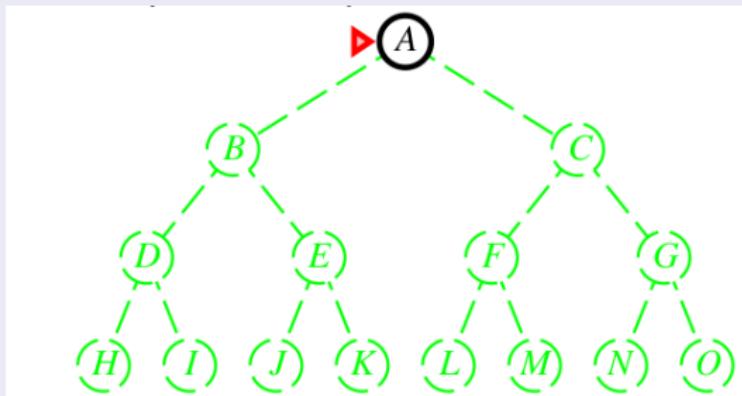
Depth-First Search

- Idea: Expand first the deepest unexpanded nodes
- Implementation: frontier/fringe implemented as a LIFO queue (aka stack)
==> novel successors pushed to the top of the stack

Depth-First Search Strategy (DFS)

Depth-First Search

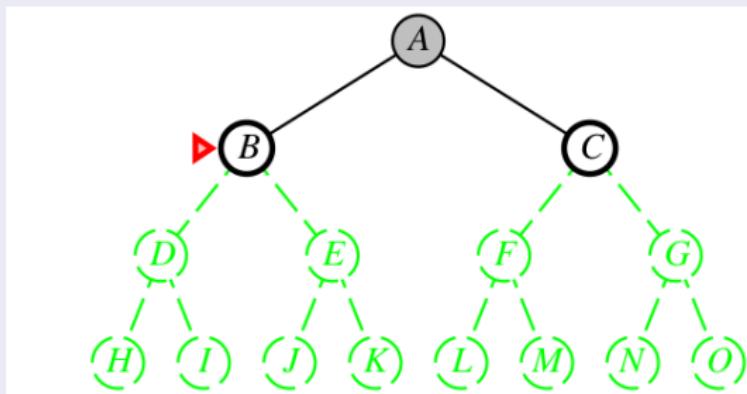
- Idea: Expand first the deepest unexpanded nodes
- Implementation: frontier/fringe implemented as a LIFO queue (aka stack)
==> novel successors pushed to the top of the stack



Depth-First Search Strategy (DFS)

Depth-First Search

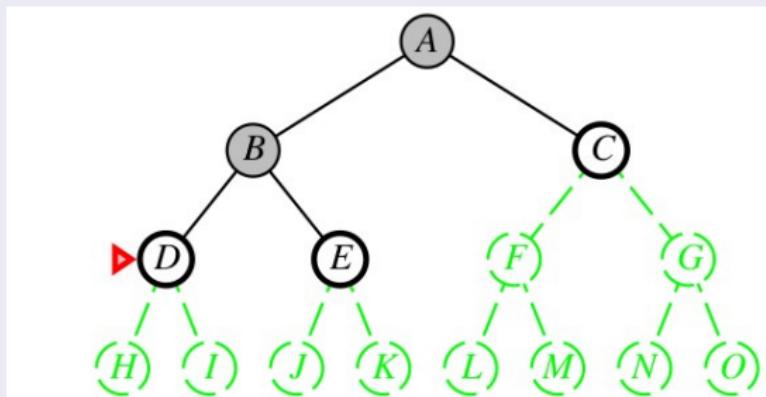
- Idea: Expand first the deepest unexpanded nodes
- Implementation: frontier/fringe implemented as a LIFO queue (aka stack)
==> novel successors pushed to the top of the stack



Depth-First Search Strategy (DFS)

Depth-First Search

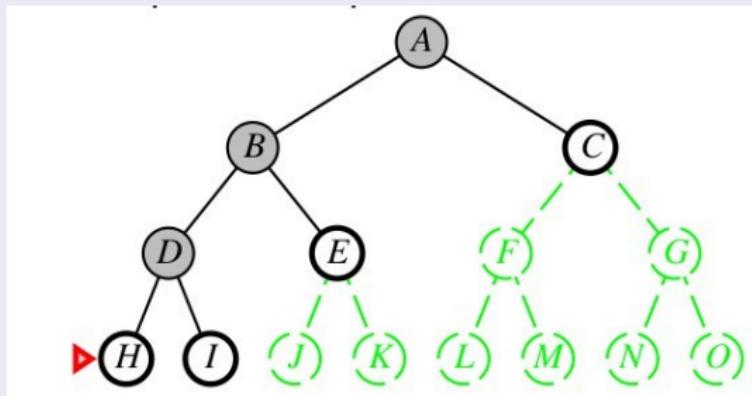
- Idea: Expand first the deepest unexpanded nodes
- Implementation: frontier/fringe implemented as a LIFO queue (aka stack)
==> novel successors pushed to the top of the stack



Depth-First Search Strategy (DFS)

Depth-First Search

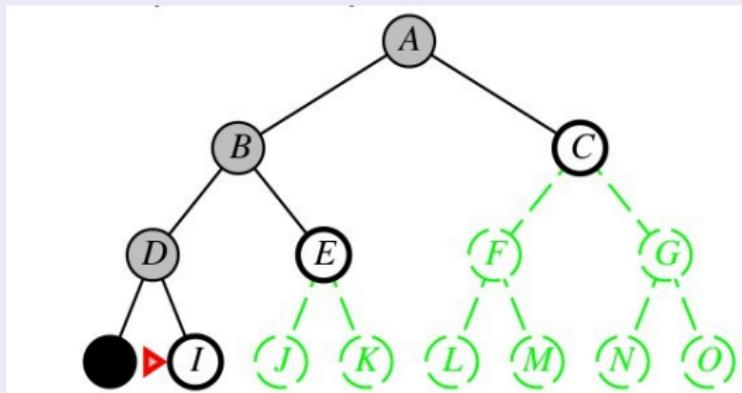
- Idea: Expand first the deepest unexpanded nodes
- Implementation: frontier/fringe implemented as a LIFO queue (aka stack)
==> novel successors pushed to the top of the stack



Depth-First Search Strategy (DFS)

Depth-First Search

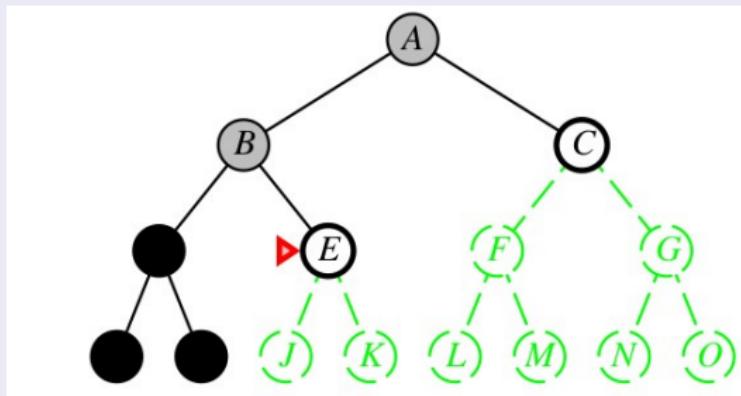
- Idea: Expand first the deepest unexpanded nodes
- Implementation: frontier/fringe implemented as a LIFO queue (aka stack)
==> novel successors pushed to the top of the stack



Depth-First Search Strategy (DFS)

Depth-First Search

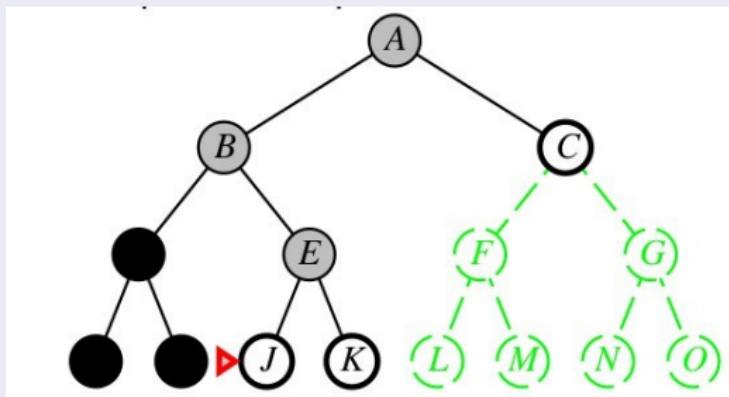
- Idea: Expand first the deepest unexpanded nodes
- Implementation: frontier/fringe implemented as a LIFO queue (aka stack)
==> novel successors pushed to the top of the stack



Depth-First Search Strategy (DFS)

Depth-First Search

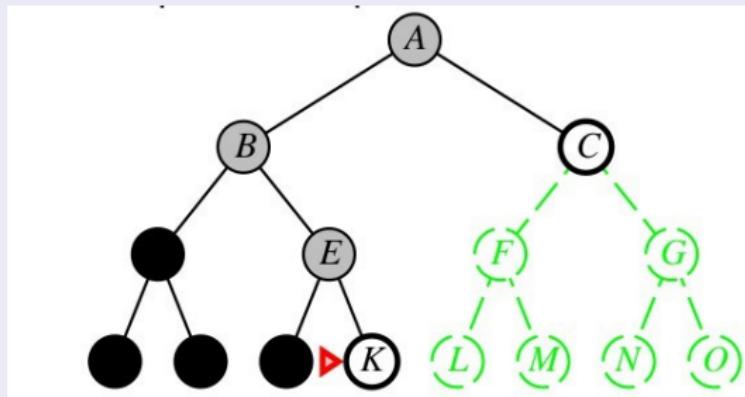
- Idea: Expand first the deepest unexpanded nodes
- Implementation: frontier/fringe implemented as a LIFO queue (aka stack)
==> novel successors pushed to the top of the stack



Depth-First Search Strategy (DFS)

Depth-First Search

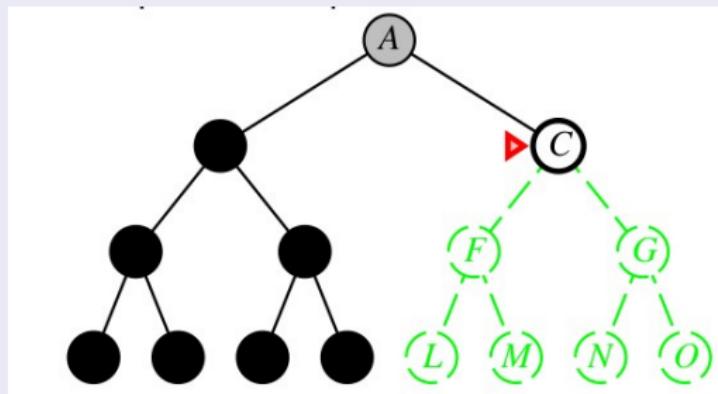
- Idea: Expand first the deepest unexpanded nodes
- Implementation: frontier/fringe implemented as a LIFO queue (aka stack)
==> novel successors pushed to the top of the stack



Depth-First Search Strategy (DFS)

Depth-First Search

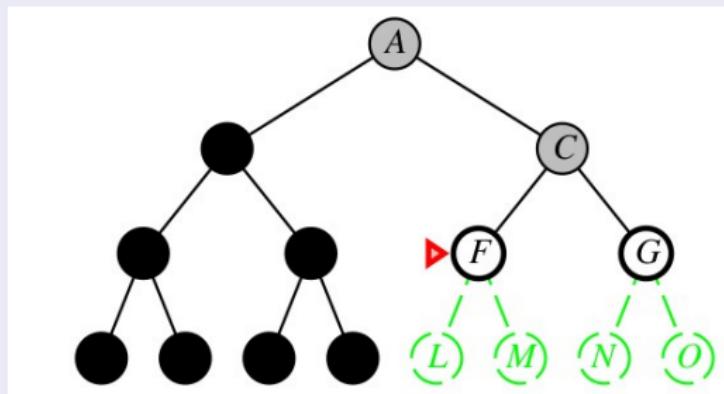
- Idea: Expand first the deepest unexpanded nodes
- Implementation: frontier/fringe implemented as a LIFO queue (aka stack)
==> novel successors pushed to the top of the stack



Depth-First Search Strategy (DFS)

Depth-First Search

- Idea: Expand first the deepest unexpanded nodes
- Implementation: frontier/fringe implemented as a LIFO queue (aka stack)
==> novel successors pushed to the top of the stack



Depth-First Search

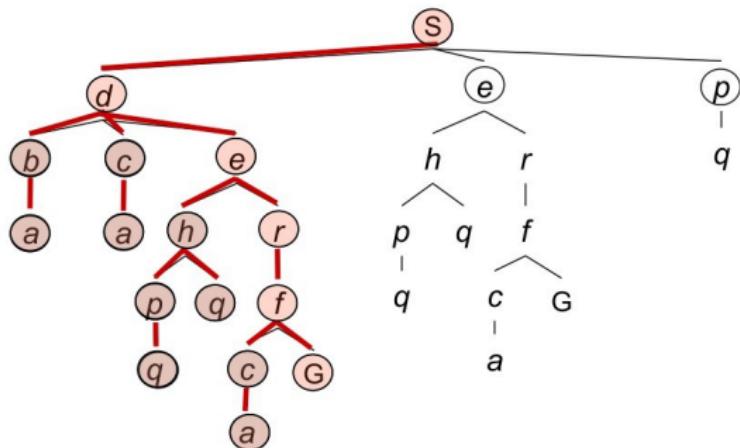
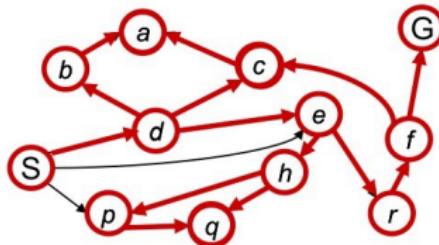
DFS on a Graph

Similar to BFS, using a LIFO access for frontier/fringe rather than FIFO.

Depth-First Search

Strategy: expand a deepest node first

*Implementation:
Fringe is a LIFO stack*



Depth-First Search (DFS): Properties

- How many steps?

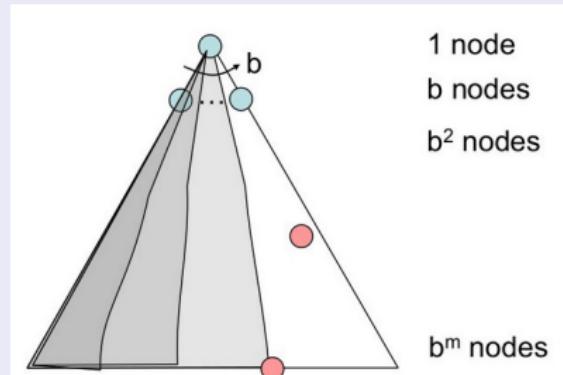
- could process the whole tree! \Rightarrow if m finite, takes $O(b^m)$ time

- How much memory?

- only siblings on path to root
 - $\Rightarrow O(bm)$ memory size

- Is it complete?

- if infinite state space: no
 - if finite state space:



(Oc D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

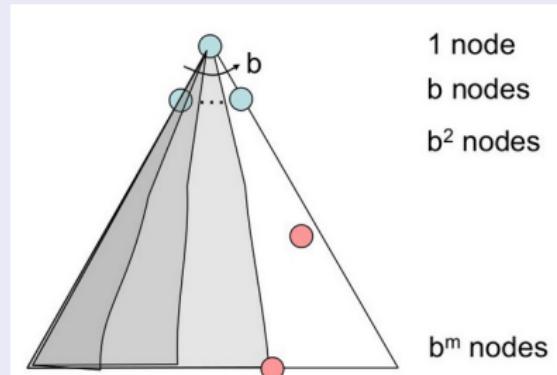
- Is it optimal?

- No, regardless of depth/cost
 - \Rightarrow "leftmost" solution

Memory requirement much better than BFS: $O(bm)$ vs. $O(b^d)$!

Depth-First Search (DFS): Properties

- How many steps?
 - could process the whole tree! \Rightarrow if m finite, takes $O(b^m)$ time
- How much memory?
 - only siblings on path to root
 - $\Rightarrow O(bm)$ memory size
- Is it complete?
 - if infinite state space: no
 - if finite state space:



(Oc D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

- Is it optimal?
 - No, regardless of depth/cost
 \Rightarrow "leftmost" solution

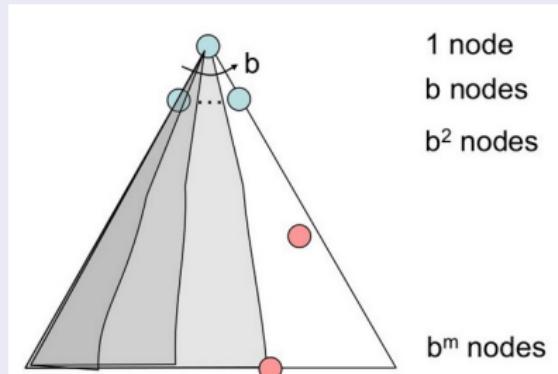
Memory requirement much better than BFS: $O(bm)$ vs. $O(b^d)$!

Depth-First Search (DFS): Properties

- How many steps?
 - could process the whole tree! \Rightarrow if m finite, takes $O(b^m)$ time
- How much memory?
 - only siblings on path to root
 - $\Rightarrow O(bm)$ memory size

Is it complete?

- if infinite state space: no
- if finite state space:



(Oc D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

- Is it optimal?
 - No, regardless of depth/cost
 - \Rightarrow "leftmost" solution

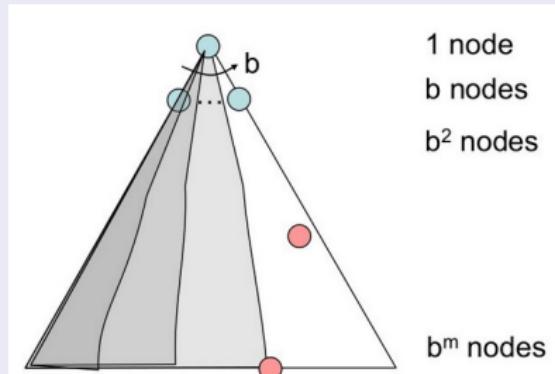
Memory requirement much better than BFS: $O(bm)$ vs. $O(b^d)$!

Depth-First Search (DFS): Properties

- How many steps?
 - could process the whole tree! \Rightarrow if m finite, takes $O(b^m)$ time
- How much memory?
 - only siblings on path to root
 - $\Rightarrow O(bm)$ memory size

Is it complete?

- if infinite state space: no
- if finite state space:



(Oc D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

- Is it optimal?
 - No, regardless of depth/cost
 - \Rightarrow "leftmost" solution

Memory requirement much better than BFS: $O(bm)$ vs. $O(b^d)$!

Depth-First Search (DFS): Properties

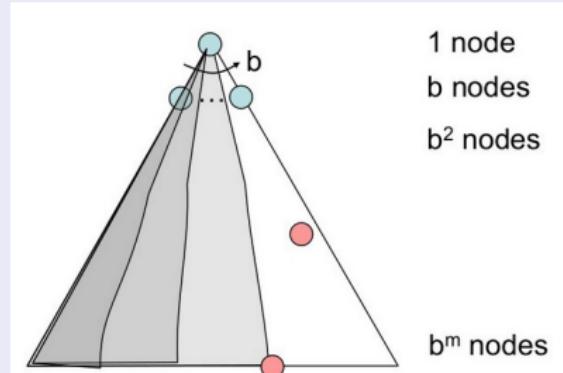
- How many steps?
 - could process the whole tree! \Rightarrow if m finite, takes $O(b^m)$ time
- How much memory?
 - only siblings on path to root
 - $\Rightarrow O(bm)$ memory size

Is it complete?

- if infinite state space: no
- if finite state space:
 - graph version: yes
 - tree version: only if we prevent loops

Is it optimal?

No, regardless of



(Oc D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

Memory requirement much better than BFS: $O(bm)$ vs. $O(b^d)$!
 \Rightarrow typically preferred to BFS

Depth-First Search (DFS): Properties

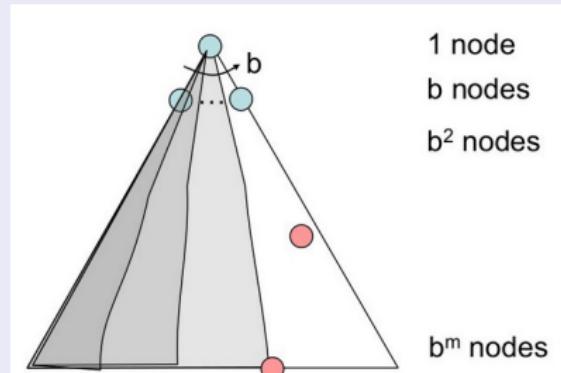
- How many steps?
 - could process the whole tree! \Rightarrow if m finite, takes $O(b^m)$ time
- How much memory?
 - only siblings on path to root
 - $\Rightarrow O(bm)$ memory size

Is it complete?

- if infinite state space: no
- if finite state space:
 - graph version: yes
 - tree version: only if we prevent loops

Is it optimal?

No, regardless of



(Oc D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

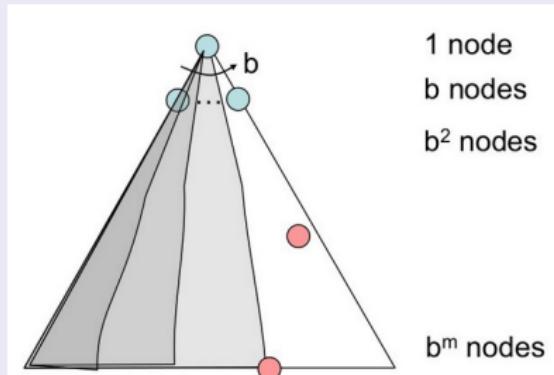
Memory requirement much better than BFS: $O(bm)$ vs. $O(b^d)$!
 \Rightarrow typically preferred to BFS

Depth-First Search (DFS): Properties

- How many steps?
 - could process the whole tree! \Rightarrow if m finite, takes $O(b^m)$ time
- How much memory?
 - only siblings on path to root
 - $\Rightarrow O(bm)$ memory size
- Is it complete?
 - if infinite state space: no
 - if finite state space:
 - graph version: yes
 - tree version: only if we prevent loops

Is it optimal?

No, regardless of



(Oc D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

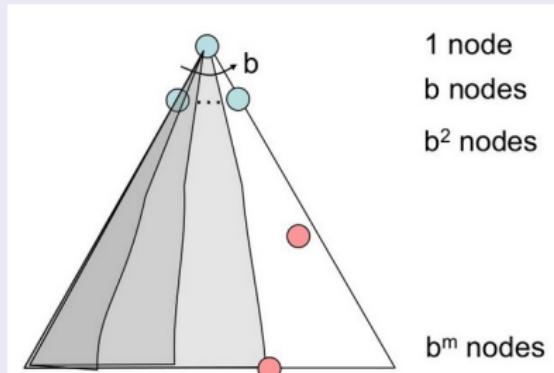
Memory requirement much better than BFS: $O(bm)$ vs. $O(b^d)$!
 \Rightarrow typically preferred to BFS

Depth-First Search (DFS): Properties

- How many steps?
 - could process the whole tree! \Rightarrow if m finite, takes $O(b^m)$ time
- How much memory?
 - only siblings on path to root
 - $\Rightarrow O(bm)$ memory size
- Is it complete?
 - if infinite state space: no
 - if finite state space:
 - graph version: yes
 - tree version: only if we prevent loops

Is it optimal?

No, regardless of



(Oc D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

Memory requirement much better than BFS: $O(bm)$ vs. $O(b^d)$!
 \Rightarrow typically preferred to BFS

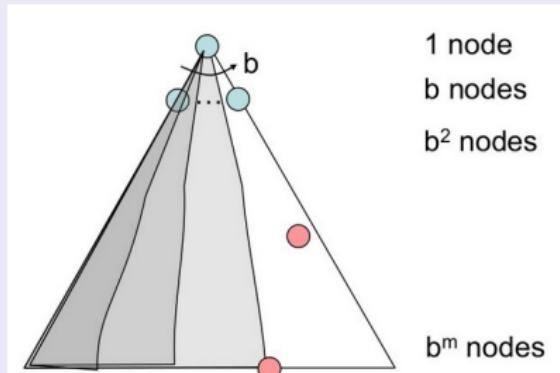
Depth-First Search (DFS): Properties

- How many steps?
 - could process the whole tree! \Rightarrow if m finite, takes $O(b^m)$ time
- How much memory?
 - only siblings on path to root
 - $\Rightarrow O(bm)$ memory size
- Is it complete?
 - if infinite state space: no
 - if finite state space:
 - graph version: yes
 - tree version: only if we prevent loops

Is it optimal?

No, regardless of

Memory requirement much better than BFS: $O(bm)$ vs. $O(b^d)$!
 \Rightarrow typically preferred to BFS



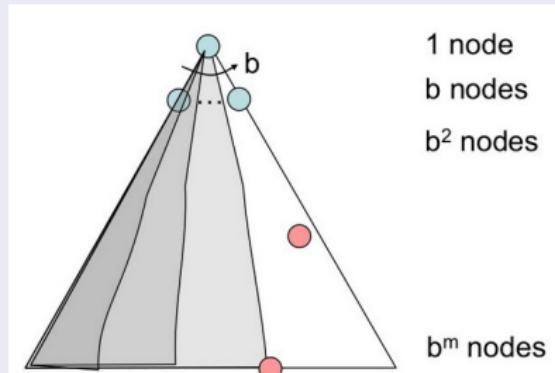
(Oc D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

Depth-First Search (DFS): Properties

- How many steps?
 - could process the whole tree! \Rightarrow if m finite, takes $O(b^m)$ time
- How much memory?
 - only siblings on path to root
 - $\Rightarrow O(bm)$ memory size
- Is it complete?
 - if infinite state space: no
 - if finite state space:
 - graph version: yes
 - tree version: only if we prevent loops

Is it optimal?

No, regardless of



(Oc D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

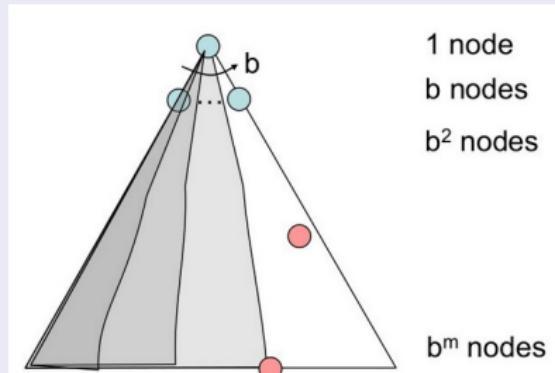
Memory requirement much better than BFS: $O(bm)$ vs. $O(b^d)$!
 \Rightarrow typically preferred to BFS

Depth-First Search (DFS): Properties

- How many steps?
 - could process the whole tree! \Rightarrow if m finite, takes $O(b^m)$ time
- How much memory?
 - only siblings on path to root
 - $\Rightarrow O(bm)$ memory size
- Is it complete?
 - if infinite state space: no
 - if finite state space:
 - graph version: yes
 - tree version: only if we prevent loops

- Is it optimal?

No, regardless of



(Oc D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

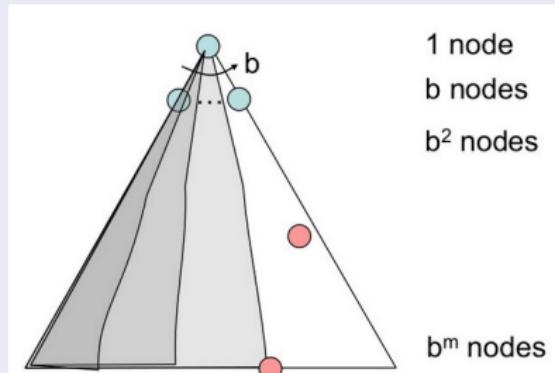
Memory requirement much better than BFS: $O(bm)$ vs. $O(b^d)$!
 \Rightarrow typically preferred to BFS

Depth-First Search (DFS): Properties

- How many steps?
 - could process the whole tree! \Rightarrow if m finite, takes $O(b^m)$ time
- How much memory?
 - only siblings on path to root
 - $\Rightarrow O(bm)$ memory size
- Is it complete?
 - if infinite state space: no
 - if finite state space:
 - graph version: yes
 - tree version: only if we prevent loops

• Is it optimal?

No, regardless of



(Oc D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

Memory requirement much better than BFS: $O(bm)$ vs. $O(b^d)$!
 \Rightarrow typically preferred to BFS

A Variant of DFS: Backtracking Search

Backtracking Search

- Idea: **only one successor is generated at the time**
 - each partially-expanded node remembers which successor to generate next
 - generate a successor by modifying the current state description, rather than copying it first
 - Applied in CSP, SAT/SMT and Logic Programming

\implies only $O(m)$ memory is needed rather than $O(bm)$

A Variant of DFS: Backtracking Search

Backtracking Search

- Idea: **only one successor is generated at the time**
 - each partially-expanded node remembers which successor to generate next
 - generate a successor by modifying the current state description, rather than copying it first
 - Applied in CSP, SAT/SMT and Logic Programming

\Rightarrow only $O(m)$ memory is needed rather than $O(bm)$

Outline

- 1 [Problem-Solving Agents](#)
- 2 [Example Problems](#)
- 3 [Search Generalities](#)
- 4 [Uninformed Search Strategies](#)
 - Breadth-First [Search](#)
 - Uniform-cost [Search](#)
 - [Depth-First Search](#)
 - [Depth-Limited Search & Iterative Deepening](#)
- 5 [Informed Search Strategies](#)
 - Greedy [Search](#)
 - [A* Search](#) Heuristic Functions

Depth-Limited Search (DLS) Strategy

Depth-Limited Search (DLS)

- Idea: depth-first search with depth limit /
 - i.e., nodes at depth l treated as having no successors
 - DFS is DLS with $l = +\infty$
- solves the infinite-path problem of DFS
=> allows DFS deal with infinite-state spaces
- useful also if maximum-depth is known by domain knowledge
 - e.g., if maximum node distance in a graph (diameter) is known
 - Ex: Romania trip: 9 steps

Drawbacks (d : depth of the shallowest goal):

if $d > l \Rightarrow$ incomplete

if $d < l \Rightarrow$ takes $O(b^d)$ instead of $O(b^l)$ steps

Depth-Limited Search (DLS) Strategy

Depth-Limited Search (DLS)

- Idea: depth-first search with depth limit l
 - i.e., nodes at depth l treated as having no successors
 - DFS is DLS with $l = +\infty$
- solves the infinite-path problem of DFS
=> allows DFS deal with infinite-state spaces
- useful also if maximum-depth is known by domain knowledge
 - e.g., if maximum node distance in a graph (diameter) is known
 - Ex: Romania trip: 9 steps

Drawbacks (d : depth of the shallowest goal):

if $d > l \Rightarrow$ incomplete

if $d < l \Rightarrow$ takes $O(b^d)$ instead of $O(b^l)$ steps

Depth-Limited Search (DLS) Strategy

Depth-Limited Search (DLS)

- Idea: depth-first search with depth limit l
 - i.e., nodes at depth l treated as having no successors
 - DFS is DLS with $l = +\infty$
- solves the infinite-path problem of DFS
=> allows DFS deal with infinite-state spaces
- useful also if maximum-depth is known by domain knowledge
 - e.g., if maximum node distance in a graph (diameter) is known
 - Ex: Romania trip: 9 steps

Drawbacks (d : depth of the shallowest goal):

if $d > l \Rightarrow$ incomplete

if $d < l \Rightarrow$ takes $O(b^d)$ instead of $O(b^l)$ steps

Depth-Limited Search (DLS) Strategy

Depth-Limited Search (DLS)

- Idea: depth-first search with depth limit /
 - i.e., nodes at depth l treated as having no successors
 - DFS is DLS with $l = +\infty$
- solves the infinite-path problem of DFS
=> allows DFS deal with infinite-state spaces
- useful also if maximum-depth is known by domain knowledge
 - e.g., if maximum node distance in a graph (**diameter**) is known
 - Ex: Romania trip: 9 steps

Drawbacks (d : depth of the shallowest goal):

if $d > l \Rightarrow$ incomplete

if $d < l \Rightarrow$ takes $O(b^d)$ instead of $O(b^l)$ steps

Depth-Limited Search (DLS) Strategy

Depth-Limited Search (DLS)

- Idea: depth-first search with depth limit l
 - i.e., nodes at depth l treated as having no successors
 - DFS is DLS with $l = +\infty$
- solves the infinite-path problem of DFS
=> allows DFS deal with infinite-state spaces
- useful also if maximum-depth is known by domain knowledge
 - e.g., if maximum node distance in a graph (diameter) is known
 - Ex: Romania trip: 9 steps

Drawbacks (d : depth of the shallowest goal):

if $d > l \Rightarrow$ incomplete

if $d < l \Rightarrow$ takes $O(b^l)$ instead of $O(b^d)$ steps

Depth-Limited Search (DLS) Strategy

Depth-Limited Search (DLS)

- Idea: depth-first search with depth limit /
 - i.e., nodes at depth l treated as having no successors
 - DFS is DLS with $l = +\infty$
- solves the infinite-path problem of DFS
=> allows DFS deal with infinite-state spaces
- useful also if maximum-depth is known by domain knowledge
 - e.g., if maximum node distance in a graph (diameter) is known
 - Ex: Romania trip: 9 steps

Drawbacks (d : depth of the shallowest goal):

if $d > l \Rightarrow$ incomplete

if $d < l \Rightarrow$ takes $O(b^l)$ instead of $O(b^d)$ steps

Depth-Limited Search (DLS) Strategy

Depth-Limited Search (DLS)

- Idea: depth-first search with depth limit /
 - i.e., nodes at depth l treated as having no successors
 - DFS is DLS with $l = +\infty$
- solves the infinite-path problem of DFS
=> allows DFS deal with infinite-state spaces
- useful also if maximum-depth is known by domain knowledge
 - e.g., if maximum node distance in a graph (diameter) is known
 - Ex: Romania trip: 9 steps

Drawbacks (d : depth of the shallowest goal):

if $d > l \Rightarrow$ incomplete

if $d < l \Rightarrow$ takes $O(b^l)$ instead of $O(b^d)$ steps

Depth-Limited Search (DLS) Strategy [cont.]

Recursive DLS

```
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?  $\leftarrow$  false
    for each action in problem.ACTIONS(node.STATE) do
      child  $\leftarrow$  CHILD-NODE(problem, node, action)
      result  $\leftarrow$  RECURSIVE-DLS(child, problem, limit - 1)
      if result = cutoff then cutoff_occurred?  $\leftarrow$  true
      else if result  $\neq$  failure then return result
    if cutoff_occurred? then return cutoff else return failure
```

(Oc S. Russell & P. Norwig, AIMA)

Iterative-Deepening Search Strategy (IDS)

Iterative-Deepening Search

- Idea: call iteratively DLS for increasing depths $l = 0, 1, 2, 3\dots$
- combines the advantages of breadth- and depth-first strategies
 - complete (like BFS)
 - takes $O(b^d)$ steps (like BFS and DFS)
 - requires $O(bd)$ memory (like DFS)
 - explores a single branch at a time (like DFS)
 - optimal only if step cost = 1
- optimal variants exist: iterative-lengthening search (see AIMA)

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

Iterative-Deepening Search Strategy (IDS)

Iterative-Deepening Search

- Idea: call iteratively DLS for increasing depths $l = 0, 1, 2, 3\dots$
- combines the advantages of breadth- and depth-first strategies
 - complete (like BFS)
 - takes $O(b^d)$ steps (like BFS and DFS)
 - requires $O(bd)$ memory (like DFS)
 - explores a single branch at a time (like DFS)
 - optimal only if step cost = 1
- optimal variants exist: iterative-lengthening search (see AIMA)

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

Iterative-Deepening Search Strategy (IDS)

Iterative-Deepening Search

- Idea: call iteratively DLS for increasing depths $l = 0, 1, 2, 3\dots$
- combines the advantages of breadth- and depth-first strategies
 - complete (like BFS)
 - takes $O(b^d)$ steps (like BFS and DFS)
 - requires $O(bd)$ memory (like DFS)
 - explores a single branch at a time (like DFS)
- optimal only if step cost = 1
 - optimal variants exist: iterative-lengthening search (see AIMA)

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

Iterative-Deepening Search Strategy (IDS)

Iterative-Deepening Search

- Idea: call iteratively DLS for increasing depths $l = 0, 1, 2, 3\dots$
- combines the advantages of breadth- and depth-first strategies
 - complete (like BFS)
 - takes $O(b^d)$ steps (like BFS and DFS)
 - requires $O(bd)$ memory (like DFS)
 - explores a single branch at a time (like DFS)
- optimal only if step cost = 1
 - optimal variants exist: iterative-lengthening search (see AIMA)

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

Iterative-Deepening Search Strategy (IDS)

Iterative-Deepening Search

- Idea: call iteratively DLS for increasing depths $l = 0, 1, 2, 3\dots$
- combines the advantages of breadth- and depth-first strategies
 - complete (like BFS)
 - takes $O(b^d)$ steps (like BFS and DFS)
 - requires $O(bd)$ memory (like DFS)
 - explores a single branch at a time (like DFS)
 - optimal only if step cost = 1
- optimal variants exist: iterative-lengthening search (see AIMA)

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

Iterative-Deepening Search Strategy (IDS)

Iterative-Deepening Search

- Idea: call iteratively DLS for increasing depths $l = 0, 1, 2, 3\dots$
- combines the advantages of breadth- and depth-first strategies
 - complete (like BFS)
 - takes $O(b^d)$ steps (like BFS and DFS)
 - requires $O(bd)$ memory (like DFS)
 - explores a single branch at a time (like DFS)
- optimal only if step cost = 1
 - optimal variants exist: iterative-lengthening search (see AIMA)

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

Iterative-Deepening Search Strategy (IDS)

Iterative-Deepening Search

- Idea: call iteratively DLS for increasing depths $l = 0, 1, 2, 3\dots$
- combines the advantages of breadth- and depth-first strategies
 - complete (like BFS)
 - takes $O(b^d)$ steps (like BFS and DFS)
 - requires $O(bd)$ memory (like DFS)
 - explores a single branch at a time (like DFS)
- optimal only if step cost = 1
 - optimal variants exist: **iterative-lengthening search** (see AIMA)

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

Iterative-Deepening Search Strategy (IDS)

Iterative-Deepening Search

- Idea: call iteratively DLS for increasing depths $l = 0, 1, 2, 3\dots$
- combines the advantages of breadth- and depth-first strategies
 - complete (like BFS)
 - takes $O(b^d)$ steps (like BFS and DFS)
 - requires $O(bd)$ memory (like DFS)
 - explores a single branch at a time (like DFS)
- optimal only if step cost = 1
 - optimal variants exist: **iterative-lengthening search** (see AIMA)

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

Iterative-Deepening Search: Example

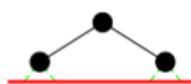
Limit = 0



(Oc S. Russell & P. Norwig, AIMA)

Iterative-Deepening Search: Example

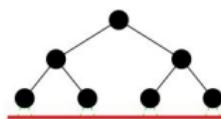
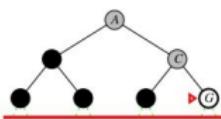
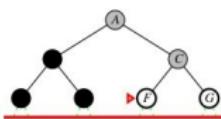
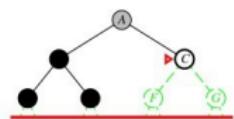
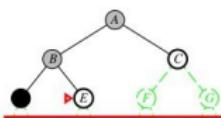
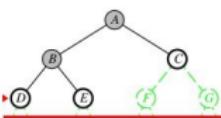
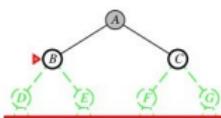
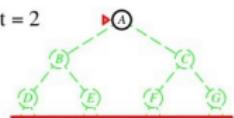
Limit = 1



(Oc S. Russell & P. Norwig, AIMA)

Iterative-Deepening Search: Example

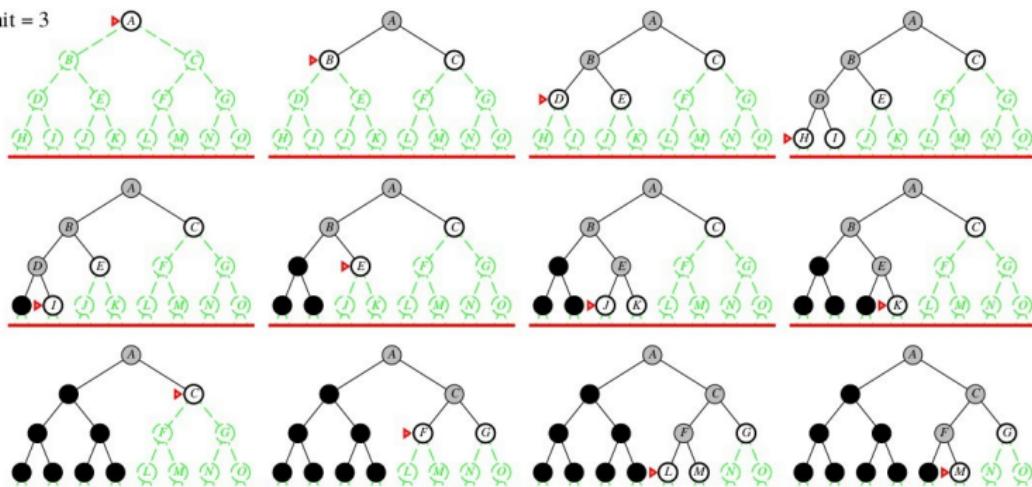
Limit = 2



(Oc S. Russell & P. Norwig, AIMA)

Iterative-Deepening Search: Example

Limit = 3



(Oc S. Russell & P. Norwig, AIMa)

Iterative-Deepening Search Strategy (IDS) [cont.]

Remark: Why “only” $O(b^d)$ steps?

- may seem wasteful since states are generated multiple times...
- ... however, only a small fraction of nodes are multiply generated
- number of repeatedly-generated nodes decreases exponentially with number of repetitions
 - depth 1 (b nodes): repeated d times
 - depth 2 (b^2 nodes): repeated $d - 1$ times
 - ...
 - depth d (b^d nodes): repeated 1 time

⇒ The total number of generated nodes is:

$$\begin{aligned}N(\text{IDS}) &= (d)b^1 + (d-1)b^2 \\&\quad + \dots + (1)b^d = O(b^d)\end{aligned}$$

$$\begin{aligned}\bullet \quad N(\text{BFS}) &= b^1 + b^2 + \dots + b^d \\&= O(b^d)\end{aligned}$$

Ex. with $b = 10$ and $d = 3$:

$$\begin{aligned}N(\text{IDS}) &= 50 + 400 + 3,000 + 20,\\&\quad 000 = 100,000 = 123,000\end{aligned}$$

Iterative-Deepening Search Strategy (IDS) [cont.]

Remark: Why “only” $O(b^d)$ steps?

- may seem wasteful since states are generated multiple times...
- ... however, only a small fraction of nodes are multiply generated
- number of repeatedly-generated nodes decreases exponentially with number of repetitions
 - depth 1 (b nodes): repeated d times
 - depth 2 (b^2 nodes): repeated $d - 1$ times
 - ...
 - depth d (b^d nodes): repeated 1 time

⇒ The total number of generated nodes is:

$$\begin{aligned}N(\text{IDS}) &= (d)b^1 + (d-1)b^2 \\&\quad + \dots + (1)b^d = O(b^d)\end{aligned}$$

$$\begin{aligned}\bullet \quad N(\text{BFS}) &= b^1 + b^2 + \dots + b^d \\&= O(b^d)\end{aligned}$$

Ex. with $b = 10$ and $d = 3$:

$$\begin{aligned}N(\text{IDS}) &= 50 + 400 + 3,000 + 20,\\&\quad 000 = 100,000 = 123,000\end{aligned}$$

Iterative-Deepening Search Strategy (IDS) [cont.]

Remark: Why “only” $O(b^d)$ steps?

- may seem wasteful since states are generated multiple times...
- ... however, only a small fraction of nodes are multiply generated
- number of repeatedly-generated nodes decreases exponentially with number of repetitions
 - depth 1 (b nodes): repeated d times
 - depth 2 (b^2 nodes): repeated $d - 1$ times
 - ...
 - depth d (b^d nodes): repeated 1 time

⇒ The total number of generated nodes is:

$$\begin{aligned}N(\text{IDS}) &= (d)b^1 + (d-1)b^2 \\&\quad + \dots + (1)b^d = O(b^d)\end{aligned}$$

$$\begin{aligned}\bullet \quad N(\text{BFS}) &= b^1 + b^2 + \dots + b^d \\&= O(b^d)\end{aligned}$$

Ex. with $b = 10$ and $d = 3$:

$$\begin{aligned}N(\text{IDS}) &= 50 + 400 + 3,000 + 20,\\&\quad 000 = 100,000 = 123,000\end{aligned}$$

Iterative-Deepening Search Strategy (IDS) [cont.]

Remark: Why “only” $O(b^d)$ steps?

- may seem wasteful since states are generated multiple times...
- ... however, only a small fraction of nodes are multiply generated
- number of repeatedly-generated nodes decreases exponentially with number of repetitions
 - depth 1 (b nodes): repeated d times
 - depth 2 (b^2 nodes): repeated $d - 1$ times
 - ...
 - depth d (b^d nodes): repeated 1 time

==> The total number of generated nodes is:

$$\begin{aligned}N(\text{IDS}) &= (d)b^1 + (d-1)b^2 \\&+ \dots + (1)b^d = O(b^d)\end{aligned}$$

- $\begin{aligned}N(\text{BFS}) &= b^1 + b^2 + \dots + b^d \\&= O(b^d)\end{aligned}$

Ex: with $b = 10$ and $d = 5$:

$$\begin{aligned}N(\text{IDS}) &= 50 + 400 + 3,000 + 20,000 + 100,000 = 123,000\end{aligned}$$

Iterative-Deepening Search Strategy (IDS) [cont.]

Remark: Why “only” $O(b^d)$ steps?

- may seem wasteful since states are generated multiple times...
- ... however, only a small fraction of nodes are multiply generated
- number of repeatedly-generated nodes decreases exponentially with number of repetitions
 - depth 1 (b nodes): repeated d times
 - depth 2 (b^2 nodes): repeated $d - 1$ times
 - ...
 - depth d (b^d nodes): repeated 1 time

==> The total number of generated nodes is:

$$\begin{aligned}N(\text{IDS}) &= (d)b^1 + (d-1)b^2 \\&\quad + \dots + (1)b^d = O(b^d)\end{aligned}$$

$$\begin{aligned}N(\text{BFS}) &= b^1 + b^2 + \dots + b^d \\&= O(b^d)\end{aligned}$$

Ex: with $b = 10$ and $d = 5$:

$$\begin{aligned}N(\text{IDS}) &= 50 + 400 + 3,000 + 20,000 + 100,000 = 123,000\end{aligned}$$

Iterative-Deepening Search Strategy (IDS) [cont.]

Remark: Why “only” $O(b^d)$ steps?

- may seem wasteful since states are generated multiple times...
- ... however, only a small fraction of nodes are multiply generated
- number of repeatedly-generated nodes decreases exponentially with number of repetitions
 - depth 1 (b nodes): repeated d times
 - depth 2 (b^2 nodes): repeated $d - 1$ times
 - ...
 - depth d (b^d nodes): repeated 1 time

==> The total number of generated nodes is:

$$\begin{aligned}N(\text{IDS}) &= (d)b^1 + (d - 1)b^2 \\&+ \dots + (1)b^d = O(b^d)\end{aligned}$$

- $\begin{aligned}N(\text{BFS}) &= b^1 + b^2 + \dots + b^d \\&= O(b^d)\end{aligned}$

Ex: with $b = 10$ and $d = 5$:

$$\begin{aligned}N(\text{IDS}) &= 50 + 400 + 3,000 + 20,000 + 100,000 = 123,000\end{aligned}$$

Iterative-Deepening Search Strategy (IDS) [cont.]

Remark: Why “only” $O(b^d)$ steps?

- may seem wasteful since states are generated multiple times...
- ... however, only a small fraction of nodes are multiply generated
- number of repeatedly-generated nodes decreases exponentially with number of repetitions
 - depth 1 (b nodes): repeated d times
 - depth 2 (b^2 nodes): repeated $d - 1$ times
 - ...
 - depth d (b^d nodes): repeated 1 time

==> The total number of generated nodes is:

$$\begin{aligned}N(\text{IDS}) &= (d)b^1 + (d-1)b^2 \\&\quad + \dots + (1)b^d = O(b^d)\end{aligned}$$

- $\begin{aligned}N(\text{BFS}) &= b^1 + b^2 + \dots + b^d \\&= O(b^d)\end{aligned}$

Ex: with $b = 10$ and $d = 5$:

$$\begin{aligned}N(\text{IDS}) &= 50 + 400 + 3,000 + 20,000 + 100,000 = 123,000\end{aligned}$$

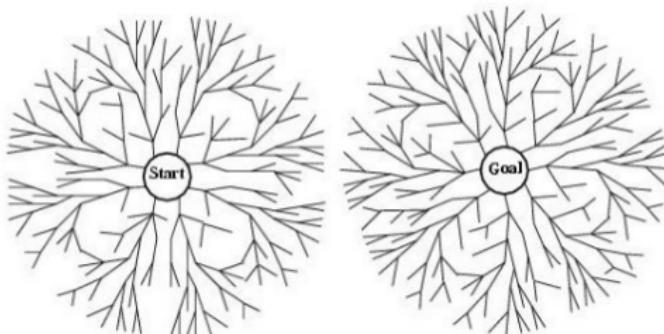
Bidirectional Search [hints]

- Idea: Two simultaneous searches:
 - forward: from start node
 - backward: from goal nodechecking if the node belongs to the other frontier before expansion

Rationale: $b^{d/2} + b^{d/2} << b^d$

==> number of steps and memory consumption are $\approx 2b^{d/2}$

backward
some cases



Uninformed Search Strategies: Comparison

Evaluation of tree-search strategies

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	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(bl)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes ^c	Yes	No	No	Yes ^c	Yes ^{c,d}

^a: complete if b is finite

^b: complete if step costs $\geq E$ for some positive E

^c: optimal if step costs are all identical

^d: if both directions use breadth-first search

(Oc S. Russell & P. Norwig, AIMA)

For graph searches, the main differences are:

- depth-first search is complete for finite-state spaces
- space & time complexities are bounded by the state space size

Outline

1 Problem-Solving Agents

2 Example Problems

3 Search Generalities

4 Uninformed Search Strategies

- Breadth-First Search
- Uniform-cost Search

Depth-First Search

Depth-Limited Search &

Iterative Deepening

5 Informed Search Strategies

- Greedy Search

A* Search Heuristic

Functions

Informed Search Strategies

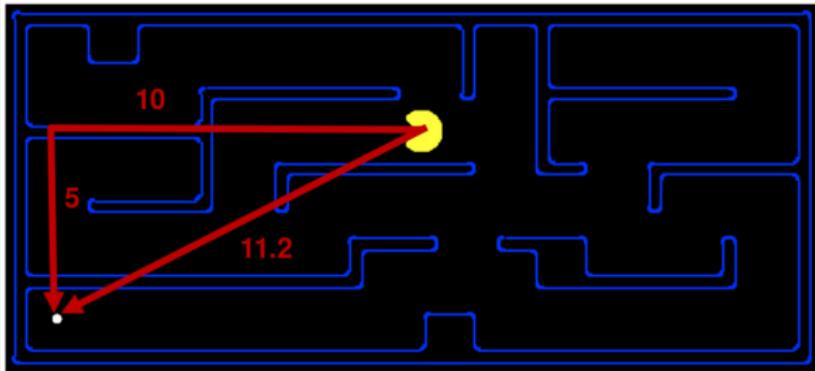
Some general principles

- The intelligence of a system cannot be measured only in terms of search capacity, but in **the ability to use knowledge about the problem to reduce/mitigate the combinatorial explosion**
- If the system has some control on the order in which candidate solutions are generated, then it is useful to use this order so that actual solutions have a high chance to appear earlier
- **Intelligence, for a system with limited processing capacity, is the wise choice of what to do next**

Heuristic search and heuristic functions

Heuristic search and heuristic functions

- Idea: don't ignore the goal when selecting nodes
- Intuition: often there is extra knowledge that can be used to guide the search towards the goal: heuristics
- A heuristic is
 - a function $h(n)$ that estimates how close a state n is to a goal
 - designed for a particular search problem
 - Ex Manhattan distance, Euclidean distance for pathing



Best-first Search Strategies

General approach of informed search: Best-first search

- Best-first search: node selected for expansion based on an evaluation function $f(n)$
 - represent a cost estimate \Rightarrow choose node which appears best
 - implemented like uniform-cost search, with f instead of g
 - \Rightarrow the frontier is a priority queue sorted in decreasing order of $h(n)$
 - both tree-based and graph-based versions
 - most often f includes a heuristic function $h(n)$
- Heuristic function $h(n) \in \mathbb{R}^+$: estimated cost of the cheapest path from the state at node n to a goal state
 - $h(n) \geq 0 \forall n$
 - If G is goal, then $h(G) = 0$
 - implements extra domain knowledge
 - depends only on state, not on node
 - (e.g., independent on paths)
- Main strategies:
 - Greedy search
 - A^* search

Best-first Search Strategies

General approach of informed search: Best-first search

- Best-first search: node selected for expansion based on an evaluation function $f(n)$
 - represent a cost estimate \Rightarrow choose node which appears best
 - implemented like uniform-cost search, with f instead of g
 - \Rightarrow the frontier is a priority queue sorted in decreasing order of $h(n)$
 - both tree-based and graph-based versions
 - most often f includes a heuristic function $h(n)$
- Heuristic function $h(n) \in \mathbb{R}^+$: estimated cost of the cheapest path from the state at node n to a goal state
 - $h(n) \geq 0 \forall n$
 - If G is goal, then $h(G) = 0$
 - implements extra domain knowledge
 - depends only on state, not on node
 - (e.g., independent on paths)

Main strategies:

- Greedy search
- A^* search

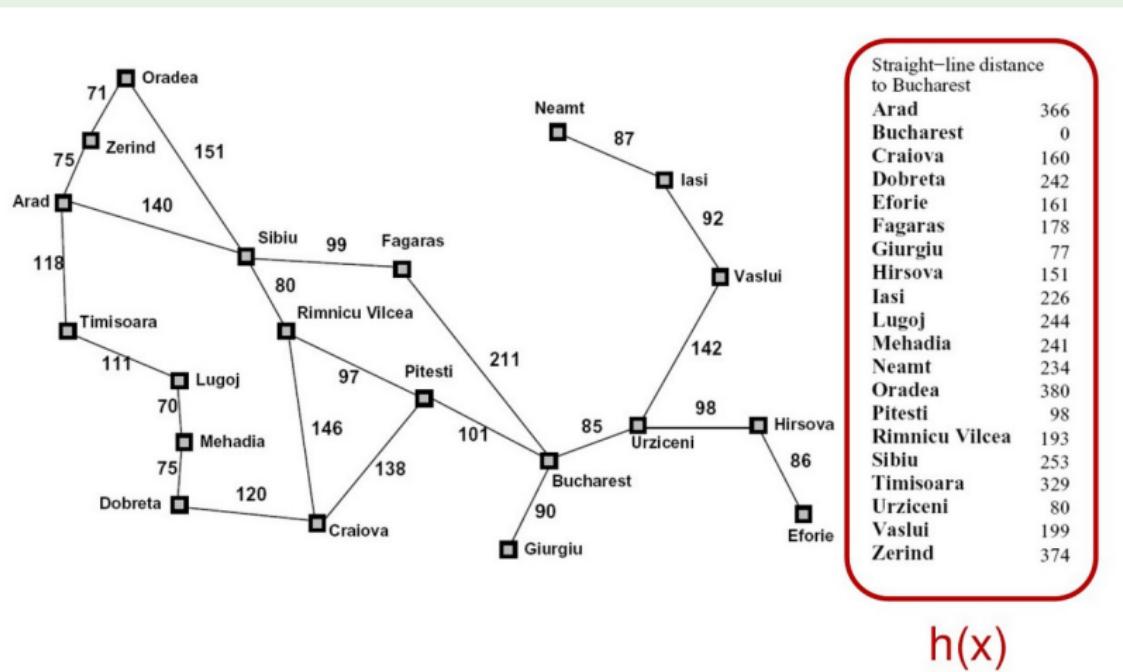
Best-first Search Strategies

General approach of informed search: Best-first search

- Best-first search: node selected for expansion based on an evaluation function $f(n)$
 - represent a cost estimate \Rightarrow choose node which appears best
 - implemented like uniform-cost search, with f instead of g
 - \Rightarrow the frontier is a priority queue sorted in decreasing order of $h(n)$
 - both tree-based and graph-based versions
 - most often f includes a heuristic function $h(n)$
- Heuristic function $h(n) \in \mathbb{R}^+$: estimated cost of the cheapest path from the state at node n to a goal state
 - $h(n) \geq 0 \forall n$
 - If G is goal, then $h(G) = 0$
 - implements extra domain knowledge
 - depends only on state, not on node
 - (e.g., independent on paths)
- Main strategies:
 - Greedy search
 - A^* search

Example: Straight-Line Distance $h_{SLD}(n)$

- $h(n) =^{\text{def}} h_{SLD}(n)$: straight-line distance heuristic
 - different from actual minimum-path distance
 - cannot be computed from the problem description itself



Outline

1 Problem-Solving Agents

2 Example Problems

3 Search Generalities

4 Uninformed Search Strategies

- Breadth-First Search
- Uniform-cost Search

Depth-First Search

Depth-Limited Search &
Iterative Deepening

5 Informed Search Strategies

- Greedy Search

A* Search Heuristic

Functions

Greedy Best-First Search Strategy (GBFS)

Greedy Best-First Search (aka Greedy Search)

- Idea: Expand first the node n with lowest estimate cost to the closest goal, $h(n)$
- Implementation: same as uniform-cost search, with $g(n) = \overset{\text{def}}{=} h(n)$
- expands the node that appears to be closest to goal
both tree and graph versions

Greedy Best-First Search Strategy (GBFS)

Greedy Best-First Search (aka Greedy Search)

- Idea: Expand first the node n with lowest estimate cost to the closest goal, $h(n)$
- Implementation: same as uniform-cost search, with $g(n) = \overset{\text{def}}{=} h(n)$
- — expands the node that appears to be closest to goal both tree and graph versions

Greedy Best-First Search Strategy (GBFS)

Greedy Best-First Search (aka Greedy Search)

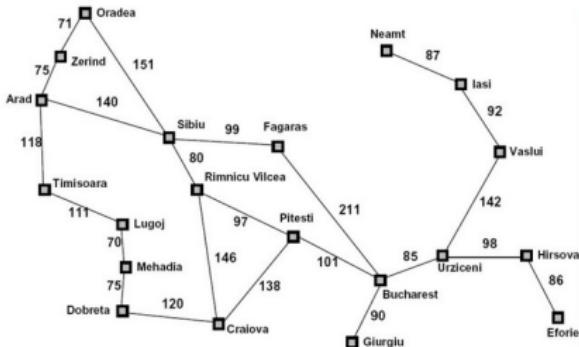
- Idea: Expand first the node n with lowest estimate cost to the closest goal, $h(n)$
- Implementation: same as uniform-cost search, with $g(n) =^{\text{def}} h(n)$
- \Rightarrow expands the node that appears to be closest to goal
both tree and graph versions

Greedy Best-First Search Strategy (GBFS)

Greedy Best-First Search (aka Greedy Search)

- Idea: Expand first the node n with lowest estimate cost to the closest goal, $h(n)$
- Implementation: same as uniform-cost search, with $g(n) =^{\text{def}} h(n)$
- \Rightarrow expands the node that appears to be closest to goal both tree and graph versions

Greedy Best-First Search Strategy: Example

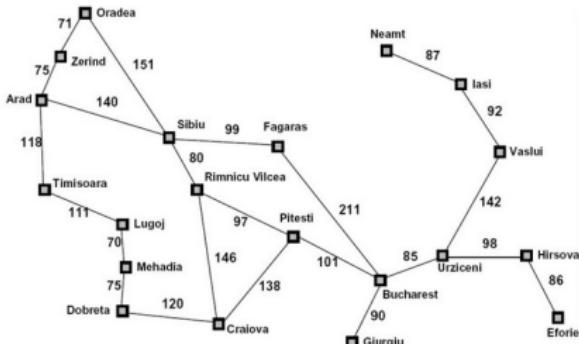


Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

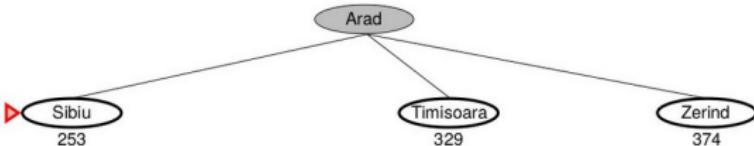
$h(x)$

► Arad
366

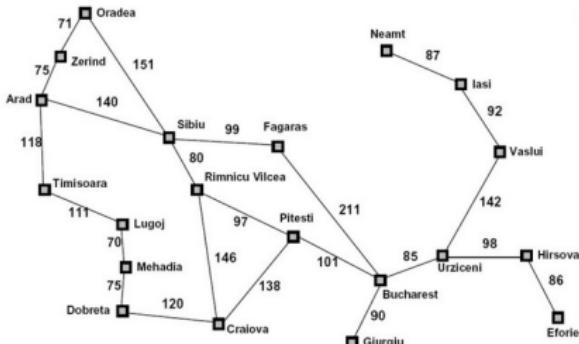
Greedy Best-First Search Strategy: Example



$h(x)$

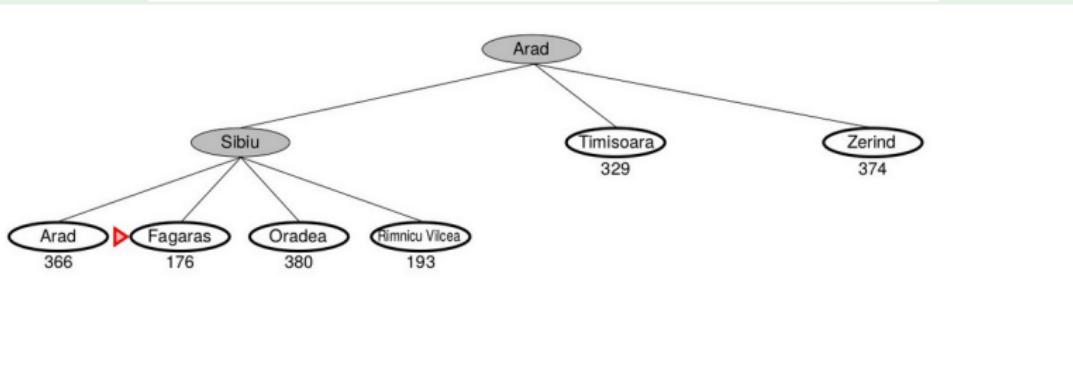


Greedy Best-First Search Strategy: Example

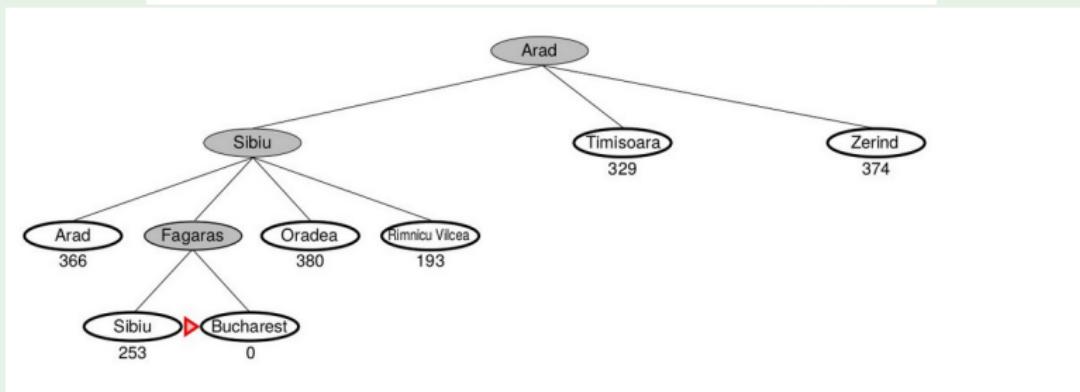
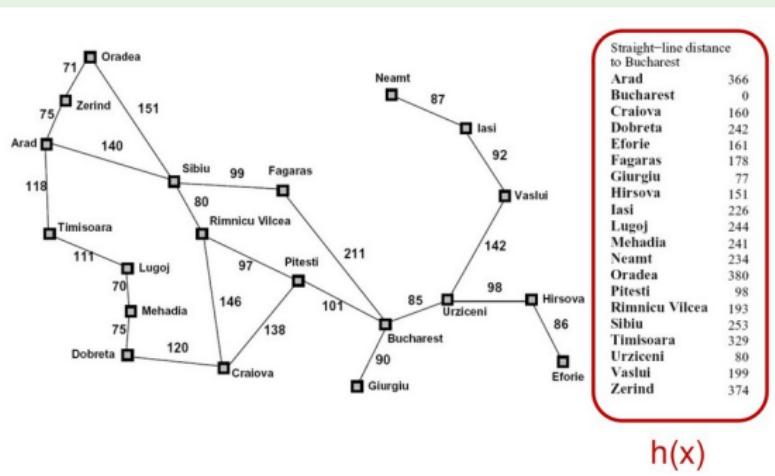


Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

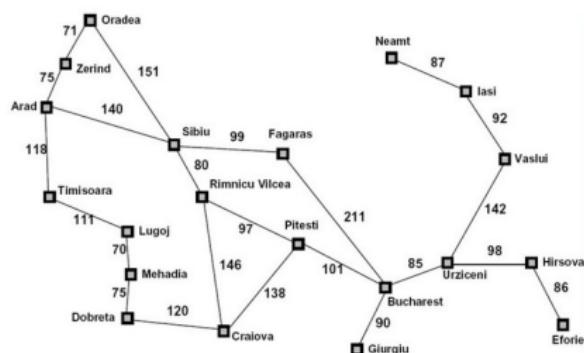
$h(x)$



Greedy Best-First Search Strategy: Example

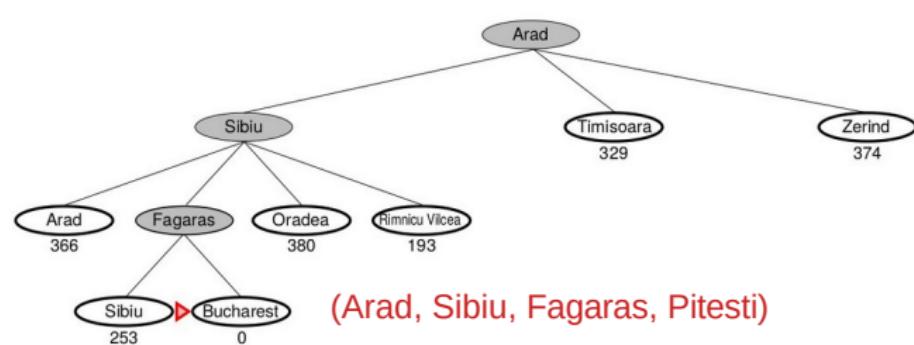


Greedy Best-First Search Strategy: Example

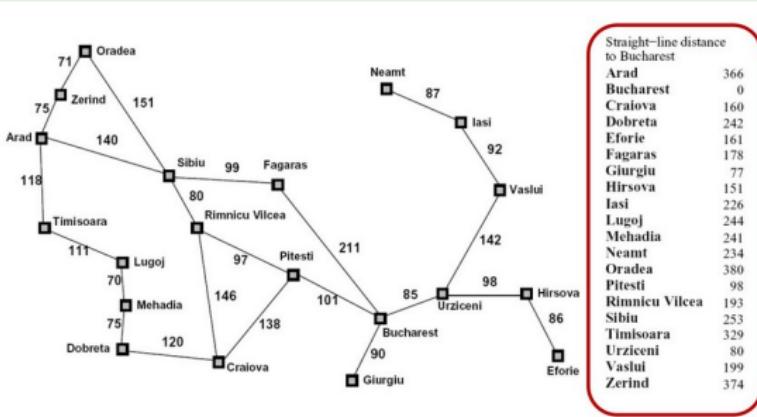


Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

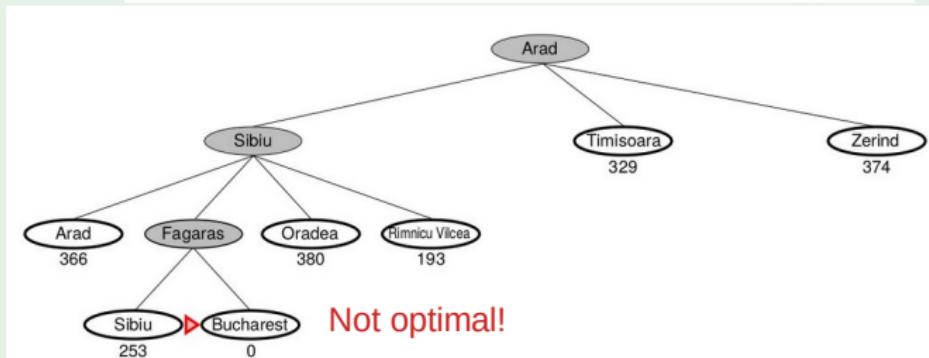
$h(x)$



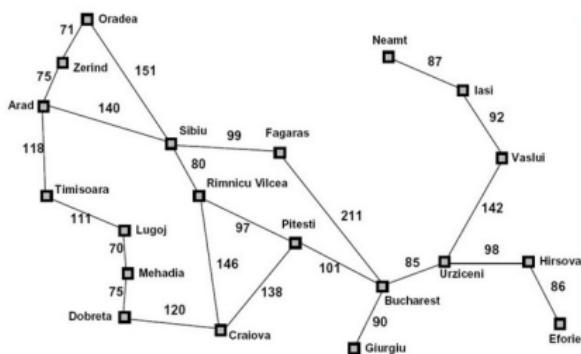
Greedy Best-First Search Strategy: Example



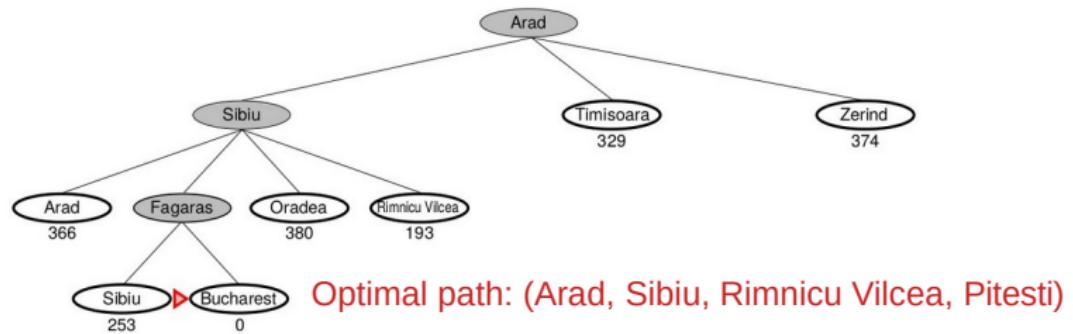
$h(x)$



Greedy Best-First Search Strategy: Example

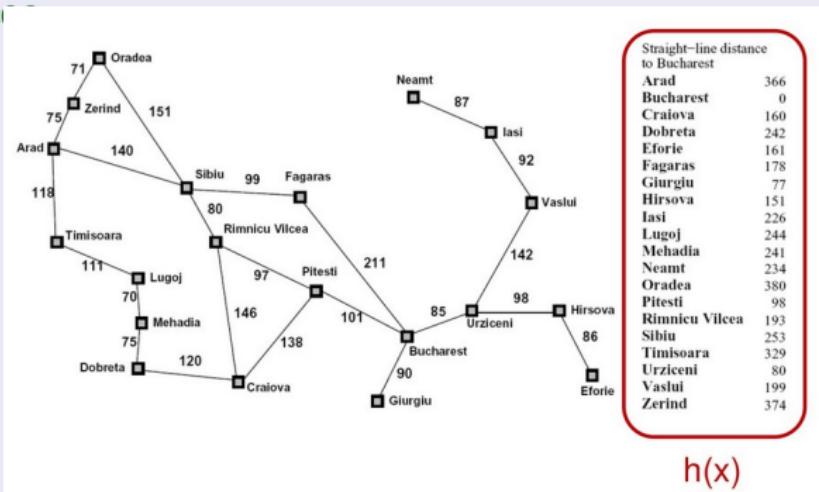


$h(x)$



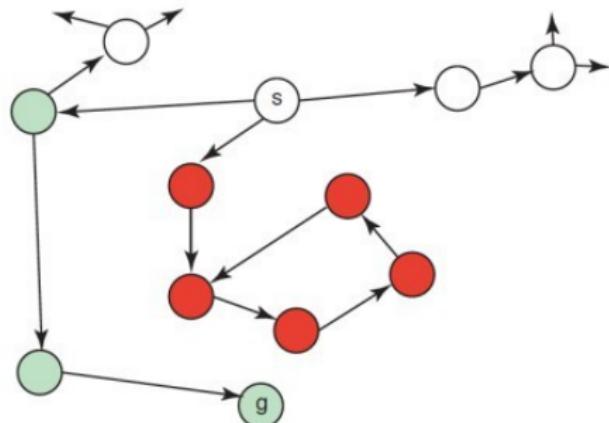
Greedy Best-First Search: (Non-)Optimality

- Greedy best-first search is not optimal
 - it is not guaranteed to find the best solution
 - it is not guaranteed to find the best path toward a solution
- picks the node with minimum (estimated) distance to goal, regardless the cost to reach it
 - Ex: when in Sibiu, it picks Fagaras rather than Rimnicu Vilcea



Greedy Best-First Search: (In-)Completeness

- Tree-based Greedy best-first search is not complete
 - may lead to infinite loops
- Graph-based version complete (if state space finite) substantially same completeness issues as DFS



(Courtesy of Maria Simi, UniPi)

Greedy Best-First Search (GBFS): Properties

- How many steps?

- in worst cases may explore all states
==> takes $O(b^d)$ time
- if good heuristics:
==> may give good improvements

- How much memory?

- max frontier size: b^d
==> $O(b^d)$ memory size

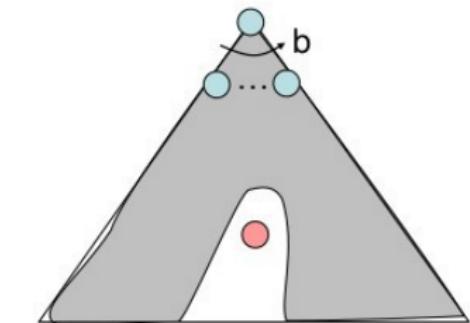
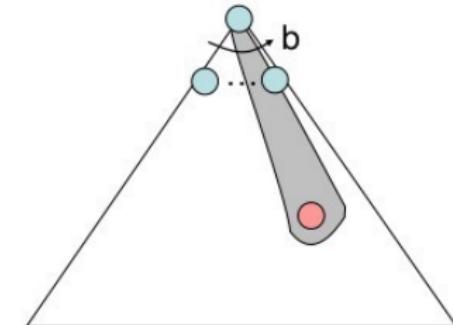
-

Is it complete?

tree: no

- graph: yes if space finite

Is it optimal?

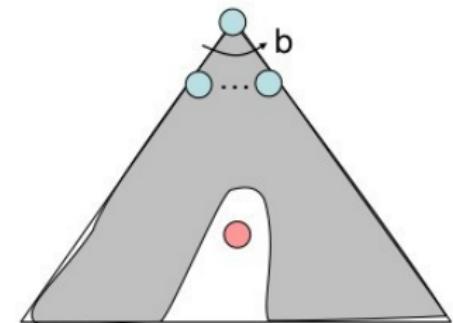
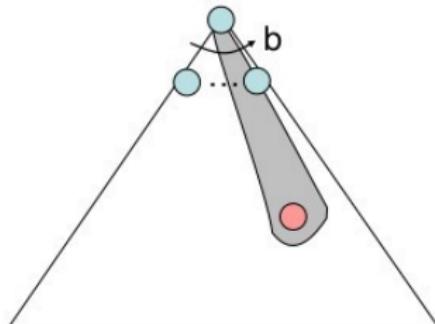


(Oc D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

Greedy Best-First Search (GBFS): Properties

- How many steps?
 - in worst cases may explore all states
==> takes $O(b^d)$ time
 - if good heuristics:
==> may give good improvements

- How much memory?
 - max frontier size: b^d
==> $O(b^d)$ memory size
- Is it complete?
 - tree: no
 - graph: yes if space finite
- Is it optimal?



(Oc D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

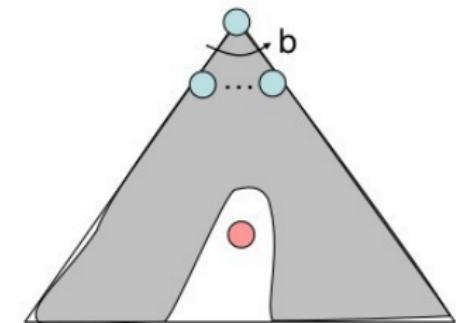
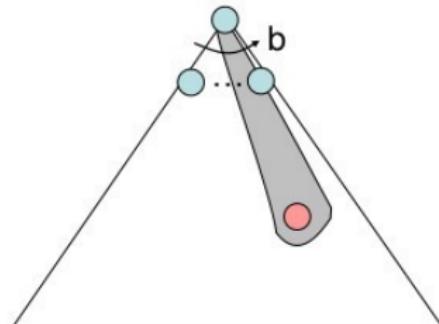
Greedy Best-First Search (GBFS): Properties

- How many steps?
 - in worst cases may explore all states
==> takes $O(b^d)$ time
 - if good heuristics:
==> may give good improvements

- How much memory?
 - max frontier size: b^d
==> $O(b^d)$ memory size
 - Is it complete?

- tree: no
- graph: yes if space finite

- Is it optimal?



(Oc D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

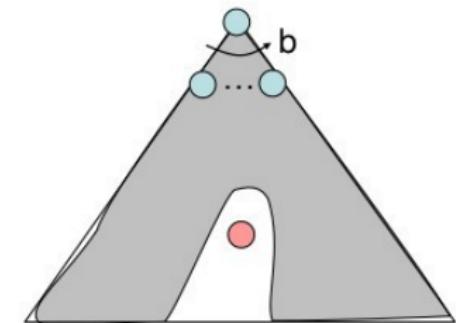
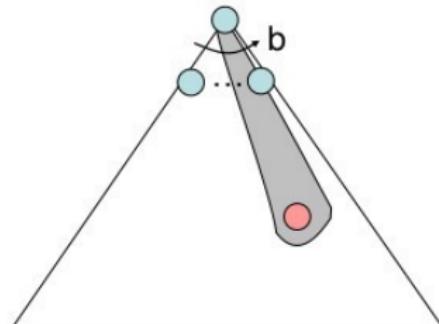
Greedy Best-First Search (GBFS): Properties

- How many steps?
 - in worst cases may explore all states
==> takes $O(b^d)$ time
 - if good heuristics:
==> may give good improvements

- How much memory?
 - max frontier size: b^d
==> $O(b^d)$ memory size

- Is it complete?
 - tree: no
 - graph: yes if space finite

- Is it optimal?



(Oc D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

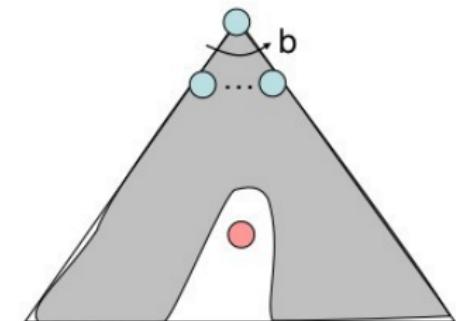
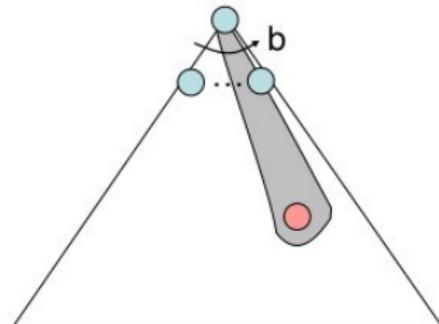
Greedy Best-First Search (GBFS): Properties

- How many steps?
 - in worst cases may explore all states
==> takes $O(b^d)$ time
 - if good heuristics:
==> may give good improvements

- How much memory?
 - max frontier size: b^d
==> $O(b^d)$ memory size

- Is it complete?
 - tree: no
 - graph: yes if space finite

- Is it optimal?



(Oc D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

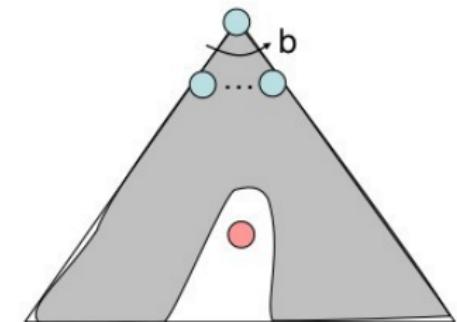
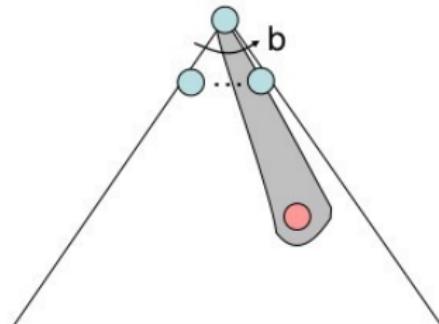
Greedy Best-First Search (GBFS): Properties

- How many steps?
 - in worst cases may explore all states
==> takes $O(b^d)$ time
 - if good heuristics:
==> may give good improvements

- How much memory?
 - max frontier size: b^d
==> $O(b^d)$ memory size

- Is it complete?
 - tree: no
 - graph: yes if space finite

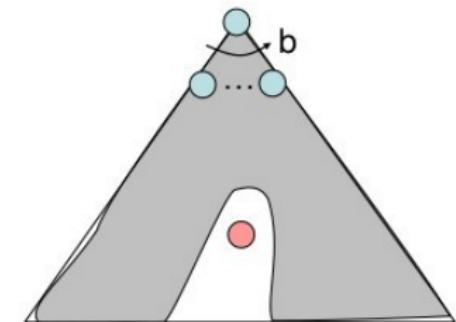
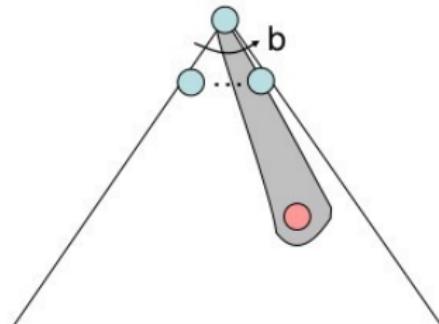
Is it optimal?



(Oc D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

Greedy Best-First Search (GBFS): Properties

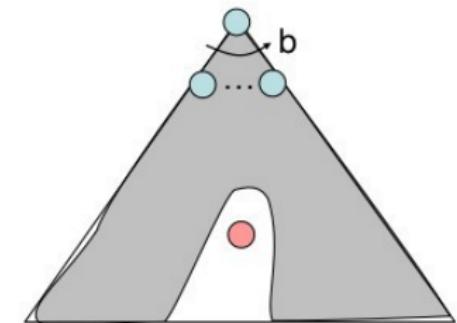
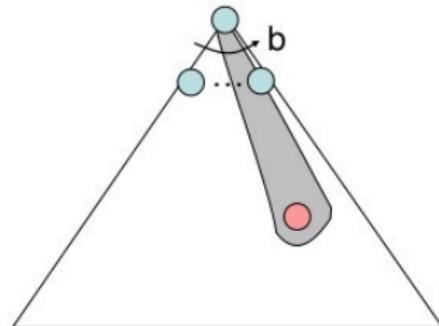
- How many steps?
 - in worst cases may explore all states
==> takes $O(b^d)$ time
 - if good heuristics:
==> may give good improvements
- How much memory?
 - max frontier size: b^d
==> $O(b^d)$ memory size
 - Is it complete?
 - tree: no
 - graph: yes if space finite
- Is it optimal?



(Oc D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

Greedy Best-First Search (GBFS): Properties

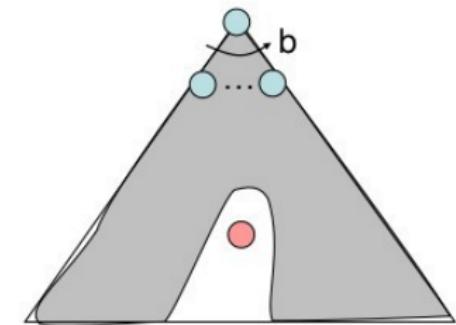
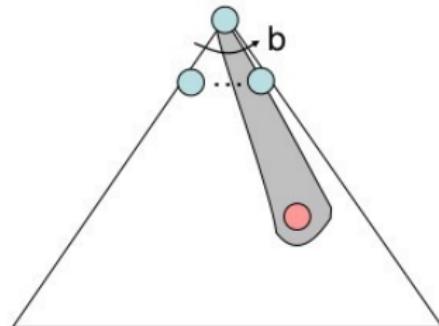
- How many steps?
 - in worst cases may explore all states
==> takes $O(b^d)$ time
 - if good heuristics:
==> may give good improvements
- How much memory?
 - max frontier size: b^d
==> $O(b^d)$ memory size
 - Is it complete?
 - tree: no
 - graph: yes if space finite
- Is it optimal?



(Oc D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

Greedy Best-First Search (GBFS): Properties

- How many steps?
 - in worst cases may explore all states
==> takes $O(b^d)$ time
 - if good heuristics:
==> may give good improvements
- How much memory?
 - max frontier size: b^d
==> $O(b^d)$ memory size
 - Is it complete?
 - tree: no
 - graph: yes if space finite
- Is it optimal?



(Oc D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

Outline

1 Problem-Solving Agents

2 Example Problems

3 Search Generalities

4 Uninformed Search Strategies

- Breadth-First Search
- Uniform-cost Search

Depth-First Search

Depth-Limited Search &

Iterative Deepening

5 Informed Search Strategies

- Greedy Search

A* Search Heuristic

Functions

A* Search Strategy

A* Search

- Best-known form of best-first search
- Idea: avoid expanding paths that are already expensive
- Combine Uniform-Cost and Greedy search: $f(n) = g(n) + h(n)$
 - $g(n)$: cost so far to reach n
 - $h(n)$: estimated cost to goal from n
 - $f(n)$: estimated total cost of path through n to goal

==> Expand first the node n with lowest estimated cost of

- the cheapest solution through n

Implementation: same as uniform-cost search, with $g(n) + h(n)$ instead of $g(n)$

A* Search Strategy

A* Search

- Best-known form of best-first search
- Idea: avoid expanding paths that are already expensive
- Combine Uniform-Cost and Greedy search: $f(n) = g(n) + h(n)$
 - $g(n)$: cost so far to reach n
 - $h(n)$: estimated cost to goal from n
 - $f(n)$: estimated total cost of path through n to goal

==> Expand first the node n with lowest estimated cost of

- the cheapest solution through n

Implementation: same as uniform-cost search, with $g(n) + h(n)$ instead of $g(n)$

A* Search Strategy

A* Search

- Best-known form of best-first search
- Idea: avoid expanding paths that are already expensive
- Combine Uniform-Cost and Greedy search: $f(n) = g(n) + h(n)$
 - $g(n)$: cost so far to reach n
 - $h(n)$: estimated cost to goal from n
 - $f(n)$: estimated total cost of path through n to goal

==> Expand first the node n with lowest estimated cost of

- the cheapest solution through n

Implementation: same as uniform-cost search, with $g(n) + h(n)$ instead of $g(n)$

A* Search Strategy

A* Search

- Best-known form of best-first search
- Idea: avoid expanding paths that are already expensive
- Combine Uniform-Cost and Greedy search: $f(n) = g(n) + h(n)$
 - $g(n)$: cost so far to reach n
 - $h(n)$: estimated cost to goal from n
 - $f(n)$: estimated total cost of path through n to goal

==> Expand first the node n with lowest estimated cost of

- the cheapest solution through n

Implementation: same as uniform-cost search, with $g(n) + h(n)$ instead of $g(n)$

A* Search Strategy

A* Search

- Best-known form of best-first search
- Idea: avoid expanding paths that are already expensive
- Combine Uniform-Cost and Greedy search: $f(n) = g(n) + h(n)$
 - $g(n)$: cost so far to reach n
 - $h(n)$: estimated cost to goal from n
 - $f(n)$: estimated total cost of path through n to goal

==> Expand first the node n with lowest estimated cost of
the cheapest solution through n

Implementation: same as uniform-cost search, with $g(n) + h(n)$

instead of $g(n)$

A* Search Strategy

A* Search

- Best-known form of best-first search
- Idea: avoid expanding paths that are already expensive
- Combine Uniform-Cost and Greedy search: $f(n) = g(n) + h(n)$
 - $g(n)$: cost so far to reach n
 - $h(n)$: estimated cost to goal from n
 - $f(n)$: estimated total cost of path through n to goal

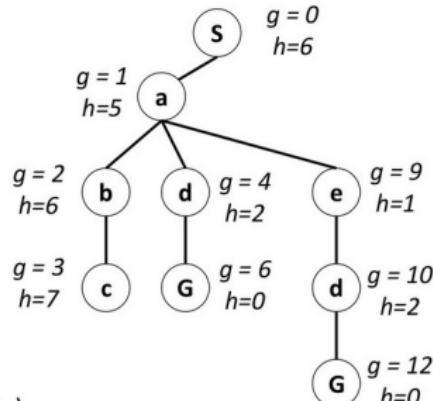
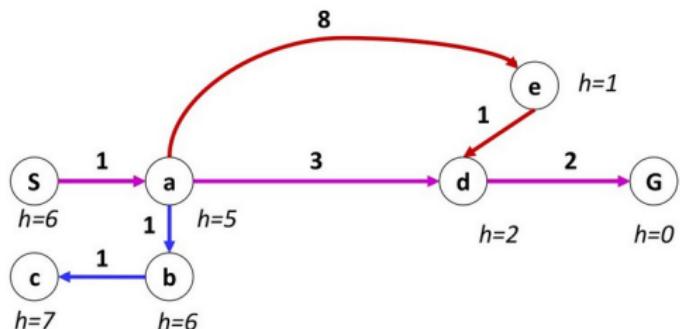
==> Expand first the node n with lowest estimated cost of

- the cheapest solution through n

Implementation: same as uniform-cost search, with $g(n) + h(n)$ instead of $g(n)$

A* Search Strategy: Example

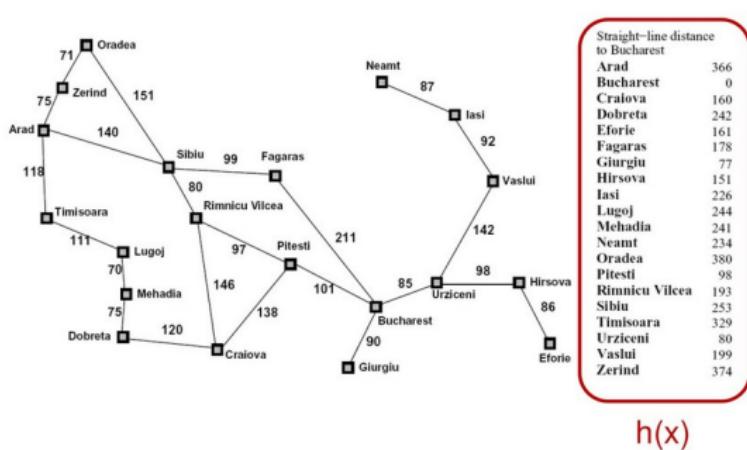
- Uniform-cost orders by path cost, or *backward cost* $g(n)$
- Greedy orders by goal proximity, or *forward cost* $h(n)$



- A* Search orders by the sum: $f(n) = g(n) + h(n)$

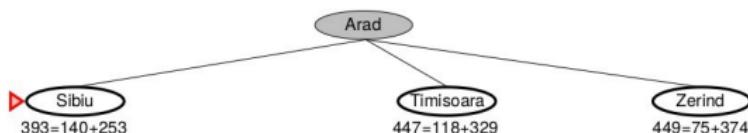
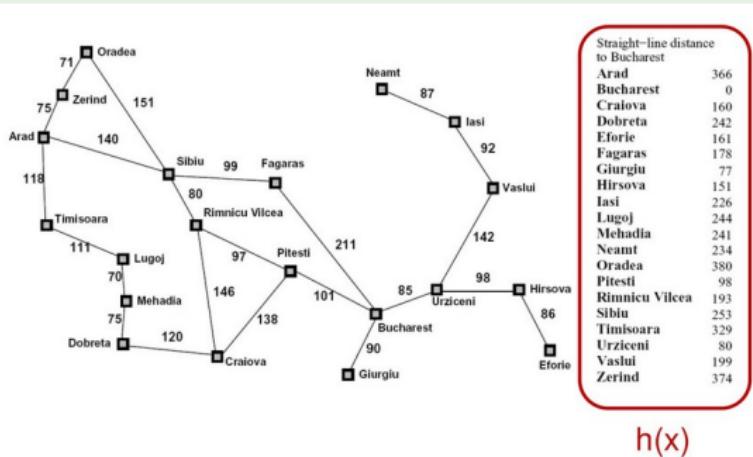
(Oc D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

A* Search Strategy: Example

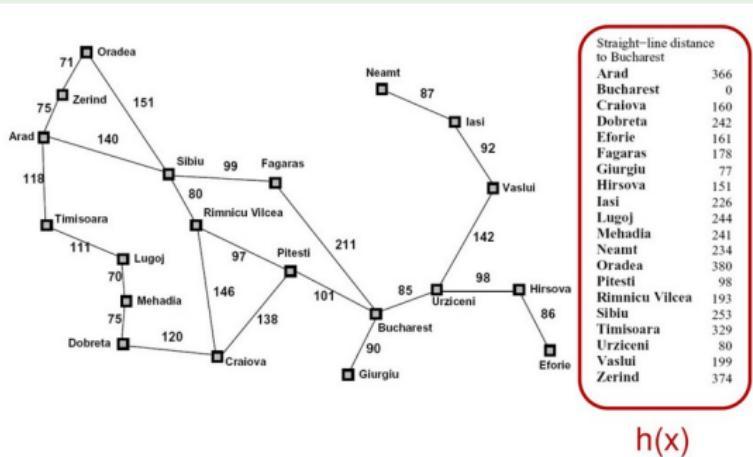


► Arad

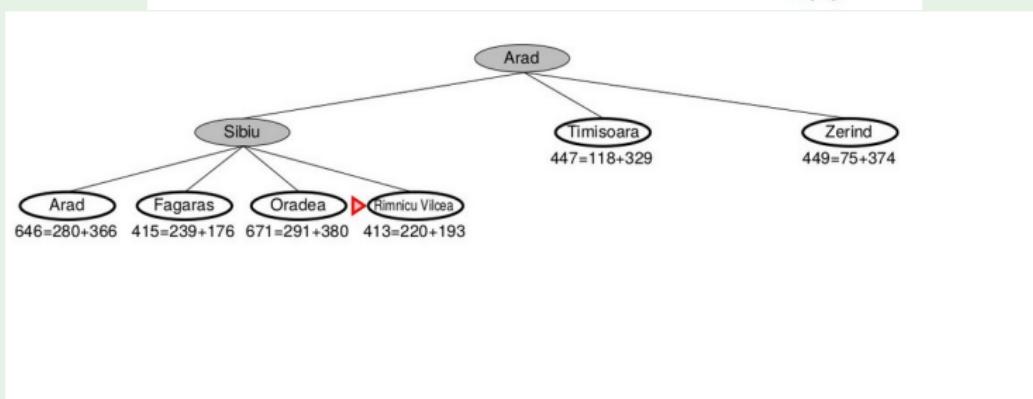
A* Search Strategy: Example



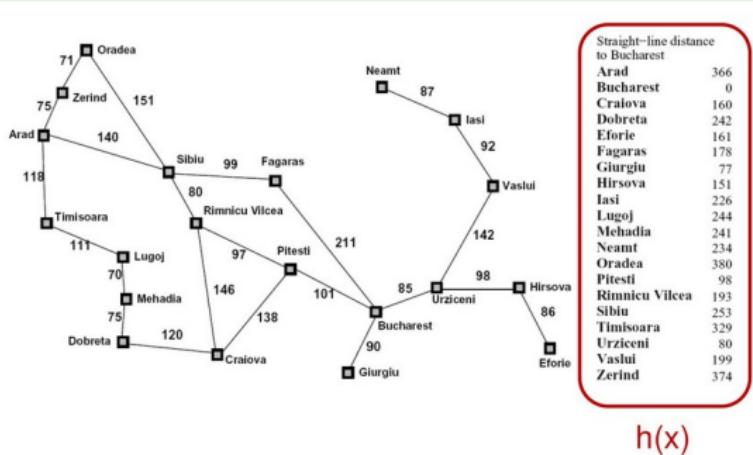
A* Search Strategy: Example



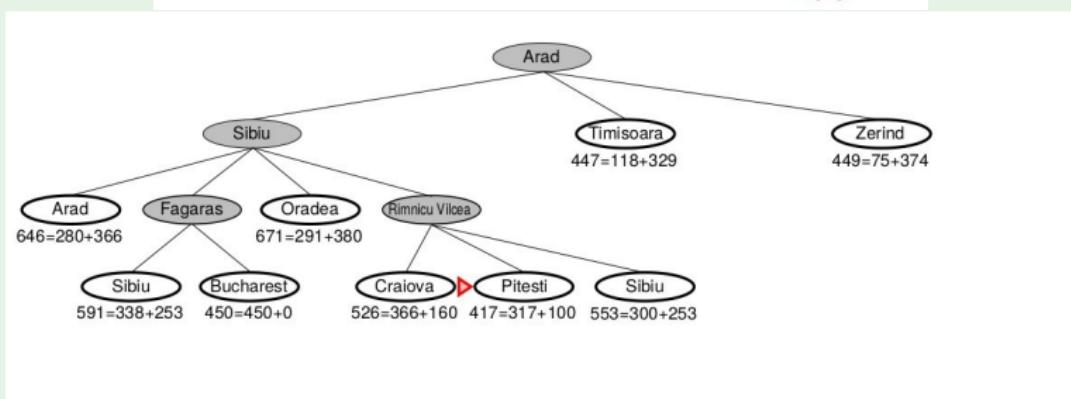
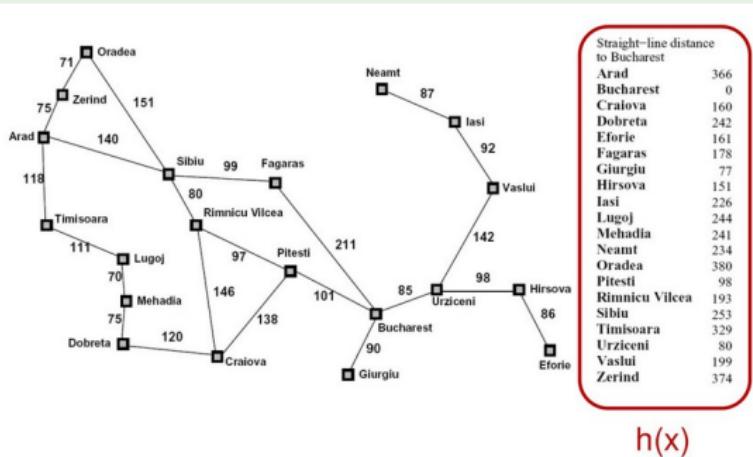
$h(x)$



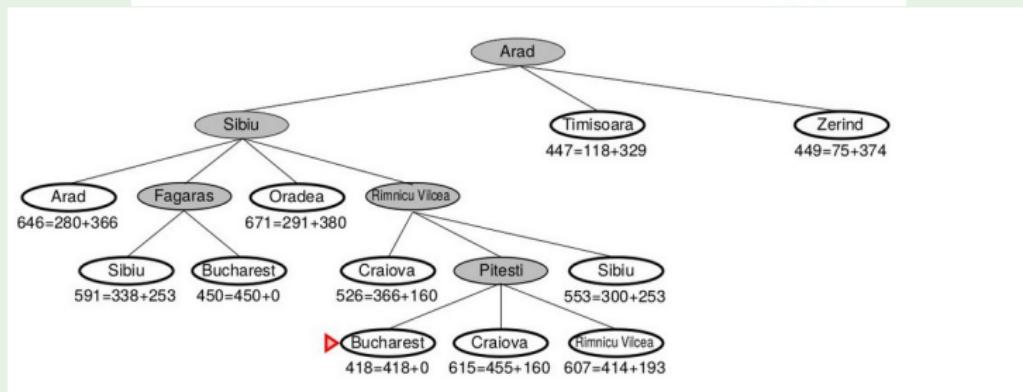
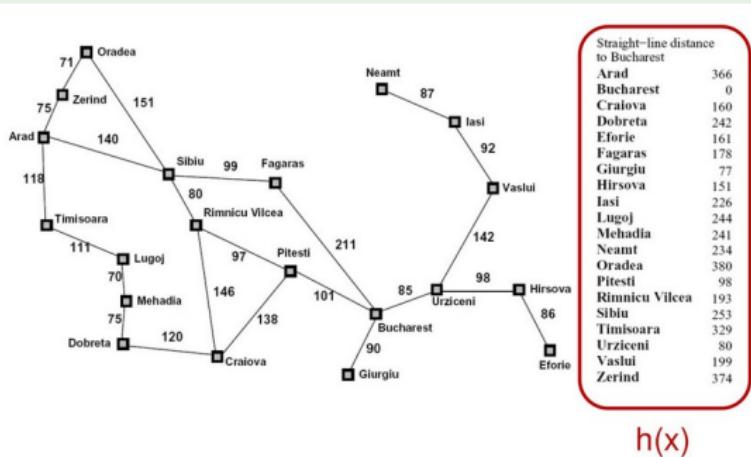
A* Search Strategy: Example



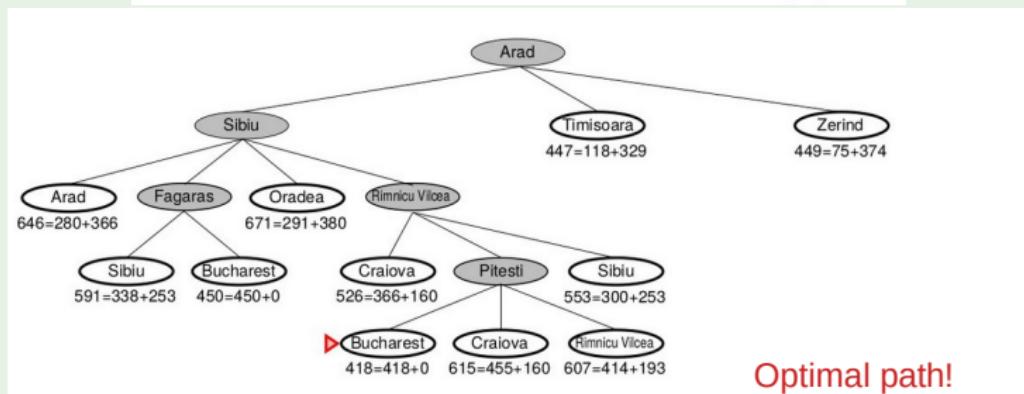
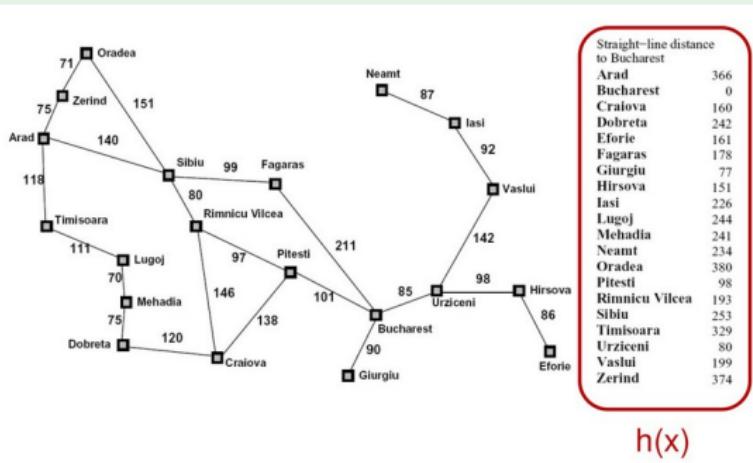
A* Search Strategy: Example



A* Search Strategy: Example



A* Search Strategy: Example



A* Search: Admissible and Consistent Heuristics

Admissible heuristics $h(n)$

- $h(n)$ is **admissible** (aka **optimistic**) iff it never overestimates the cost to reach the goal:
 - $h(n) \leq h^*(n)$ where $h^*(n)$ is the true cost from n
 - ex: **the straight-line distance $h_{SDL}()$ to Bucharest**

Consistent heuristics $h(n)$

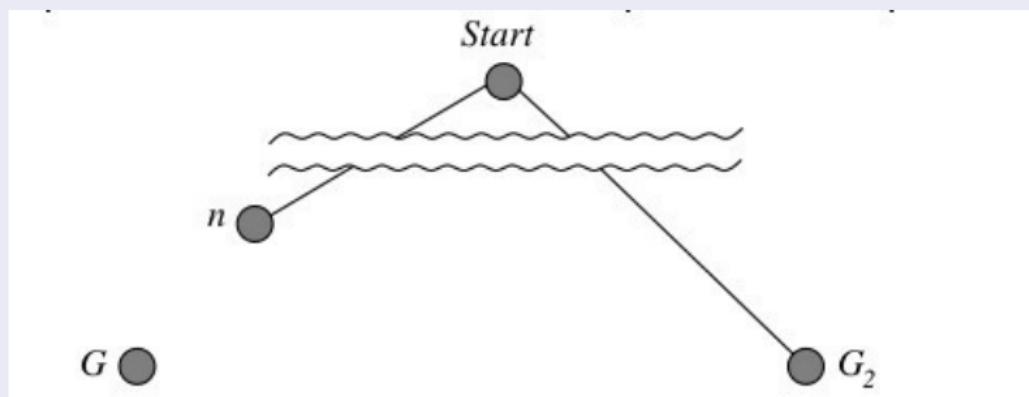
- $h(n)$ is **consistent** (aka **monotonic**) iff, for every successor n' of n generated by any action a with step cost $c(n, a, n')$,
$$h(n) \leq c(n, a, n') + h(n')$$
 - verifies the triangular inequality
 - ex: **the straight-line distance $h_{SDL}()$ to Bucharest**

A* Tree Search: Optimality

If $h(n)$ is admissible, then A* tree search is optimal

- Suppose some sub-optimal goal G_2 is in the frontier queue.
- n unexpanded node on a shortest path to an optimal goal G .
- then:
$$\begin{aligned} f(G_2) &= g(G_2) \text{ since } h(G_2) = 0 \\ &> g(G) \text{ since } G_2 \text{ sub-optimal} \\ &\geq f(n) \text{ since } h \text{ is admissible} \end{aligned}$$

\Rightarrow A* will not pick G_2 from the frontier queue before n



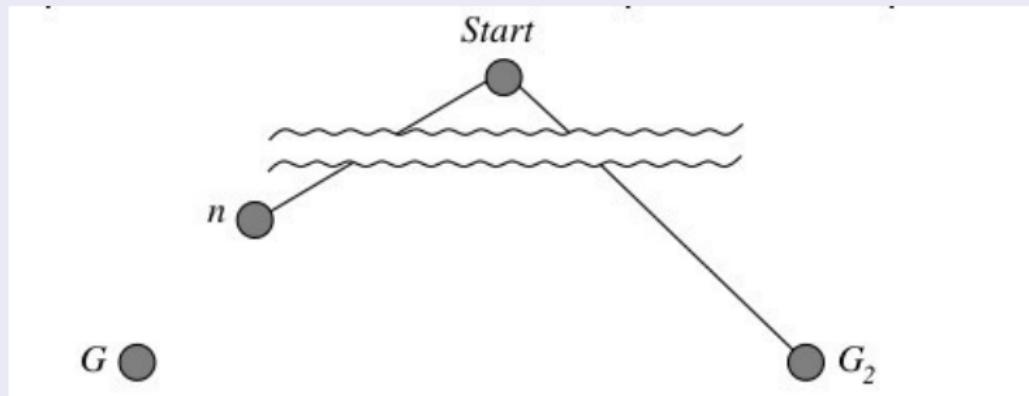
A* Tree Search: Optimality

If $h(n)$ is admissible, then A* tree search is optimal

- Suppose some sub-optimal goal G_2 is in the frontier queue.
- n unexpanded node on a shortest path to an optimal goal G .
- then:

$$\begin{aligned} f(G_2) &= g(G_2) \text{ since } h(G_2) = 0 \\ &> g(G) \text{ since } G_2 \text{ sub-optimal} \\ &\geq f(n) \text{ since } h \text{ is admissible} \end{aligned}$$

\Rightarrow A* will not pick G_2 from the frontier queue before n

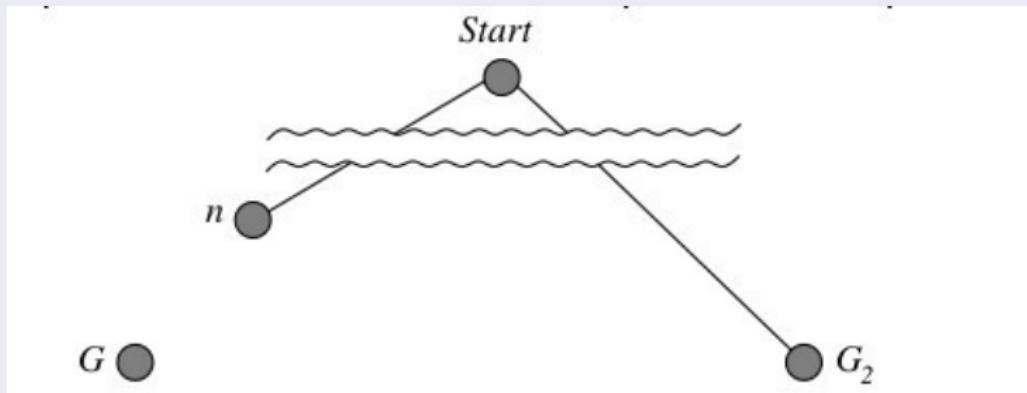


A* Tree Search: Optimality

If $h(n)$ is admissible, then A* tree search is optimal

- Suppose some sub-optimal goal G_2 is in the frontier queue.
- n unexpanded node on a shortest path to an optimal goal G .
- then:
$$\begin{aligned} f(G_2) &= g(G_2) \text{ since } h(G_2) = 0 \\ &> g(G) \text{ since } G_2 \text{ sub-optimal} \\ &\geq f(n) \text{ since } h \text{ is admissible} \end{aligned}$$

\Rightarrow A* will not pick G_2 from the frontier queue before n

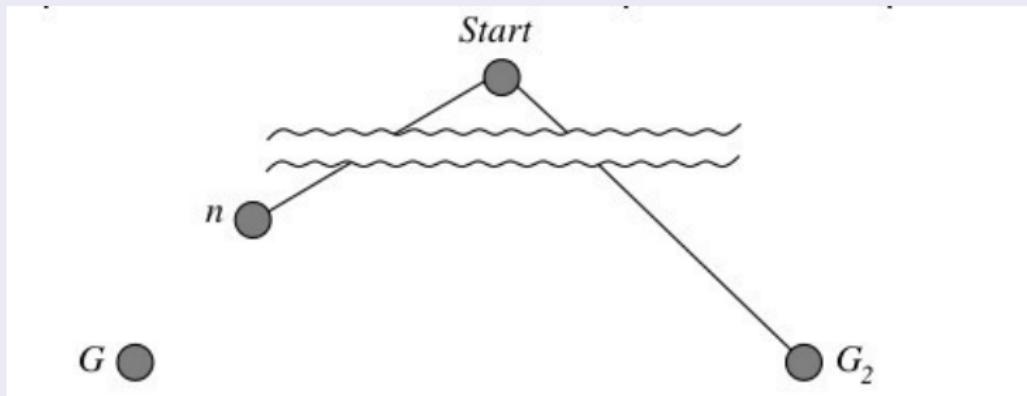


A* Tree Search: Optimality

If $h(n)$ is admissible, then A* tree search is optimal

- Suppose some sub-optimal goal G_2 is in the frontier queue.
- n unexpanded node on a shortest path to an optimal goal G .
- then:
$$\begin{aligned} f(G_2) &= g(G_2) \text{ since } h(G_2) = 0 \\ &> g(G) \text{ since } G_2 \text{ sub-optimal} \\ &\geq f(n) \text{ since } h \text{ is admissible} \end{aligned}$$

\Rightarrow A* will not pick G_2 from the frontier queue before n



A* Graph Search: Properties

Properties

- ① if $h(n)$ is consistent, then $h(n)$ is admissible (straightforward)
- ② If $h(n)$ is consistent, then $f(n)$ is non-decreasing along any path:
 - $f(n^m) = g(n^m) + h(n^m) = g(n) + c(n, a, n^m) + h(n^m) \geq g(n) + h(n) = f(n)$
- ③

If (Graph) A* selects a node n from the frontier, then the optimal path to that node has been found

- if not so, there would be a node n^m in the frontier on the
- optimal path to n (because of the graph separation property)
 - since f is non-decreasing along any path, $f(n^m) \leq f(n)$
- \Rightarrow ~~the second bullet is true because the path before $f(n^m) < f(n)$~~



\Rightarrow A* (graph search) expands nodes in non-decreasing order of f

A* Graph Search: Properties

Properties

- ① if $h(n)$ is consistent, then $h(n)$ is admissible (straightforward)
- ② If $h(n)$ is consistent, then $f(n)$ is non-decreasing along any path:
 - $f(n^m) = g(n^m) + h(n^m) = g(n) + c(n, a, n^m) + h(n^m) \geq g(n) + h(n) = f(n)$
- ③ If (Graph) A* selects a node n from the frontier, then the optimal path to that node has been found
 - if not so, there would be a node n^m in the frontier on the
 - optimal path to n (because of the graph separation property)
 - since f is non-decreasing along any path, $f(n^m) \leq f(n)$
 - \Rightarrow ~~the cost difference between the optimal path before $f(n^m) < f(n)$~~



\Rightarrow A* (graph search) expands nodes in non-decreasing order of f

A* Graph Search: Properties

Properties

- ① if $h(n)$ is consistent, then $h(n)$ is admissible (straightforward)
- ② If $h(n)$ is consistent, then $f(n)$ is non-decreasing along any path:
 - $f(n^m) = g(n^m) + h(n^m) = g(n) + c(n, a, n^m) + h(n^m) \geq g(n) + h(n) = f(n)$
- ③

If (Graph) A* selects a node n from the frontier, then the optimal path to that node has been found

- if not so, there would be a node n^m in the frontier on the
- optimal path to n (because of the graph separation property)
 - since f is non-decreasing along any path, $f(n^m) \leq f(n)$
- \Rightarrow ~~the second bullet is true because the path before $f(n^m) < f(n)$~~



\Rightarrow A* (graph search) expands nodes in non-decreasing order of f

A* Graph Search: Properties

Properties

- ① if $h(n)$ is consistent, then $h(n)$ is admissible (straightforward)
- ② If $h(n)$ is consistent, then $f(n)$ is non-decreasing along any path:
 - $f(n^m) = g(n^m) + h(n^m) = g(n) + c(n, a, n^m) + h(n^m) \geq g(n) + h(n) = f(n)$
- ③ If (Graph) A* selects a node n from the frontier, then the optimal path to that node has been found

if not so, there would be a node n^m in the frontier on the
optimal path to n (because of the graph separation property)
since f is non-decreasing along any path, $f(n^m) \leq f(n)$
 \Rightarrow ~~Since n^m is on the optimal path before n , therefore~~ $f(n^m) < f(n)$

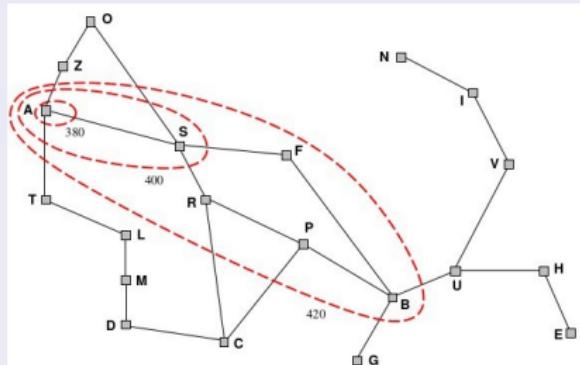


\Rightarrow A* (graph search) expands nodes in non-decreasing order of f

A* Graph Search: Optimality

If $h(n)$ is consistent, then A* graph search is optimal

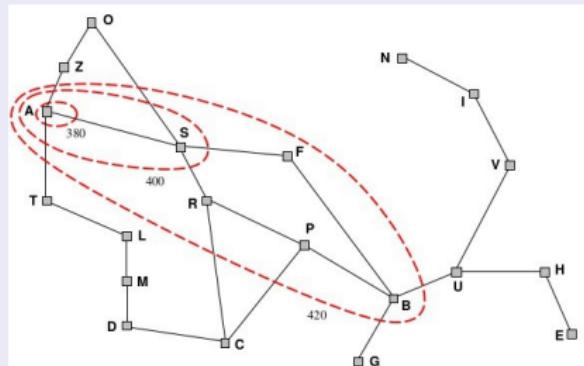
- A* expands nodes in order of non-decreasing f value
- Gradually adds “f-contours” of nodes (as BFS adds layers)
 - contour i has all nodes with $f = f_i$, s.t. $f_i < f_{i+1}$
 - cannot expand contour f_{i+1} until contour f_i is fully expanded
- If C^* is the cost of the optimal solution path
 - 1 A* expands all nodes s.t. $f(n) < C^*$
 - 2 A* might expand some of the nodes on “goal contour” s.t. $f(n) = C^*$ before selecting a goal node.
 - 3 A* does not expand nodes s.t. $f(n) > C^*$ (pruning)



A* Graph Search: Optimality

If $h(n)$ is consistent, then A* graph search is optimal

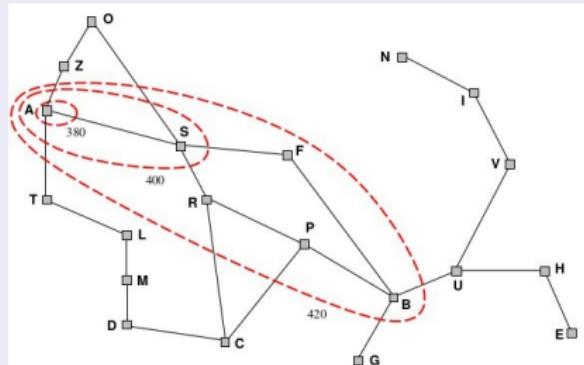
- A* expands nodes in order of non-decreasing f value
- Gradually adds “f-contours” of nodes (as BFS adds layers)
 - contour i has all nodes with $f = f_i$, s.t. $f_i < f_{i+1}$
 - cannot expand contour f_{i+1} until contour f_i is fully expanded
- If C^* is the cost of the optimal solution path
 - 1 A* expands all nodes s.t. $f(n) < C^*$
 - 2 A* might expand some of the nodes on “goal contour” s.t. $f(n) = C^*$ before selecting a goal node.
 - 3 A* does not expand nodes s.t. $f(n) > C^*$ (pruning)



A* Graph Search: Optimality

If $h(n)$ is consistent, then A* graph search is optimal

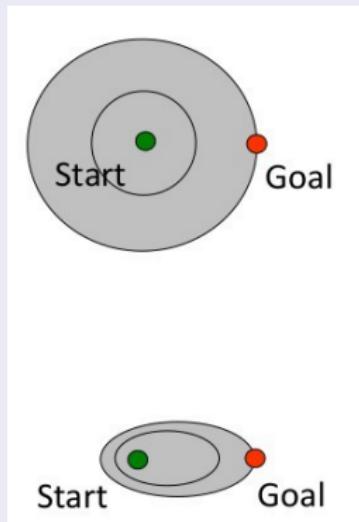
- A* expands nodes in order of non-decreasing f value
- Gradually adds “f-contours” of nodes (as BFS adds layers)
 - contour i has all nodes with $f = f_i$, s.t. $f_i < f_{i+1}$
 - cannot expand contour f_{i+1} until contour f_i is fully expanded
- If C^* is the cost of the optimal solution path
 - 1 A* expands all nodes s.t. $f(n) < C^*$
 - 2 A* might expand some of the nodes on “goal contour” s.t. $f(n) = C^*$ before selecting a goal node.
 - 3 A* does not expand nodes s.t. $f(n) > C^*$ (pruning)



UCS vs A* Contours

Intuition

- UCS expands equally in all “directions”
- A* expands mainly toward the goal

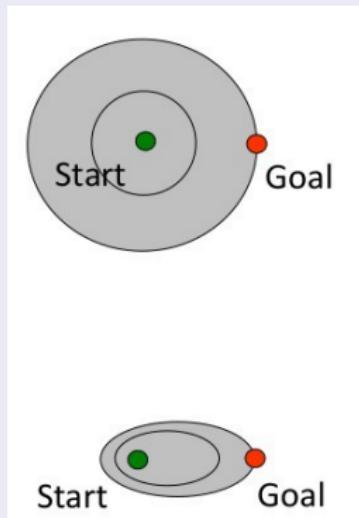


(Oc D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

UCS vs A* Contours

Intuition

- UCS expands equally in all “directions”
- A* expands mainly toward the goal



(Oc D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

A* Search: Completeness

If all step costs exceed some finite c and b is finite,
then there are only finitely many nodes n s.t. $f(n) \leq C^*$

$\implies A^*$ is complete.

A* Search: Properties

Let $c = \frac{h^+ - h}{h^*}$ (relative error)

b^e : effective branching factor

How many steps?

takes $O((b^c)^d)$ time

if good heuristics, may give dramatic improvements

How much memory?

Keeps all nodes in memory

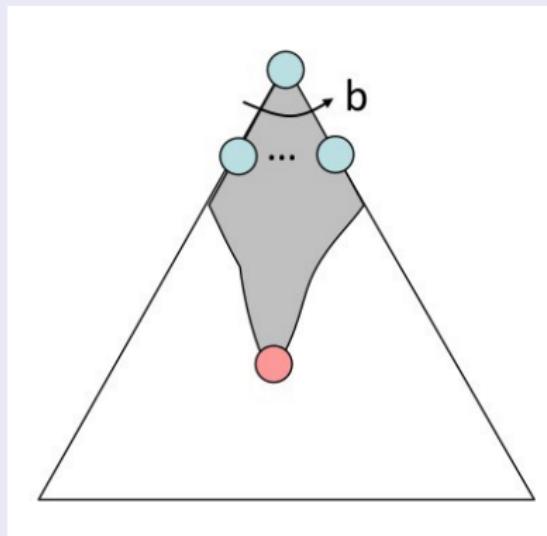
$\Rightarrow O(b^e)$ memory size

(like UCS)

- Is it complete?

- Yes

- Is it optimal?



(Oc D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

Memory requirement is a major problem also for

A*

A* Search: Properties

Let $c = \frac{h^+ - h}{h^*}$ (relative error)

b^e effective branching factor

- How many steps?

- takes $O((b^c)^d)$ time

- if good heuristics, may give dramatic improvements

- How much memory?

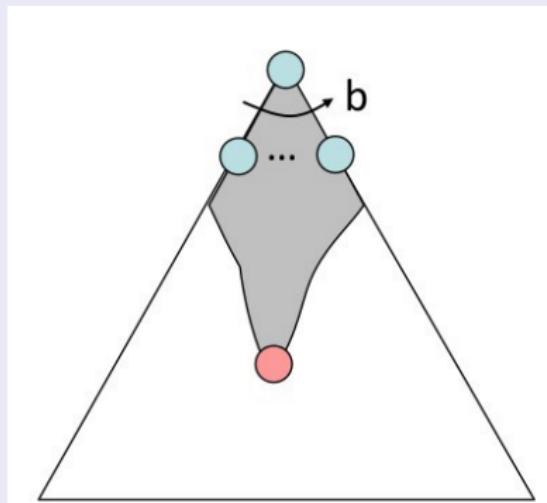
- Keeps all nodes in memory
 $\Rightarrow O(b^e)$ memory size

- (like UCS)

- Is it complete?

- yes

- Is it optimal?



(Oc D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

Memory requirement is a major problem also for

A*

A* Search: Properties

Let $c = \frac{h^+ - h}{h^*}$ (relative error)

b^e effective branching factor

How many steps?

takes $O((b^c)^d)$ time

if good heuristics, may give dramatic improvements

How much memory?

Keeps all nodes in memory

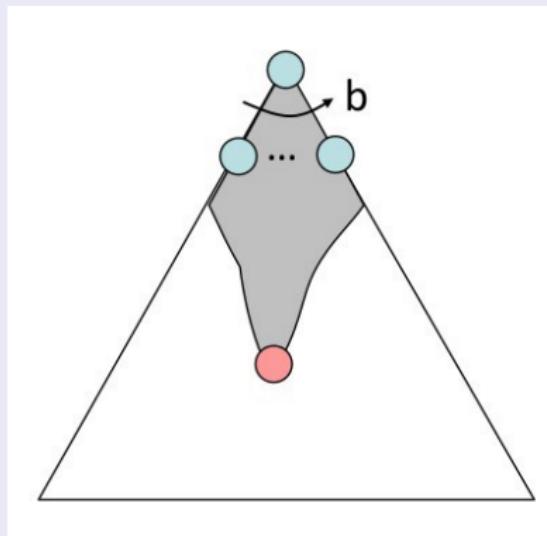
$\Rightarrow O(b^d)$ memory size

(like UCS)

Is it complete?

Yes

Is it optimal?



(Oc D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

Memory requirement is a major problem also for

A*

A* Search: Properties

Let $c = \frac{h^+ - h}{h^*}$ (relative error)

b^e effective branching factor

How many steps?

takes $O((b^c)^d)$ time

if good heuristics, may give dramatic improvements

How much memory?

Keeps all nodes in memory

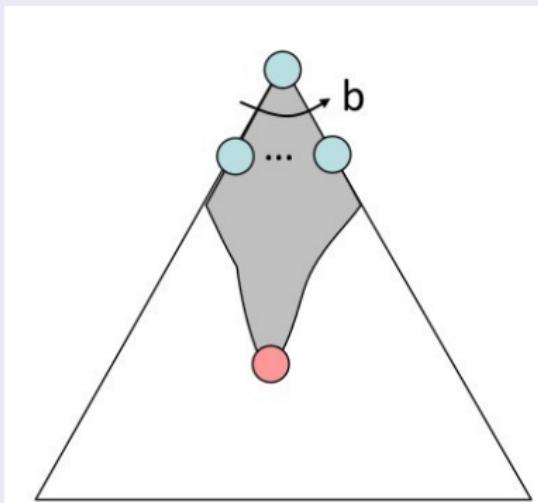
$\Rightarrow O(b^d)$ memory size

(like UCS)

Is it complete?

Yes

Is it optimal?



(Oc D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

Memory requirement is a major problem also for

A*

A* Search: Properties

Let $c = \frac{h^+ - h}{h^*}$ (relative error)

b^e effective branching factor

How many steps?

takes $O((b^c)^d)$ time

- if good heuristics, may give dramatic improvements

How much memory?

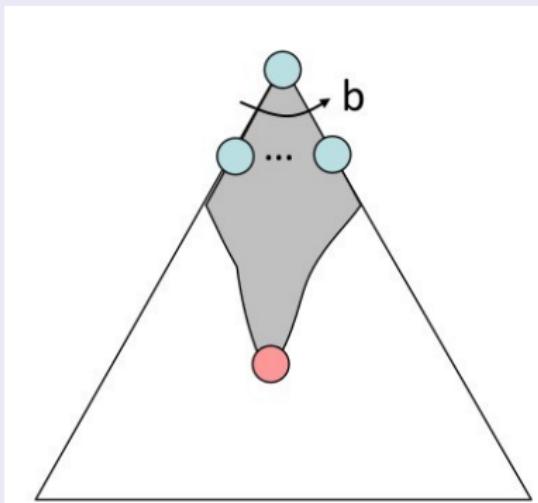
Keeps all nodes in memory
 $\Rightarrow O(b^d)$ memory size

- (like UCS)

- Is it complete?

- yes

- Is it optimal?



(Oc D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

Memory requirement is a major problem also for

A*

A* Search: Properties

Let $c = \frac{h^+ - h}{h^*}$ (relative error)

b^e effective branching factor

- How many steps?

- takes $O((b^c)^d)$ time

- if good heuristics, may give dramatic improvements

- How much memory?

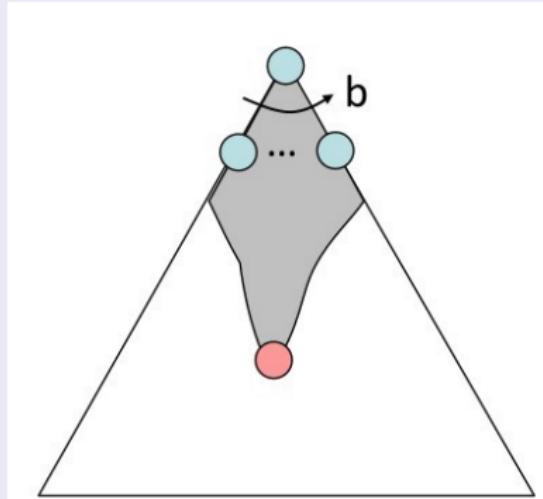
- Keeps all nodes in memory
 $\Rightarrow O(b^d)$ memory size

- (like UCS)

- Is it complete?

- yes

- Is it optimal?



(Oc D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

Memory requirement is a major problem also for

A*

A* Search: Properties

Let $c = \frac{h^+ - h}{h^*}$ (relative error)

b^c effective branching factor

How many steps?

takes $O((b^c)^d)$ time

if good heuristics, may give dramatic improvements

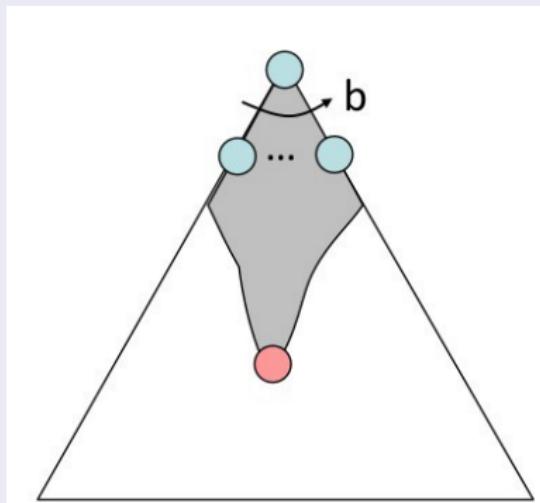
How much memory?

Keeps all nodes in memory
 $\Rightarrow O(b^d)$ memory size

- (like UCS)
 - Is it complete?

yes

- Is it optimal?



(Oc D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

Memory requirement is a major problem also for

A*

A* Search: Properties

Let $c = \frac{h^+ - h}{h^*}$ (relative error)

b^e effective branching factor

How many steps?

takes $O((b^c)^d)$ time

if good heuristics, may give dramatic improvements

How much memory?

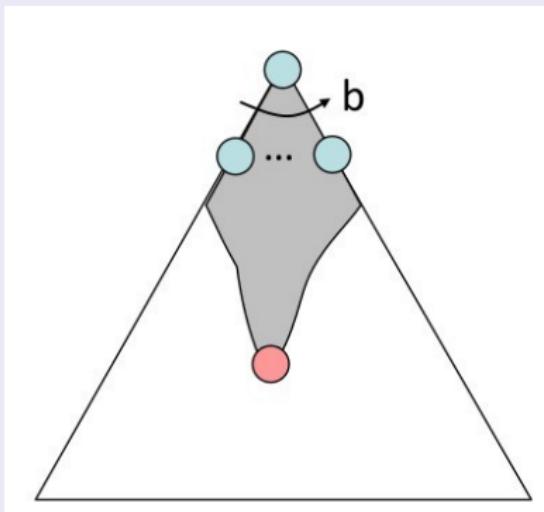
Keeps all nodes in memory
 $\Rightarrow O(b^d)$ memory size

(like UCS)

- Is it complete?

- yes

- Is it optimal?



(Oc D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

Memory requirement is a major problem also for

A*

A* Search: Properties

Let $c = \frac{h^+ - h}{h^*}$ (relative error)

b^c : effective branching factor

How many steps?

takes $O((b^c)^d)$ time

if good heuristics, may give dramatic improvements

How much memory?

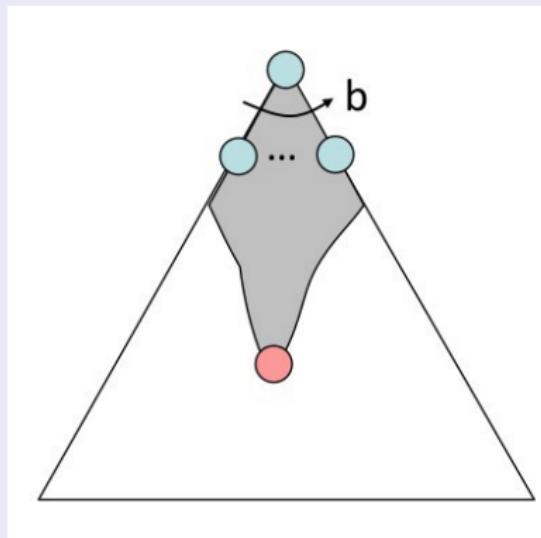
Keeps all nodes in memory
==> $O(b^d)$ memory size

(like UCS)

- Is it complete?

- yes

- Is it optimal?



(Oc D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

Memory requirement is a major problem also for

A*

Memory-bounded Heuristic Search (hints)

Some solutions to A^* space problems (maintain completeness and optimality)

- Iterative-deepening A^* (IDA*)
 - here cutoff information is the f-cost ($g+h$) instead of depth
- Recursive best-first search(RBFS)
 - attempts to mimic standard best-first search with linear space
- (simple) Memory-bounded A^* ((S)MA*)
 - drop the worst-leaf node when memory is full

Outline

1 Problem-Solving Agents

2 Example Problems

3 Search Generalities

4 Uninformed Search Strategies

- Breadth-First Search
- Uniform-cost Search

Depth-First Search

Depth-Limited Search &
Iterative Deepening

5 Informed Search Strategies

- Greedy Search

~~A* Search~~ Heuristic

Functions

Admissible Heuristics

Main problem

What is the best admissible/consistent heuristic?

Dominance of Admissible Heuristics

Dominance

Let $h_1(n)$, $h_2(n)$ admissible heuristics.

- $h_2(n)$ dominates $h_1(n)$ iff $h_2(n) \geq h_1(n)$ for all n.
==> $h_2(n)$ is better for search
 - is nearer to $h^*(n)$

Let $h_1(n)$, $h_2(n)$ admissible heuristics. Let $h_{12} = \max(h_1(n),$

- $h_2(n))$. h_{12} is also admissible

h_{12} dominates both $h_1(n)$, $h_2(n)$

Dominance of Admissible Heuristics

Dominance

Let $h_1(n)$, $h_2(n)$ admissible heuristics.

- $h_2(n)$ dominates $h_1(n)$ iff $h_2(n) \geq h_1(n)$ for all n.
==> $h_2(n)$ is better for search
 - is nearer to $h^*(n)$

Let $h_1(n)$, $h_2(n)$ admissible heuristics. Let $h_{12} = \max(h_1(n),$

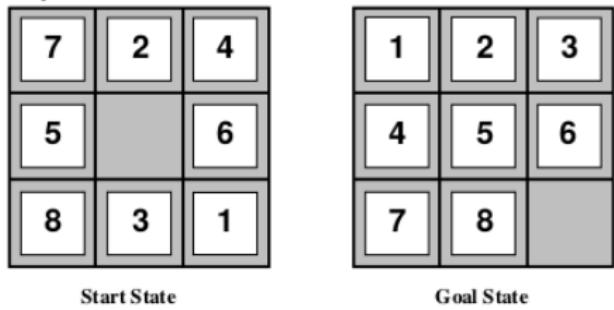
- $h_2(n))$. h_{12} is also admissible

h_{12} dominates both $h_1(n)$, $h_2(n)$

Admissible Heuristics: Example

Ex: Heuristics for the 8-puzzle

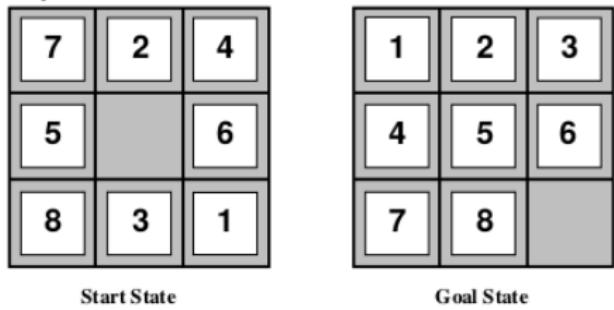
- $h_1(n)$: number of misplaced tiles
- $h_2(n)$: total Manhattan distance over all tiles
 - (i.e., # of squares from desired location of each tile)
- $h_1(S) \approx 6$
- $h_2(S) \approx 4+0+3+3+1+0+2+1 = 14$
- $h^*(S) \approx 26$
- both $h_1(n)$, $h_2(n)$ admissible (\leq number of actual steps to solve)
- $h_2(n)$ dominates $h_1(n)$



Admissible Heuristics: Example

Ex: Heuristics for the 8-puzzle

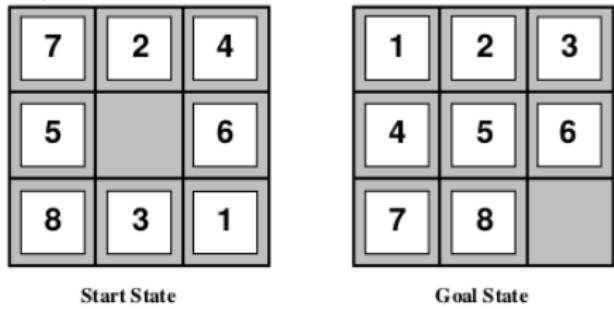
- $h_1(n)$: number of misplaced tiles
- $h_2(n)$: total Manhattan distance over all tiles
 - (i.e., # of squares from desired location of each tile)
- $h_1(S)$? 6
- $h_2(S)$? $4+0+3+3+1+0+2+1 = 14$
- $h^*(S)$? 26
- both $h_1(n)$, $h_2(n)$ admissible (\leq number of actual steps to solve)
- $h_2(n)$ dominates $h_1(n)$



Admissible Heuristics: Example

Ex: Heuristics for the 8-puzzle

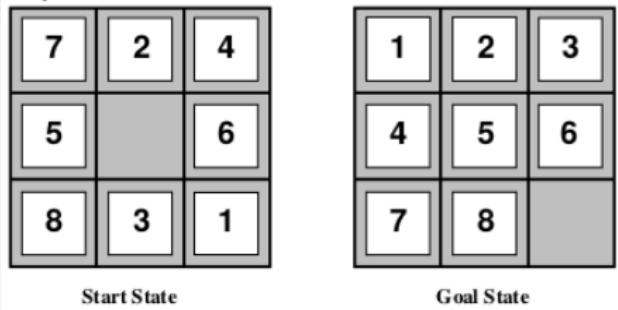
- $h_1(n)$: number of misplaced tiles
- $h_2(n)$: total Manhattan distance over all tiles
 - (i.e., # of squares from desired location of each tile)
- $h_1(S) \text{? } 6$
- $h_2(S) \text{? } 4+0+3+3+1+0+2+1 = 14$
- $h^*(S) \text{? } 26$
- both $h_1(n)$, $h_2(n)$ admissible (\leq number of actual steps to solve)
- $h_2(n)$ dominates $h_1(n)$



Admissible Heuristics: Example

Ex: Heuristics for the 8-puzzle

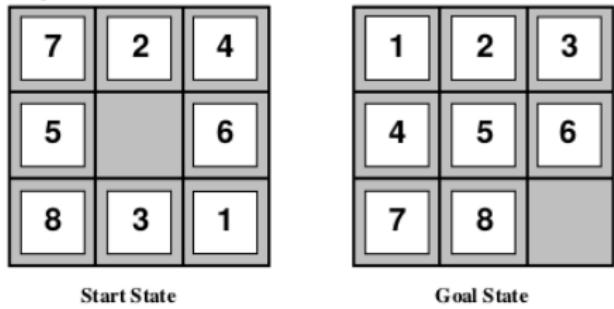
- $h_1(n)$: number of misplaced tiles
- $h_2(n)$: total Manhattan distance over all tiles
 - (i.e., # of squares from desired location of each tile)
- $h_1(S) = 6$
- $h_2(S) = 4+0+3+3+1+0+2+1 = 14$
- $h^*(S) = 26$
- both $h_1(n)$, $h_2(n)$ admissible (\leq number of actual steps to solve)
- $h_2(n)$ dominates $h_1(n)$



Admissible Heuristics: Example

Ex: Heuristics for the 8-puzzle

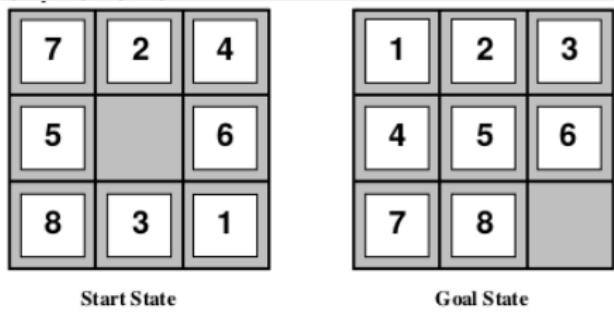
- $h_1(n)$: number of misplaced tiles
- $h_2(n)$: total Manhattan distance over all tiles
 - (i.e., # of squares from desired location of each tile)
- $h_1(S) \text{? } 6$
- $h_2(S) \text{? } 4+0+3+3+1+0+2+1 = 14$
- $h^*(S) \text{? } 26$
- both $h_1(n)$, $h_2(n)$ admissible (\leq number of actual steps to solve)
- $h_2(n)$ dominates $h_1(n)$



Admissible Heuristics: Example

Ex: Heuristics for the 8-puzzle

- $h_1(n)$: number of misplaced tiles
- $h_2(n)$: total Manhattan distance over all tiles
 - (i.e., # of squares from desired location of each tile)
- $h_1(S) \text{? } 6$
- $h_2(S) \text{? } 4+0+3+3+1+0+2+1 = 14$
- $h^*(S) \text{? } 26$
- both $h_1(n)$, $h_2(n)$ admissible (\leq number of actual steps to solve)
- $h_2(n)$ dominates $h_1(n)$

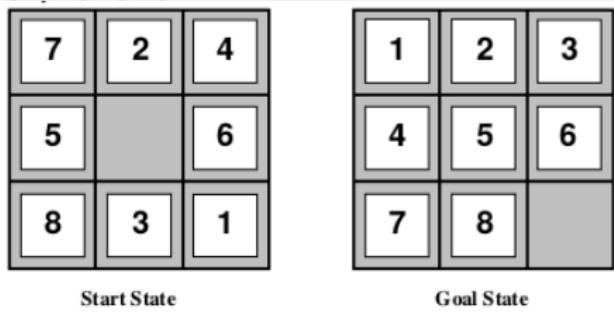


Admissible Heuristics: Example

Ex: Heuristics for the 8-puzzle

- $h_1(n)$: number of misplaced tiles
- $h_2(n)$: total Manhattan distance over all tiles
 - (i.e., # of squares from desired location of each tile)
- $h_1(S) \text{? } 6$
- $h_2(S) \text{? } 4+0+3+3+1+0+2+1 = 14$
- $h^*(S) \text{? } 26$
- both $h_1(n), h_2(n)$ admissible (\leq number of actual steps to solve)

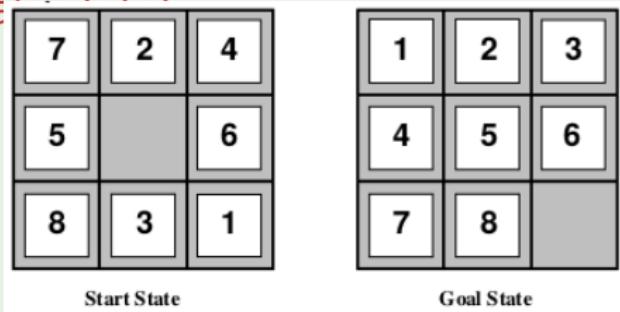
$h_2(n)$ dominates $h_1(n)$



Admissible Heuristics: Example

Ex: Heuristics for the 8-puzzle

- $h_1(n)$: number of misplaced tiles
- $h_2(n)$: total Manhattan distance over all tiles
 - (i.e., # of squares from desired location of each tile)
- $h_1(S) = 6$
- $h_2(S) = 4+0+3+3+1+0+2+1 = 14$
- $h^*(S) = 26$
- both $h_1(n)$, $h_2(n)$ admissible (\leq number of actual steps to solve)
- $h_2(n)$ dominates $h_1(n)$



Quality of Heuristics

Effective branching factor

- Effective branching factor b^* : the branching factor that a uniform tree of depth d would have in order to contain $N+1$ nodes

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

- N being the number of nodes generated by the A^* search ex: if $d=5$ and $N = 52$, then $b^* = 1.92$
 - experimental measure of b^* is fairly constant for hard problems
- ==> can provide a good guide to the heuristic's overall usefulness

Ideal value of b^* is 1

Admissible Heuristics: Example [cont.]

Average performances on 100 random samples of 8-puzzle

d	Search Cost (nodes generated)			Effective Branching Factor		
	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

(Oc S. Russell & P. Norwig, AIMA)

==> Dramatic performance improvement

Admissible Heuristics from Relaxed Problems

Idea: Admissible heuristics can be derived from the exact solution cost of a relaxed version of the problem

- Relaxed 8-puzzle: a tile can move from any tile to any other tile
 - ⇒ $h_1(n)$ gives the shortest solution
- Relaxed 8-puzzle: a tile can move to any adjacent square
 - ⇒ $h_2(n)$ gives the shortest solution

- - The relaxed problem adds edges to the state space
 - ⇒ any optimal solution in the original problem is also a solution in the relaxed problem
 - ⇒ the cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem

the derived heuristic is an exact cost for the relaxed problem

- ⇒ must obey the triangular inequality
- ⇒ consistent

Admissible Heuristics from Relaxed Problems

Idea: Admissible heuristics can be derived from the exact solution cost of a relaxed version of the problem

- Relaxed 8-puzzle: a tile can move from any tile to any other tile
- $\Rightarrow h_1(n)$ gives the shortest solution

Relaxed 8-puzzle: a tile can move to any adjacent square

$\Rightarrow h_2(n)$ gives the shortest solution



The relaxed problem adds edges to the state space

\Rightarrow any optimal solution in the original problem is also a solution in the relaxed problem

\Rightarrow the cost of an optimal solution to a relaxed problem is an

- admissible heuristic for the original problem

the derived heuristic is an exact cost for the relaxed problem

\Rightarrow must obey the triangular inequality

\Rightarrow consistent

Admissible Heuristics from Relaxed Problems

Idea: Admissible heuristics can be derived from the exact solution cost of a relaxed version of the problem

- Relaxed 8-puzzle: a tile can move from any tile to any other tile
- $\Rightarrow h_1(n)$ gives the shortest solution

Relaxed 8-puzzle: a tile can move to any adjacent square

$\Rightarrow h_2(n)$ gives the shortest solution



The relaxed problem adds edges to the state space

\Rightarrow any optimal solution in the original problem is also a solution in the relaxed problem

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

the derived heuristic is an exact cost for the relaxed problem

\Rightarrow must obey the triangular inequality

\Rightarrow consistent

Admissible Heuristics from Relaxed Problems

Idea: Admissible heuristics can be derived from the exact solution cost of a relaxed version of the problem

- Relaxed 8-puzzle: a tile can move from any tile to any other tile
 - $\Rightarrow h_1(n)$ gives the shortest solution

Relaxed 8-puzzle: a tile can move to any adjacent square

$\Rightarrow h_2(n)$ gives the shortest solution



The relaxed problem adds edges to the state space

\Rightarrow any optimal solution in the original problem is also a solution in the relaxed problem

\Rightarrow the cost of an optimal solution to a relaxed problem is an

- admissible heuristic for the original problem

the derived heuristic is an exact cost for the relaxed problem

\Rightarrow must obey the triangular inequality

\Rightarrow consistent

Admissible Heuristics from Relaxed Problems

Idea: Admissible heuristics can be derived from the exact solution cost of a relaxed version of the problem

- Relaxed 8-puzzle: a tile can move from any tile to any other tile
 - $\Rightarrow h_1(n)$ gives the shortest solution

Relaxed 8-puzzle: a tile can move to any adjacent square

$\Rightarrow h_2(n)$ gives the shortest solution



The relaxed problem adds edges to the state space

\Rightarrow any optimal solution in the original problem is also a solution in the relaxed problem

\Rightarrow the cost of an optimal solution to a relaxed problem is an

• admissible heuristic for the original problem

the derived heuristic is an exact cost for the relaxed problem

\Rightarrow must obey the triangular inequality

\Rightarrow consistent

Admissible Heuristics from Relaxed Problems

Idea: Admissible heuristics can be derived from the exact solution cost of a relaxed version of the problem

- Relaxed 8-puzzle: a tile can move from any tile to any other tile
 - $\Rightarrow h_1(n)$ gives the shortest solution
- Relaxed 8-puzzle: a tile can move to any adjacent square
 - $\Rightarrow h_2(n)$ gives the shortest solution



The relaxed problem adds edges to the state space

- \Rightarrow any optimal solution in the original problem is also a solution in the relaxed problem
- \Rightarrow the cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem

the derived heuristic is an exact cost for the relaxed problem

\Rightarrow must obey the triangular inequality

\Rightarrow consistent

Admissible Heuristics from Relaxed Problems

Idea: Admissible heuristics can be derived from the exact solution cost of a relaxed version of the problem

- Relaxed 8-puzzle: a tile can move from any tile to any other tile
 - $\Rightarrow h_1(n)$ gives the shortest solution
- Relaxed 8-puzzle: a tile can move to any adjacent square
 - $\Rightarrow h_2(n)$ gives the shortest solution



The relaxed problem adds edges to the state space

- \Rightarrow any optimal solution in the original problem is also a solution in the relaxed problem
- \Rightarrow the cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem

the derived heuristic is an exact cost for the relaxed problem

\Rightarrow must obey the triangular inequality

\Rightarrow consistent

Inferring Automatically Admissible Heuristics

Idea: If a problem definition is written down in a formal language, it is possible to construct relaxed problems automatically

Example

- 8-puzzle actions:
 - we can generate three relaxed problems by removing one or both of the conditions
 - (a) a tile can move from square A to square B if A is adjacent to B
 - (b) a tile can move from square A to square B if B is blank
 - (c) a tile can move from square A to square B
- (a) corresponds to $h_2(n)$, (c) corresponds to $h_1(n)$,

Inferring Automatically Admissible Heuristics

Idea: If a problem definition is written down in a formal language, it is possible to construct relaxed problems automatically

Example

- 8-puzzle actions:
 - a tile can move from square A to square B if
 - A is horizontally or vertically adjacent to B, and
 - B is blank
 - we can generate three relaxed problems by removing one or both of the conditions
 - (a) a tile can move from square A to square B if A is adjacent to B
 - (b) a tile can move from square A to square B if B is blank
 - (c) a tile can move from square A to square B
- ⇒ (a) corresponds to $h_2(n)$, (c) corresponds to $h_1(n)$,

Inferring Automatically Admissible Heuristics

Idea: If a problem definition is written down in a formal language, it is possible to construct relaxed problems automatically

Example

- 8-puzzle actions:
 - a tile can move from square A to square B if
 - A is horizontally or vertically adjacent to B, and
 - B is blank
- we can generate three relaxed problems by removing one or both of the conditions
 - (a) a tile can move from square A to square B if A is adjacent to B
 - (b) a tile can move from square A to square B if B is blank
 - (c) a tile can move from square A to square B

→ (a) corresponds to $h_2(n)$, (c) corresponds to $h_1(n)$,

Inferring Automatically Admissible Heuristics

Idea: If a problem definition is written down in a formal language, it is possible to construct relaxed problems automatically

Example

- 8-puzzle actions:
 - a tile can move from square A to square B if
 - A is horizontally or vertically adjacent to B, and
 - B is blank
 - we can generate three relaxed problems by removing one or both of the conditions
 - (a) a tile can move from square A to square B if A is adjacent to B
 - (b) a tile can move from square A to square B if B is blank
 - (c) a tile can move from square A to square B
- ==> (a) corresponds to $h_2(n)$, (c) corresponds to $h_1(n)$,

Inferring Automatically Admissible Heuristics

Idea: If a problem definition is written down in a formal language, it is possible to construct relaxed problems automatically

Example

- 8-puzzle actions:
 - a tile can move from square A to square B if
 - A is horizontally or vertically adjacent to B, and
 - B is blank
 - we can generate three relaxed problems by removing one or both of the conditions
 - (a) a tile can move from square A to square B if A is adjacent to B
 - (b) a tile can move from square A to square B if B is blank
 - (c) a tile can move from square A to square B
- ==> (a) corresponds to $h_2(n)$, (c) corresponds to $h_1(n)$,

Learning Admissible Heuristics

- Another way to find an admissible heuristic is through **learning from experience**:
 - Experience = solving lots of 8-puzzles
 - An **inductive learning** algorithm can be used to predict costs for other states that arise during search