Lecture *Hierarchical Planning*

Chapter:
Search

Dr. Pascal Bercher

Institute of Artificial Intelligence,
Ulm University, Germany

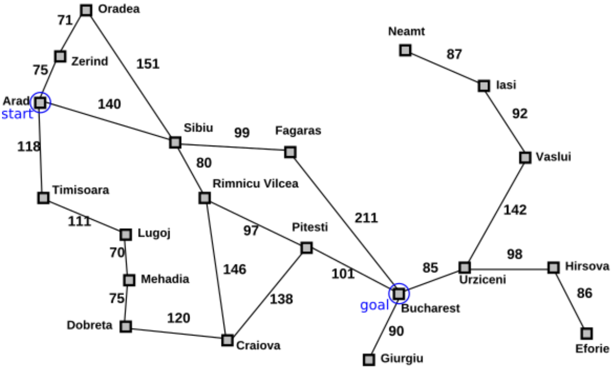
Winter Term 2018/2019

(Compiled on: February 19, 2019)

ulm university universität uulm

Overview:

Example



copyright: see slide 46[1] (modified)

How to – automatically – find a(n optimal/good) way from *Arad* to *Bucharest?*

(The edge numbers indicate action/traversal costs.)

Introduction

Search is a systematic way to solve a certain class of problems.

Introduction

Search is a systematic way to solve a certain class of problems.

How do these problems look like?

Introduction

Search is a systematic way to solve a certain class of problems.

How do these problems look like?

- They have an initial state, one or more goal states, and a set of actions with costs.

Introduction

Search is a systematic way to solve a certain class of problems.

How do these problems look like?

- They have an initial state, one or more goal states, and a set of actions with costs.
- More formally, search problems are defined upon *transition systems*.

Introduction

Search is a systematic way to solve a certain class of problems.

How do these problems look like?

- They have an initial state, one or more goal states, and a set of actions with costs.
- More formally, search problems are defined upon *transition systems*.
- Note:
  - The transition system's state does not necessarily coincide with a state known from planning! Also possible, e.g., partial plan.

Introduction

Search is a systematic way to solve a certain class of problems.

How do these problems look like?

- They have an initial state, one or more goal states, and a set of actions with costs.
- More formally, search problems are defined upon *transition systems*.
- Note:
  - The transition system's state does not necessarily coincide with a state known from planning! Also possible, e.g., partial plan.
  - Also, don't confuse the transition system's state with a search node. Search nodes can contain more information, like the path (sequence of transitions) discovered to reach the respective node.

Introduction

Search is a systematic way to solve a certain class of problems.

How do these problems look like?

- They have an initial state, one or more goal states, and a set of actions with costs.
- More formally, search problems are defined upon *transition systems*.
- Note:
  - The transition system's state does not necessarily coincide with a state known from planning! Also possible, e.g., partial plan.
  - Also, don't confuse the transition system's state with a search node. Search nodes can contain more information, like the path (sequence of transitions) discovered to reach the respective node.
  - Depending on which problem to solve, we might need to rely upon infinite state transition systems (e.g., for POCL and for hierarchical problems).

Introduction, cont'd

What are solutions to search problems?

Introduction, cont'd

What are solutions to search problems?

- That depends on the definition of the transition system. Normally, a solution to a search problem is a sequence of transitions from the transition system's initial state to a goal state. Its cost is the sum of the actions' costs.

Introduction, cont'd

What are solutions to search problems?

- That depends on the definition of the transition system. Normally, a solution to a search problem is a sequence of transitions from the transition system's initial state to a goal state. Its cost is the sum of the actions' costs.
- A solution is called optimal if

Introduction, cont'd

What are solutions to search problems?

- That depends on the definition of the transition system. Normally, a solution to a search problem is a sequence of transitions from the transition system's initial state to a goal state. Its cost is the sum of the actions' costs.
- A solution is called optimal if
  - it is the cheapest one, or

Introduction, cont'd

What are solutions to search problems?

- That depends on the definition of the transition system. Normally, a solution to a search problem is a sequence of transitions from the transition system's initial state to a goal state. Its cost is the sum of the actions' costs.
- A solution is called optimal if
  - it is the cheapest one, or
  - if it is the shortest one (if there are no action costs).

More examples

- Pathfinding (cf. first slide)

More examples

- Pathfinding (cf. first slide)
- All the ones from the last lecture!

More examples

- Pathfinding (cf. first slide)
- All the ones from the last lecture!
- Anything that can be described with a transition system, e.g., *Cannibals and Missionaries*, ...

How to Search?

Informally, *search* means:

- Maintain a so-called *search fringe* (also called *frontier* or *open list*) – a set of candidate search nodes (containing the respective state and further information).

fringe:

How to Search?

Informally, *search* means:

- Maintain a so-called *search fringe* (also called *frontier* or *open list*) – a set of candidate search nodes (containing the respective state and further information).
- Initially, that fringe contains just the initial state.

fringe: $\{(\mathit{Arad}\,;\ \text{cost:0, etc.})\}$

How to Search?

Informally, *search* means:

- Maintain a so-called *search fringe* (also called *frontier* or *open list*) – a set of candidate search nodes (containing the respective state and further information).
- Initially, that fringe contains just the initial state.
- Select and remove a "most promising" node from the fringe:

fringe: {(*Arad* ; cost:0 etc.)}

selected: —

How to Search?

Informally, *search* means:

- Maintain a so-called *search fringe* (also called *frontier* or *open list*) – a set of candidate search nodes (containing the respective state and further information).
- Initially, that fringe contains just the initial state.
- Select and remove a “most promising” node from the fringe:

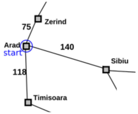
fringe: $\emptyset$

selected: (*Arad* ; cost:0 etc.)

How to Search?

Informally, *search* means:

- Maintain a so-called *search fringe* (also called *frontier* or *open list*) – a set of candidate search nodes (containing the respective state and further information).
- Initially, that fringe contains just the initial state.
- Select and remove a “most promising” node from the fringe:
  - If it’s a solution, extract a transition from the initial state/search node to it.
  - If it’s not, put all successor nodes into the fringe.



copyright: see slide 46[1] (modified)

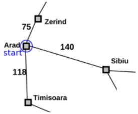
fringe: $\emptyset$

selected: $($*Arad* ; cost:0 etc. $)$

How to Search?

Informally, *search* means:

- Maintain a so-called *search fringe* (also called *frontier* or *open list*) – a set of candidate search nodes (containing the respective state and further information).
- Initially, that fringe contains just the initial state.
- Select and remove a "most promising" node from the fringe:
  - If it's a solution, extract a transition from the initial state/search node to it.
  - If it's not, put all successor nodes into the fringe.



copyright: see slide 46[1] (modified)

fringe: {(*Zerind* ; cost:75 etc.), (*Sibiu* ; cost:140 etc.), (*Timisoara* ; cost:118 etc.)}

selected: —

How to Search?

Informally, *search* means:

- Maintain a so-called *search fringe* (also called *frontier* or *open list*) – a set of candidate search nodes (containing the respective state and further information).
- Initially, that fringe contains just the initial state.
- Select and remove a “most promising” node from the fringe:
  - If it’s a solution, extract a transition from the initial state/search node to it.
  - If it’s not, put all successor nodes into the fringe.

(Some of the) open questions:

How to Search?

Informally, *search* means:

- Maintain a so-called *search fringe* (also called *frontier* or *open list*) – a set of candidate search nodes (containing the respective state and further information).
- Initially, that fringe contains just the initial state.
- Select and remove a “most promising” node from the fringe:
  - If it’s a solution, extract a transition from the initial state/search node to it.
  - If it’s not, put all successor nodes into the fringe.

(Some of the) open questions:

- Which node is the “most promising” search node?

How to Search?

Informally, *search* means:

- Maintain a so-called *search fringe* (also called *frontier* or *open list*) – a set of candidate search nodes (containing the respective state and further information).
- Initially, that fringe contains just the initial state.
- Select and remove a “most promising” node from the fringe:
  - If it’s a solution, extract a transition from the initial state/search node to it.
  - If it’s not, put all successor nodes into the fringe.

(Some of the) open questions:

- Which node is the “most promising” search node?
- Which nodes to put back to the fringe?

How to Search?

Informally, *search* means:

- Maintain a so-called *search fringe* (also called *frontier* or *open list*) – a set of candidate search nodes (containing the respective state and further information).
- Initially, that fringe contains just the initial state.
- Select and remove a “most promising” node from the fringe:
  - If it’s a solution, extract a transition from the initial state/search node to it.
  - If it’s not, put all successor nodes into the fringe.

(Some of the) open questions:

- Which node is the “most promising” search node?
- Which nodes to put back to the fringe?
  - What about states that were already visited?

How to Search?

Informally, *search* means:

- Maintain a so-called *search fringe* (also called *frontier* or *open list*) – a set of candidate search nodes (containing the respective state and further information).
- Initially, that fringe contains just the initial state.
- Select and remove a "most promising" node from the fringe:
  - If it's a solution, extract a transition from the initial state/search node to it.
  - If it's not, put all successor nodes into the fringe.

(Some of the) open questions:

- Which node is the "most promising" search node?
- Which nodes to put back to the fringe?
  - What about states that were already visited?
  - What about states that are solutions?

Node Selections

Which are the most promising search nodes?

Node Selections

Which are the most promising search nodes?

- Base this estimate on well-informed heuristics $\rightarrow$ *informed search*

Node Selections

Which are the most promising search nodes?

- Base this estimate on well-informed heuristics $\rightarrow$ *informed search*
  - Requires heuristics, but

Node Selections

Which are the most promising search nodes?

- Base this estimate on well-informed heuristics $\rightarrow$ *informed search*
  - Requires heuristics, but
  - potentially, they are *much* more efficient.

Node Selections

Which are the most promising search nodes?

- Base this estimate on well-informed heuristics → *informed search*
  - Requires heuristics, but
  - potentially, they are *much* more efficient.
- Do not use any information about the goal → *uninformed, blind search*

Node Selections

Which are the most promising search nodes?

- Base this estimate on well-informed heuristics → *informed search*
  - Requires heuristics, but
  - potentially, they are *much* more efficient.
- Do not use any information about the goal → *uninformed, blind search*
  - Works also if no heuristics are known.

Node Selections

Which are the most promising search nodes?

- Base this estimate on well-informed heuristics $\rightarrow$ *informed search*
  - Requires heuristics, but
  - potentially, they are *much* more efficient.
- Do not use any information about the goal $\rightarrow$ *uninformed, blind search*
  - Works also if no heuristics are known.
  - Often very inefficient compared to informed search.

Node Insertions

How to deal with states that were visited before?

Node Insertions

How to deal with states that were visited before?

- Just visit them again $\rightarrow$ *tree search*

Node Insertions

How to deal with states that were visited before?

- Just visit them again $\rightarrow$ *tree search*
  - Potentially less efficient (e.g., imagine a cyclic transition system *without* a reachable solution)

Node Insertions

How to deal with states that were visited before?

- Just visit them again $\rightarrow$ *tree search*
  - Potentially less efficient (e.g., imagine a cyclic transition system *without* a reachable solution)
  - It is easier to obtain optimality guarantees

Node Insertions

How to deal with states that were visited before?

- Just visit them again → *tree search*
  - Potentially less efficient (e.g., imagine a cyclic transition system *without* a reachable solution)
  - It is easier to obtain optimality guarantees
- Ignore duplicates → *graph search*

Node Insertions

How to deal with states that were visited before?

- Just visit them again $\rightarrow$ *tree search*
  - Potentially less efficient (e.g., imagine a cyclic transition system *without* a reachable solution)
  - It is easier to obtain optimality guarantees
- Ignore duplicates $\rightarrow$ *graph search*
  - Requires a *visited list* (also called *closed list* or *explored set*).

Node Insertions

How to deal with states that were visited before?

- Just visit them again → *tree search*
  - Potentially less efficient (e.g., imagine a cyclic transition system *without* a reachable solution)
  - It is easier to obtain optimality guarantees
- Ignore duplicates → *graph search*
  - Requires a *visited list* (also called *closed list* or *explored set*).
  - Potentially more efficient (but may also require more space due to storing that set).

Node Insertions

How to deal with states that were visited before?

- Just visit them again $\rightarrow$ *tree search*
  - Potentially less efficient (e.g., imagine a cyclic transition system *without* a reachable solution)
  - It is easier to obtain optimality guarantees
- Ignore duplicates $\rightarrow$ *graph search*
  - Requires a *visited list* (also called *closed list* or *explored set*).
  - Potentially more efficient (but may also require more space due to storing that set).
  - We need to be more careful if we want to guarantee optimal solutions.

Tree Search vs. Graph Search

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

---

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
    if not in the explored set
      add the node to the explored set
      expand the chosen node, adding the resulting nodes to the frontier

An informal description of the general tree-search and graph-search algorithms. The parts of GRAPH-SEARCH marked in bold italic are the additions needed to handle repeated states.

Basic Definitions and Properties

The following numbers are used to study time and space requirements of search algorithms:

- *b* Maximal branching factor of the transition system.

Basic Definitions and Properties

The following numbers are used to study time and space requirements of search algorithms:

- b Maximal branching factor of the transition system.
- d Goal depth, i.e., shortest path of transitions from initial state to a goal state.

Basic Definitions and Properties

The following numbers are used to study time and space requirements of search algorithms:

- *b* Maximal branching factor of the transition system.
- *d* Goal depth, i.e., shortest path of transitions from initial state to a goal state.
- *m* The actually deployed search depth. (Note: This is a property of the *search process*, not (directly) of the *transition system*.)

Basic Definitions and Properties

The following numbers are used to study time and space requirements of search algorithms:

- b Maximal branching factor of the transition system.
- d Goal depth, i.e., shortest path of transitions from initial state to a goal state.
- m The actually deployed search depth. (Note: This is a property of the *search process*, not (directly) of the *transition system*.)
- $h(n)$ Heuristic value of the search node's state (more later).

Basic Definitions and Properties

The following numbers are used to study time and space requirements of search algorithms:

- b Maximal branching factor of the transition system.
- d Goal depth, i.e., shortest path of transitions from initial state to a goal state.
- m The actually deployed search depth. (Note: This is a property of the *search process*, not (directly) of the *transition system*.)
- $h(n)$ Heuristic value of the search node's state (more later).
- $g(n)$ Cost spent so far during search to reach a search node n.

Basic Definitions and Properties

The following numbers are used to study time and space requirements of search algorithms:

- b Maximal branching factor of the transition system.
- d Goal depth, i.e., shortest path of transitions from initial state to a goal state.
- m The actually deployed search depth. (Note: This is a property of the *search process*, not (directly) of the *transition system*.)
- $h(n)$ Heuristic value of the search node's state (more later).
- $g(n)$ Cost spent so far during search to reach a search node n.
- g^* The cost of an optimal solution.

Basic Definitions and Properties, cont'd

The following are elemental properties of search algorithms:

Optimality A search algorithm is called *optimal* when it is guaranteed to find a cost-optimal/shortest solution (if one exists).

Basic Definitions and Properties, cont'd

The following are elemental properties of search algorithms:

Optimality A search algorithm is called *optimal* when it is guaranteed to find a cost-optimal/shortest solution (if one exists).

Completeness There are different notions of completeness. We call a search algorithm *complete* if:

Basic Definitions and Properties, cont'd

The following are elemental properties of search algorithms:

Optimality A search algorithm is called *optimal* when it is guaranteed to find a cost-optimal/shortest solution (if one exists).

Completeness There are different notions of completeness. We call a search algorithm *complete* if:

- If there is a solution, it finds one,

Basic Definitions and Properties, cont'd

The following are elemental properties of search algorithms:

Optimality A search algorithm is called *optimal* when it is guaranteed to find a cost-optimal/shortest solution (if one exists).

Completeness There are different notions of completeness. We call a search algorithm *complete* if:

- If there is a solution, it finds one,
- if there is a solution, an optimal one can be found, or

Basic Definitions and Properties, cont'd

The following are elemental properties of search algorithms:

Optimality A search algorithm is called *optimal* when it is guaranteed to find a cost-optimal/shortest solution (if one exists).

Completeness There are different notions of completeness. We call a search algorithm *complete* if:

- If there is a solution, it finds one,
- if there is a solution, an optimal one can be found, or
- all solutions can be found.

These are differently strong notions; all of them can be found in the literature. Be aware which one applies.

Basic Definitions and Properties, cont'd

The following are elemental properties of search algorithms:

Optimality A search algorithm is called *optimal* when it is guaranteed to find a cost-optimal/shortest solution (if one exists).

Completeness There are different notions of completeness. We call a search algorithm *complete* if:

- If there is a solution, it finds one,
- if there is a solution, an optimal one can be found, or
- all solutions can be found.

These are differently strong notions; all of them can be found in the literature. Be aware which one applies.

Correctness If a solution is returned, it is in fact a solution and if "no solution exists" is returned, there does in fact not exist a solution.

BFS Algorithm

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

Breadth-first search on a graph.

Question:

In which way does this algorithm differ (e.g., is more precise) than the generic graph-search algorithm?

BFS Algorithm

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
    if not in the explored set
      add the node to the explored set
      expand the chosen node, adding the resulting nodes to the frontier

An informal description of the general graph-search algorithm. The parts of GRAPH-SEARCH marked in bold italic are the additions needed to handle repeated states.

Question:

In which way does this algorithm differ (e.g., is more precise) than the generic graph-search algorithm?

BFS Algorithm

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

Breadth-first search on a graph.

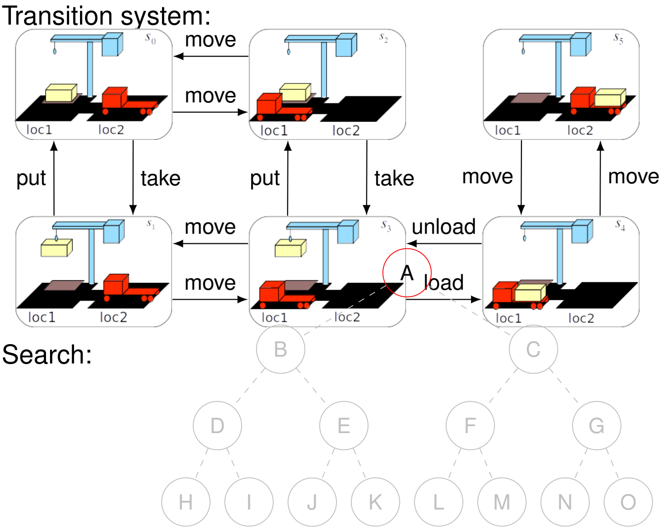
copyright: see slide 46[1] (modified)

Question:

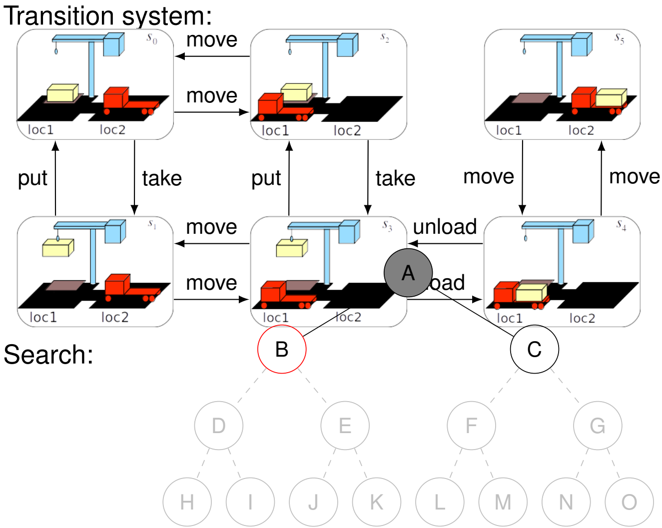
In which way does this algorithm differ (e.g., is more precise) than the generic graph-search algorithm?

- Here, the goal test is done *before insertion into the fringe* ("early goal test"), not after selection from the fringe.
- The fringe is not "generic", but implemented as FIFO.

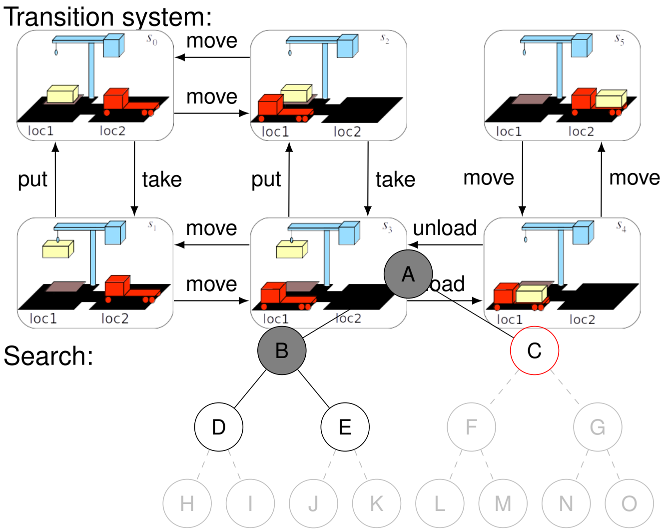
Example of BFS



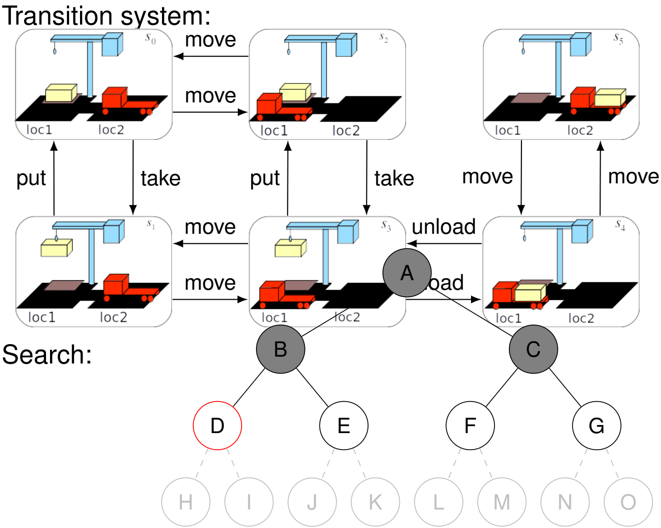
Example of BFS



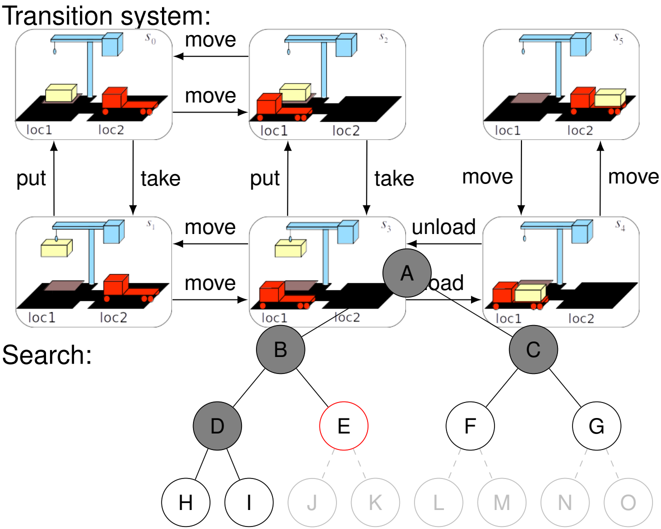
Example of BFS



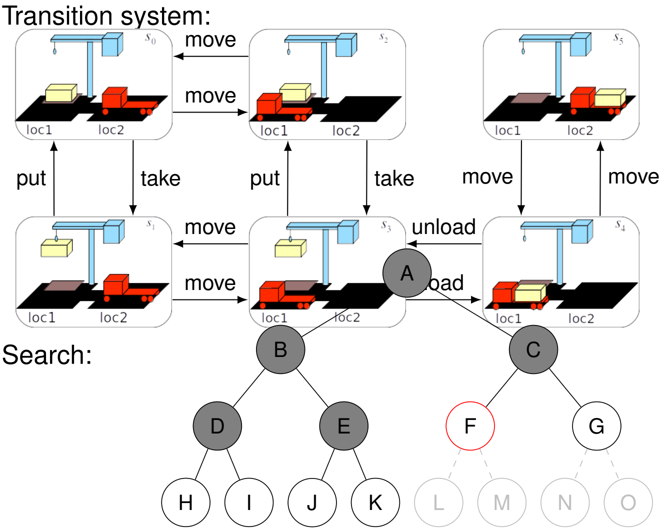
Example of BFS



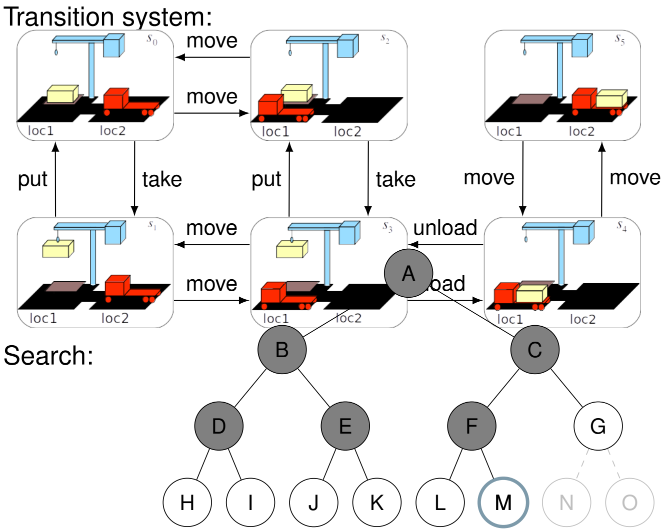
Example of BFS



Example of BFS

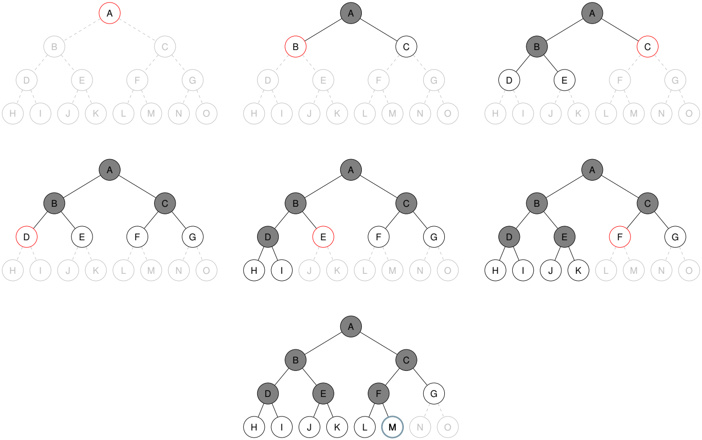


Example of BFS



Example of BFS, cont'd

At one glance:



Properties of BFS

Optimality

Properties of BFS

Optimality Depends on the costs:

- Without action costs or with unit costs:

Properties of BFS

Optimality Depends on the costs:

- Without action costs or with unit costs:
  - Obviously (both for tree- and for graph search)
  - Even optimal with an "early goal test"

Properties of BFS

Optimality Depends on the costs:

- Without action costs or with unit costs:
  - Obviously (both for tree- and for graph search)
  - Even optimal with an “early goal test”
- With action costs: No

Properties of BFS

Optimality Depends on the costs:

- Without action costs or with unit costs:
  - Obviously (both for tree- and for graph search)
  - Even optimal with an "early goal test"
- With action costs: No

Completeness

Properties of BFS

Optimality Depends on the costs:

- Without action costs or with unit costs:
  - Obviously (both for tree- and for graph search)
  - Even optimal with an "early goal test"
- With action costs: No

Completeness Depends on properties of the transition system:

Properties of BFS

Optimality Depends on the costs:

- Without action costs or with unit costs:
  - Obviously (both for tree- and for graph search)
  - Even optimal with an "early goal test"
- With action costs: No

Completeness Depends on properties of the transition system:

- Finite: Yes

Properties of BFS

Optimality Depends on the costs:

- Without action costs or with unit costs:
  - Obviously (both for tree- and for graph search)
  - Even optimal with an “early goal test”
- With action costs: No

Completeness Depends on properties of the transition system:

- Finite: Yes
- Infinite: Only with finite branching factor.

Properties of BFS

Optimality Depends on the costs:

- Without action costs or with unit costs:
  - Obviously (both for tree- and for graph search)
  - Even optimal with an "early goal test"
- With action costs: No

Completeness Depends on properties of the transition system:

- Finite: Yes
- Infinite: Only with finite branching factor.

Correctness Yes

Properties of BFS

Optimality Depends on the costs:

- Without action costs or with unit costs:
  - Obviously (both for tree- and for graph search)
  - Even optimal with an “early goal test”
- With action costs: No

Completeness Depends on properties of the transition system:

- Finite: Yes
- Infinite: Only with finite branching factor.

Correctness Yes

Space

Properties of BFS

Optimality Depends on the costs:

- Without action costs or with unit costs:
  - Obviously (both for tree- and for graph search)
  - Even optimal with an “early goal test”
- With action costs: No

Completeness Depends on properties of the transition system:

- Finite: Yes
- Infinite: Only with finite branching factor.

Correctness Yes

Space $O(b^d)$

Properties of BFS

Optimality Depends on the costs:

- Without action costs or with unit costs:
  - Obviously (both for tree- and for graph search)
  - Even optimal with an "early goal test"
- With action costs: No

Completeness Depends on properties of the transition system:

- Finite: Yes
- Infinite: Only with finite branching factor.

Correctness Yes

Space $O(b^d)$

Time

Properties of BFS

Optimality Depends on the costs:

- Without action costs or with unit costs:
  - Obviously (both for tree- and for graph search)
  - Even optimal with an "early goal test"
- With action costs: No

Completeness Depends on properties of the transition system:

- Finite: Yes
- Infinite: Only with finite branching factor.

Correctness Yes

Space $O(b^d)$

Time Just as space. (If the fringe is implemented as a priority queue rather than as queue (FIFO), the queue sorting overhead needs to be added. This does not change the asymptotic runtime, however.)

DFS Algorithm

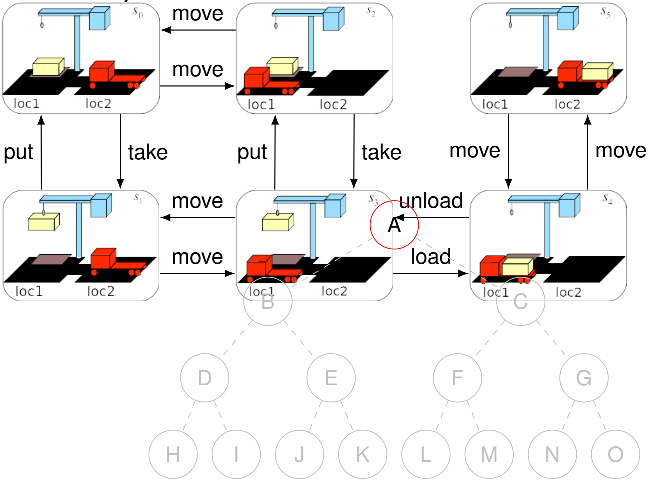
```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
     if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

Breadth-first search on a graph.

Just replace the FIFO fringe by a LIFO fringe.

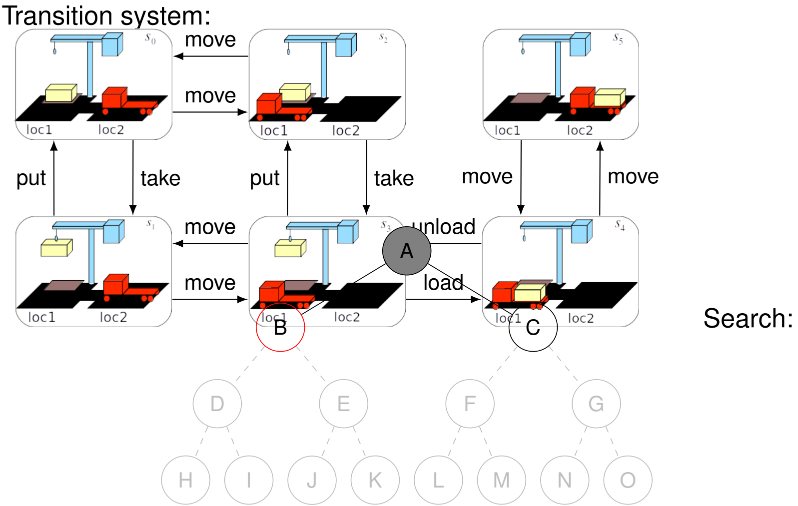
Example of DFS

Transition system:



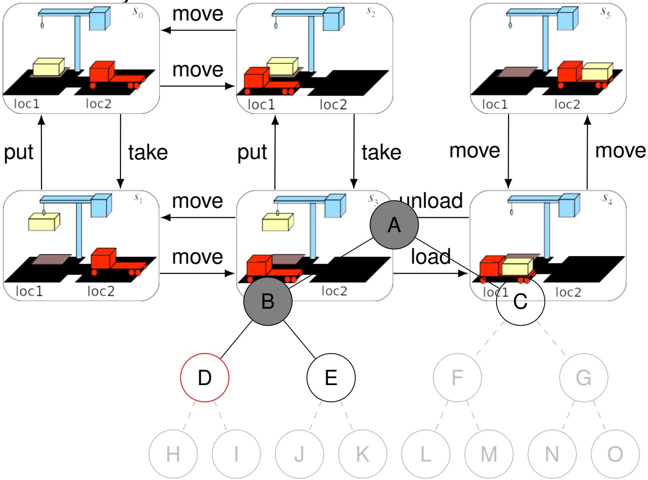
Search:

Example of DFS



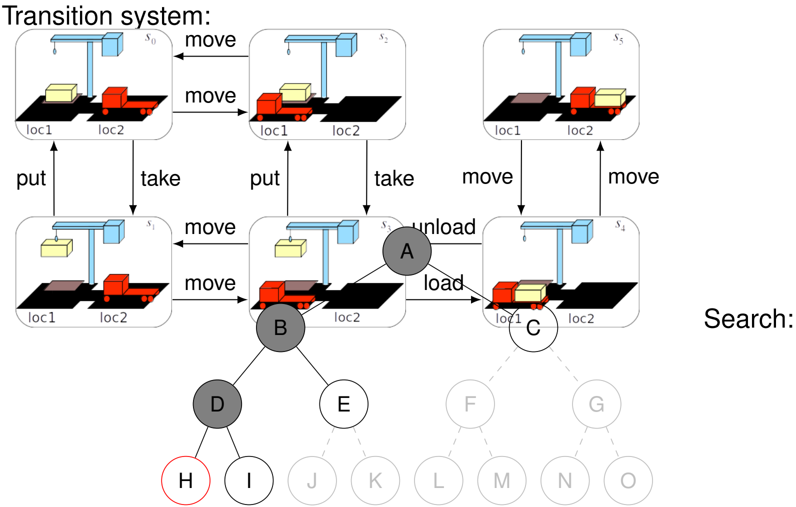
Example of DFS

Transition system:

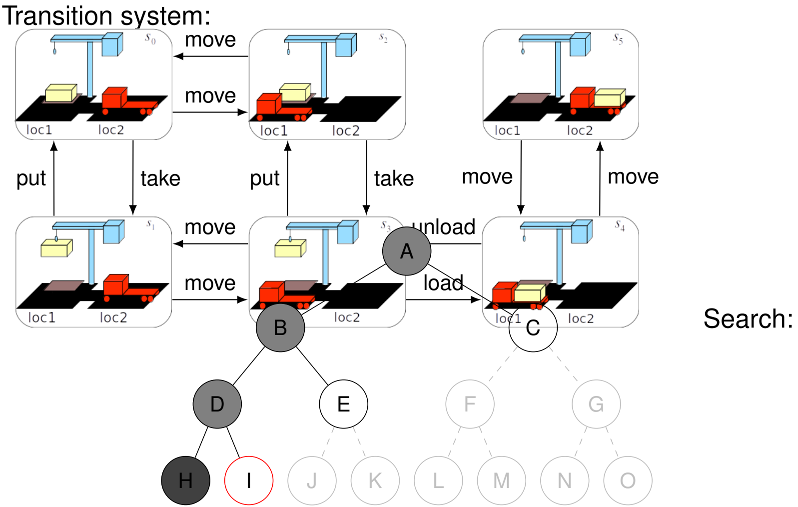


Search:

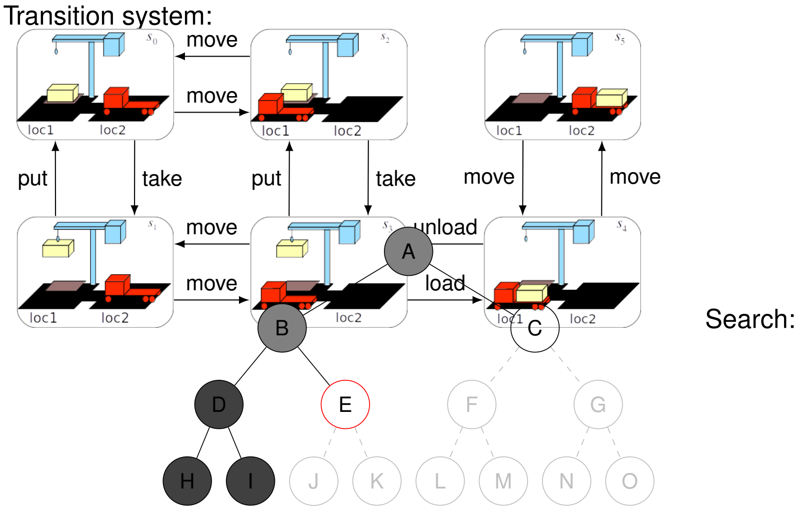
Example of DFS



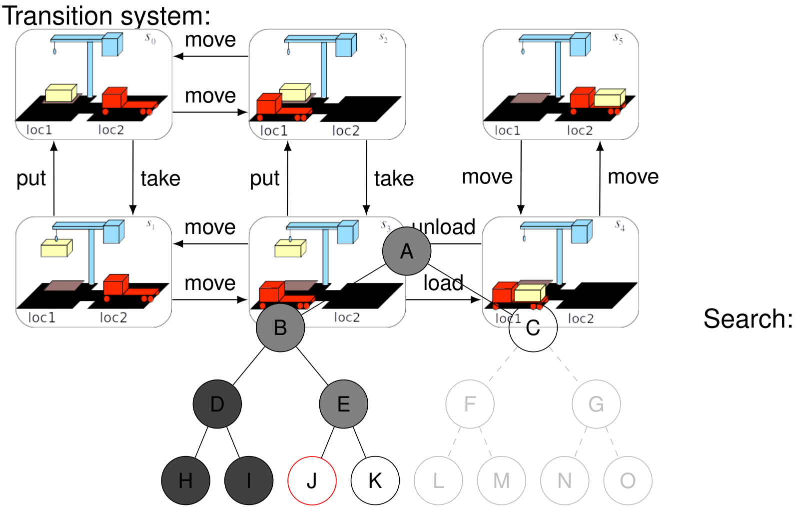
Example of DFS



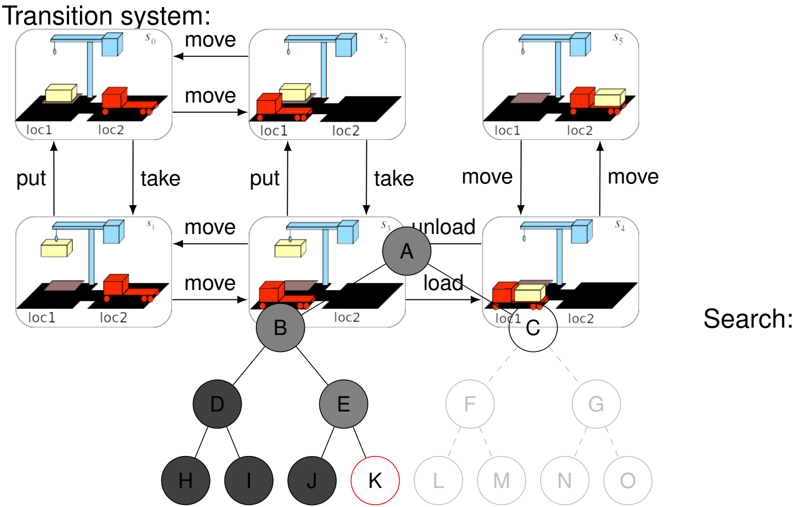
Example of DFS



Example of DFS

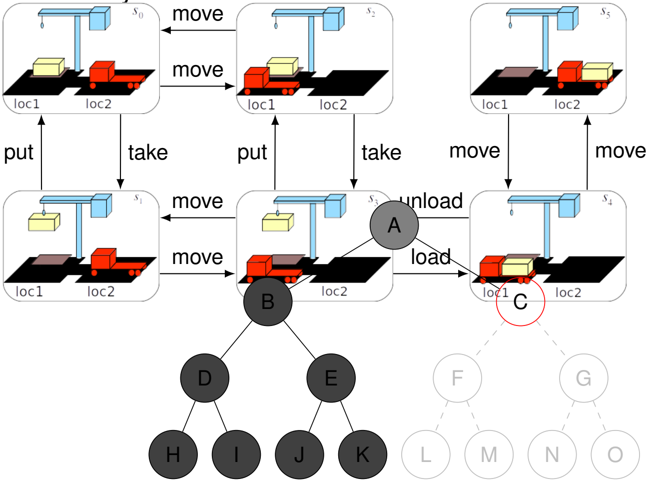


Example of DFS



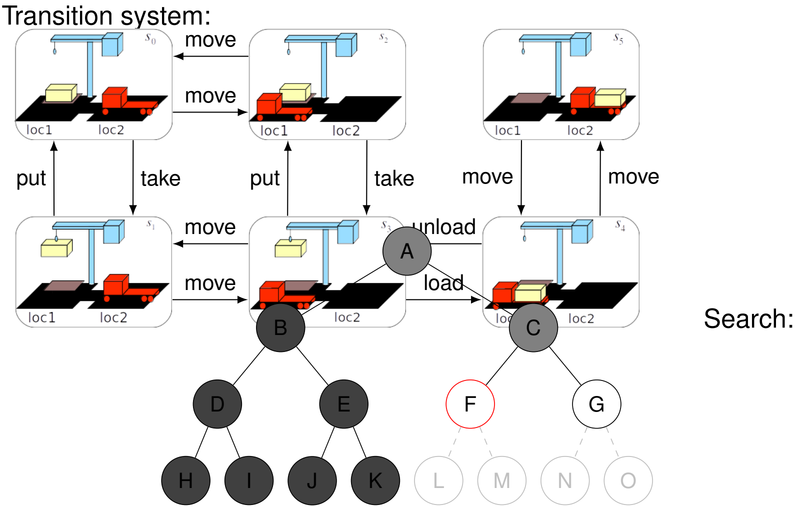
Example of DFS

Transition system:

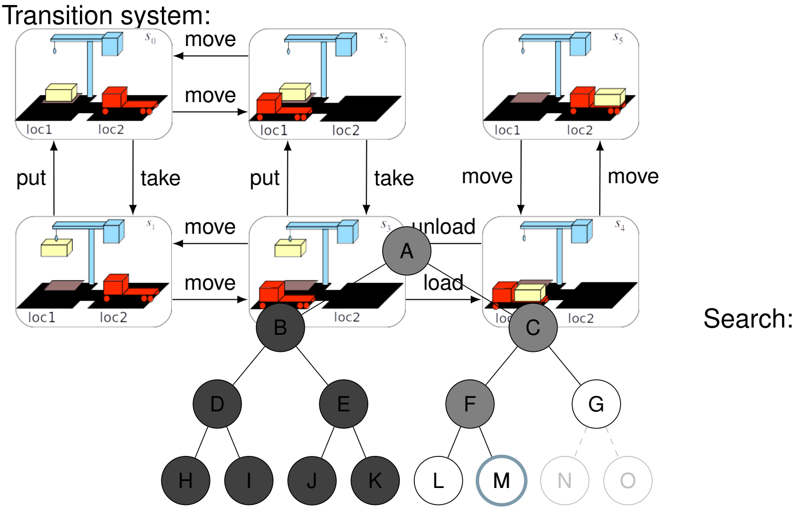


Search:

Example of DFS

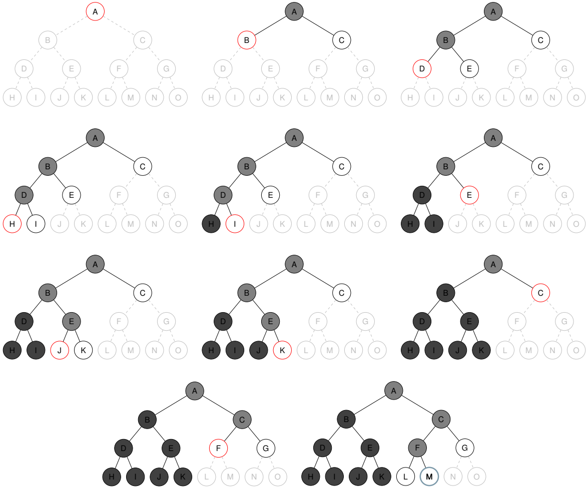


Example of DFS



Example of DFS, cont'd

At one glance:



Properties of DFS

Optimality

Properties of DFS

Optimality No

Properties of DFS

Optimality No

Completeness

Properties of DFS

Optimality No

Completeness Depends on duplicate management:

Properties of DFS

Optimality No

Completeness Depends on duplicate management:

- Tree search: only if the transition system is acyclic

Properties of DFS

Optimality No

Completeness Depends on duplicate management:

- Tree search: only if the transition system is acyclic
- Graph search: only the weakest form of completeness (and this only if the transition system is acyclic)

Properties of DFS

Optimality No

Completeness Depends on duplicate management:

- Tree search: only if the transition system is acyclic
- Graph search: only the weakest form of completeness (and this only if the transition system is acyclic)

Correctness Yes

Properties of DFS

Optimality No

Completeness Depends on duplicate management:

- Tree search: only if the transition system is acyclic
- Graph search: only the weakest form of completeness (and this only if the transition system is acyclic)

Correctness Yes

Space

Properties of DFS

Optimality No

Completeness Depends on duplicate management:

- Tree search: only if the transition system is acyclic
- Graph search: only the weakest form of completeness (and this only if the transition system is acyclic)

Correctness Yes

Space $O(b \cdot m)$ (If you only store the fringe.)

Properties of DFS

Optimality No

Completeness Depends on duplicate management:

- Tree search: only if the transition system is acyclic
- Graph search: only the weakest form of completeness (and this only if the transition system is acyclic)

Correctness Yes

Space $O(b \cdot m)$ (If you only store the fringe.)

Time

Properties of DFS

Optimality No

Completeness Depends on duplicate management:

- Tree search: only if the transition system is acyclic
- Graph search: only the weakest form of completeness (and this only if the transition system is acyclic)

Correctness Yes

Space $O(b \cdot m)$ (If you only store the fringe.)

Time $O(b^m)$ (If the fringe is implemented as a priority queue rather than as stack (LIFO), the queue sorting overhead needs to be added. This does not change the asymptotic runtime, however.)

Algorithm and Remarks

- Implements the generic tree-search or graph-search algorithms.

Algorithm and Remarks

- Implements the generic tree-search or graph-search algorithms.
- Implements fringe as priority queue that selects a node with minimal cost value $g(n)$.

Algorithm and Remarks

- Implements the generic tree-search or graph-search algorithms.
- Implements fringe as priority queue that selects a node with minimal cost value $g(n)$.
- UCS can be a regarded a modification of BFS by expanding the cheapest rather than the shallowest node. *Note:* In contrast to BFS, the early goal test is not allowed here! (Why? Example?)

Algorithm and Remarks

- Implements the generic tree-search or graph-search algorithms.
- Implements fringe as priority queue that selects a node with minimal cost value $g(n)$.
- UCS can be a regarded a modification of BFS by expanding the cheapest rather than the shallowest node. *Note:* In contrast to BFS, the early goal test is not allowed here! (Why? Example?)
- UCS can also be regarded a special case of A^* (covered later this chapter), where no heuristic is used.

Algorithm and Remarks

- Implements the generic tree-search or graph-search algorithms.
- Implements fringe as priority queue that selects a node with minimal cost value $g(n)$.
- UCS can be a regarded a modification of BFS by expanding the cheapest rather than the shallowest node. *Note:* In contrast to BFS, the early goal test is not allowed here! (Why? Example?)
- UCS can also be regarded a special case of A^* (covered later this chapter), where no heuristic is used.
- UCS is equivalent to Dijkstra's algorithm.

Properties of Uniform Cost

Optimality

Properties of Uniform Cost

Optimality Yes

Properties of Uniform Cost

Optimality Yes

Completeness

Properties of Uniform Cost

Optimality Yes

Completeness Depends on duplicate management and action costs:

Properties of Uniform Cost

Optimality Yes

Completeness Depends on duplicate management and action costs:

- Tree search: If all action costs are strictly larger than 0.

Properties of Uniform Cost

Optimality Yes

Completeness Depends on duplicate management and action costs:

- Tree search: If all action costs are strictly larger than 0.
- Graph search: Yes (except for the strongest form of completeness)

Properties of Uniform Cost

Optimality Yes

Completeness Depends on duplicate management and action costs:

- Tree search: If all action costs are strictly larger than 0.
- Graph search: Yes (except for the strongest form of completeness)
- Again, for infinite transition systems the situation is more complicated.

Properties of Uniform Cost

Optimality Yes

Completeness Depends on duplicate management and action costs:

- Tree search: If all action costs are strictly larger than 0.
- Graph search: Yes (except for the strongest form of completeness)
- Again, for infinite transition systems the situation is more complicated.

Correctness Yes

Properties of Uniform Cost

Optimality Yes

Completeness Depends on duplicate management and action costs:

- Tree search: If all action costs are strictly larger than 0.
- Graph search: Yes (except for the strongest form of completeness)
- Again, for infinite transition systems the situation is more complicated.

Correctness Yes

Space

Properties of Uniform Cost

Optimality Yes

Completeness Depends on duplicate management and action costs:

- Tree search: If all action costs are strictly larger than 0.
- Graph search: Yes (except for the strongest form of completeness)
- Again, for infinite transition systems the situation is more complicated.

Correctness Yes

Space $O(b^{1+\lfloor g^*/\varepsilon \rfloor})$, where g^* denotes the cost of an optimal solution, and ε the (positive) cost of the cheapest action.

Properties of Uniform Cost

Optimality Yes

Completeness Depends on duplicate management and action costs:

- Tree search: If all action costs are strictly larger than 0.
- Graph search: Yes (except for the strongest form of completeness)
- Again, for infinite transition systems the situation is more complicated.

Correctness Yes

Space $O(b^{1+\lfloor g^*/\varepsilon \rfloor})$, where g^* denotes the cost of an optimal solution, and ε the (positive) cost of the cheapest action.

Time Similar to space.

Motivation

- So far, search was blind: we were sorting the fringe via FIFO, LIFO, or by costs.
- Search effort can be reduced significantly, if a heuristic is used to sort the fringe.

Motivation

- So far, search was blind: we were sorting the fringe via FIFO, LIFO, or by costs.
- Search effort can be reduced significantly, if a heuristic is used to sort the fringe.

Main issues to be solved:

- What are heuristics? Where do they come from?
- Are there *formal properties* of heuristics?

Motivation

- So far, search was blind: we were sorting the fringe via FIFO, LIFO, or by costs.
- Search effort can be reduced significantly, if a heuristic is used to sort the fringe.

Main issues to be solved:

- What are heuristics? Where do they come from?
- Are there *formal properties* of heuristics?
- How to use/integrate the heuristic? (This forms the algorithm.)

What are Heuristics?

Heuriskein (Greek): Find, Discover

1957
: Methods to identify problem-solving techniques, especially in the field of mathematical proofs.

1963
: Problem-solving processes that potentially deliver solutions.

1971
: "Rules" that domain experts apply in order to find good solutions.

At Present
: Techniques that improve the average performance of problem-solving methods, but not necessarily the worst-case performance.

In Search
: In the context of search methods: functions that estimate solution costs or the goal distance.

Sliding Tile Puzzle

2	1	4	8
9	7	11	10
6	5	15	3
13	14	12	

Problem

→

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

Solution

How far are we still away?

Sliding Tile Puzzle

2	1	4	8
9	7	11	10
6	5	15	3
13	14	12	

Problem

→

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

Solution

How far are we still away?

- Number of misplaced tiles

Sliding Tile Puzzle

2	1	4	8
9	7	11	10
6	5	15	3
13	14	12	

Problem

$\longrightarrow$

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

Solution

How far are we still away?

- Number of misplaced tiles
- “Distance” (horizontal and vertical distance) per tile to goal position $\rightarrow$ *Manhatten distance*

Sliding Tile Puzzle

2	1	4	8
9	7	11	10
6	5	15	3
13	14	12	

Problem

$\longrightarrow$

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

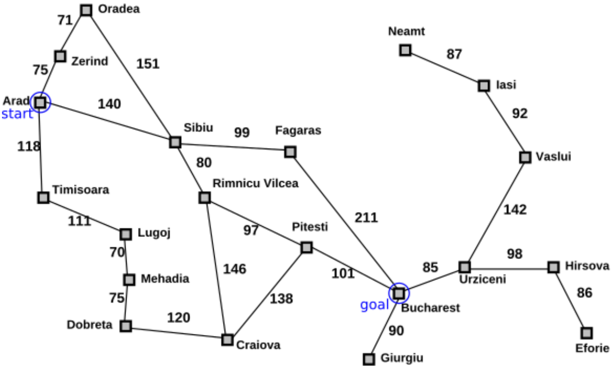
Solution

How far are we still away?

- Number of misplaced tiles
- "Distance" (horizontal and vertical distance) per tile to goal position → *Manhatten distance*
- Ignore certain tiles and use resulting solution cost as estimate.

Road Map

How to find a(n optimal/good) way from *Arad* to *Bucharest*?

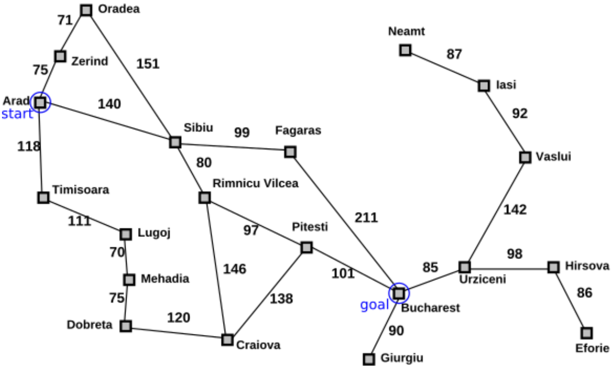


copyright: see slide 46[1] (modified)

Possible heuristics?

Road Map

How to find a(n optimal/good) way from *Arad* to *Bucharest*?

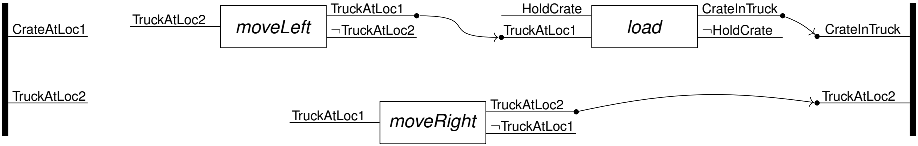


copyright: see slide 46[1] (modified)

Possible heuristics?

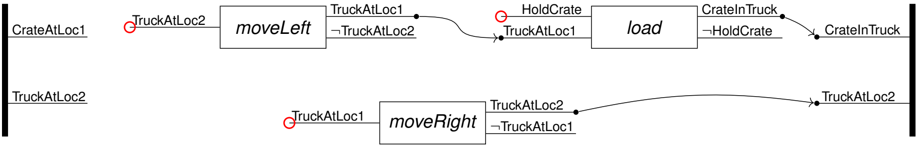
- Use the linear distance.

POCL Planning Problem



How many modifications do we have to perform?

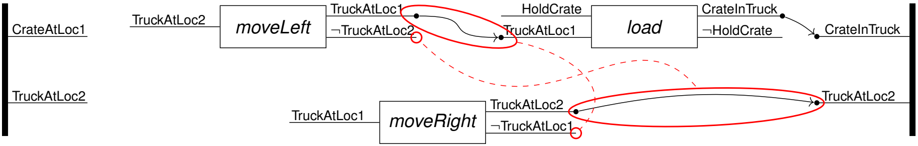
POCL Planning Problem



How many modifications do we have to perform?

- Number of open preconditions

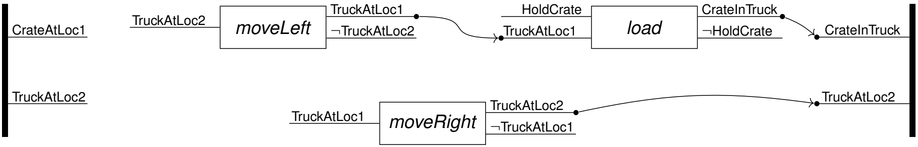
POCL Planning Problem



How many modifications do we have to perform?

- Number of open preconditions
- Number of causal treats

POCL Planning Problem



How many modifications do we have to perform?

- Number of open preconditions
- Number of causal treats
- There are a few POCL planning heuristics (for estimating the number of missing modifications or missing actions).

Heuristic Construction

How to come up with heuristics in a *domain-independent* way?

Heuristic Construction

How to come up with heuristics in a *domain-independent* way?

- Perform a *problem relaxation*.

Heuristic Construction

How to come up with heuristics in a *domain-independent* way?

- Perform a *problem relaxation*.
- Solve the relaxed problem.

Heuristic Construction

How to come up with heuristics in a *domain-independent* way?

- Perform a *problem relaxation*.
- Solve the relaxed problem.
- Use the cost (or number of actions, etc.) of the problem in the relaxed problem as approximation (i.e., heuristic) of the actual problem.

Heuristic Construction

How to come up with heuristics in a *domain-independent* way?

- Perform a *problem relaxation*.
- Solve the relaxed problem.
- Use the cost (or number of actions, etc.) of the problem in the relaxed problem as approximation (i.e., heuristic) of the actual problem.

Example Sliding Tile Puzzle:

- *Number of misplaced tiles.* Relaxation: We can always move tiles to any location, i.e., ignore all preconditions.

Heuristic Construction

How to come up with heuristics in a *domain-independent* way?

- Perform a *problem relaxation*.
- Solve the relaxed problem.
- Use the cost (or number of actions, etc.) of the problem in the relaxed problem as approximation (i.e., heuristic) of the actual problem.

Example Sliding Tile Puzzle:

- *Number of misplaced tiles.* Relaxation: We can always move tiles to any location, i.e., ignore all preconditions.
- *Manhatten distance.* Relaxation: We can move a tile, even if the neighbor tile is not free, i.e., ignore some preconditions.

Heuristic Construction

How to come up with heuristics in a *domain-independent* way?

- Perform a *problem relaxation*.
- Solve the relaxed problem.
- Use the cost (or number of actions, etc.) of the problem in the relaxed problem as approximation (i.e., heuristic) of the actual problem.

Example Sliding Tile Puzzle:

- *Number of misplaced tiles.* Relaxation: We can always move tiles to any location, i.e., ignore all preconditions.
- *Manhatten distance.* Relaxation: We can move a tile, even if the neighbor tile is not free, i.e., ignore some preconditions.
- *Ignore tiles.* Some tiles (i.e., state variables) do not exist.

Heuristic Construction

How to come up with heuristics in a *domain-independent* way?

- Perform a *problem relaxation*.
- Solve the relaxed problem.
- Use the cost (or number of actions, etc.) of the problem in the relaxed problem as approximation (i.e., heuristic) of the actual problem.

Example Sliding Tile Puzzle:

- *Number of misplaced tiles.* Relaxation: We can always move tiles to any location, i.e., ignore all preconditions.
- *Manhatten distance.* Relaxation: We can move a tile, even if the neighbor tile is not free, i.e., ignore some preconditions.
- *Ignore tiles.* Some tiles (i.e., state variables) do not exist.

In *planning*, we can exploit the underlying formalism to design a large set of domain-independent heuristics.

Definitions

Definition (Heuristic, Dominance)

Given a state transition system $ts = (S, L, c, T, I, G)$, a *heuristic h* is a function $h : S \to \mathbb{R}^+ \cup \{\infty\}$. A heuristic h_1 is said to dominate another heuristic h_2 if for all states $s \in S$, $h_1(s) \geq h_2(s)$.

Definitions

Definition (Heuristic, Dominance)

Given a state transition system $ts = (S, L, c, T, I, G)$, a *heuristic* h is a function $h : S \to \mathbb{R}^+ \cup \{\infty\}$. A heuristic h_1 is said to dominate another heuristic h_2 if for all states $s \in S$, $h_1(s) \geq h_2(s)$.

Heuristics can estimate different metrics (cf. also last lecture). Most common ones are:

Definitions

Definition (Heuristic, Dominance)

Given a state transition system $ts = (S, L, c, T, I, G)$, a *heuristic h* is a function $h : S \to \mathbb{R}^+ \cup \{\infty\}$. A heuristic h_1 is said to dominate another heuristic h_2 if for all states $s \in S$, $h_1(s) \geq h_2(s)$.

Heuristics can estimate different metrics (cf. also last lecture). Most common ones are:

- Number of actions of a solution.

Definitions

Definition (Heuristic, Dominance)

Given a state transition system $ts = (S, L, c, T, I, G)$, a *heuristic h* is a function $h : S \to \mathbb{R}^+ \cup \{\infty\}$. A heuristic h_1 is said to dominate another heuristic h_2 if for all states $s \in S$, $h_1(s) \geq h_2(s)$.

Heuristics can estimate different metrics (cf. also last lecture). Most common ones are:

- Number of actions of a solution.
- Costs of a solution.

Definitions

Definition (Heuristic, Dominance)

Given a state transition system $ts = (S, L, c, T, I, G)$, a *heuristic* h is a function $h : S \to \mathbb{R}^+ \cup \{\infty\}$. A heuristic h_1 is said to dominate another heuristic h_2 if for all states $s \in S$, $h_1(s) \geq h_2(s)$.

Heuristics can estimate different metrics (cf. also last lecture). Most common ones are:

- Number of actions of a solution.
- Costs of a solution.
- *Note:* The states of a transition system are not necessarily the same as the states of a planning problem! This depends on the search procedure and the problem class.

Definitions, cont'd I

Definition (Perfect Heuristic)

A heuristic $h^* : S \to \mathbb{R}^+$ is called *perfect*, if for all states $s \in S$ $h^*(s)$ is the cost of the cheapest transition from s to a goal $s' \in G$. Further, $h^*(s) = \infty$ for all states s for which no goal state can be reached.

Definitions, cont'd I

Definition (Perfect Heuristic)

A heuristic $h^* : S \to \mathbb{R}^+$ is called *perfect*, if for all states $s \in S$ $h^*(s)$ is the cost of the cheapest transition from s to a goal $s' \in G$. Further, $h^*(s) = \infty$ for all states s for which no goal state can be reached.

Definition (Safe Heuristic)

A heuristic h is called *safe*, if for all states $s \in S$ $h(s) = \infty$ implies $h^*(s) = \infty$.

Definitions, cont'd I

Definition (Perfect Heuristic)

A heuristic $h^* : S \to \mathbb{R}^+$ is called *perfect*, if for all states $s \in S$ $h^*(s)$ is the cost of the cheapest transition from s to a goal $s' \in G$. Further, $h^*(s) = \infty$ for all states s for which no goal state can be reached.

Definition (Safe Heuristic)

A heuristic h is called *safe*, if for all states $s \in S$ $h(s) = \infty$ implies $h^*(s) = \infty$.

Definition (Goal-aware Heuristic)

A heuristic h is called *goal-aware*, if all goal states, i.e., $s_G \in G$ holds $h(s_G) = 0$.

Definitions, cont'd II

Definition (Admissible Heuristics)

A heuristic h is called *admissible*, if for all states $s \in S$, it holds $h(s) \leq h^*(s)$.

Explanation:

Admissible heuristics give a lower (i.e., non-overestimating) bound on the "best" goal distance.

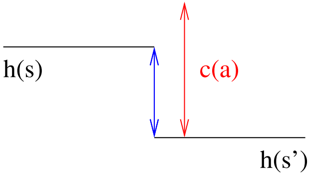
Definitions, cont'd III

Definition (Consistent Heuristics)

A heuristic h is called *consistent*, if for all transitions $(s, l, s') \in T$ holds $h(s) - h(s') \leq c(l)$ (equiv.: $h(s) \leq c(l) + h(s')$).

Explanation:

Consistency: When applying an action *a*, the heuristic value cannot decrease by more than the cost of *a*.



How to Use the Heuristic During Search?

For selecting a search node from the search fringe, pick a search node n with the cheapest f value. f can depend on many properties, for example:

- The heuristic value $h(n)$ of n.
- The cost value $g(n)$ of n.
- The depth $d(n)$ of n.

Informed Search Algorithms

The most basic informed search algorithms are:

Informed Search Algorithms

The most basic informed search algorithms are:

Greedy $f(n) = h(n)$ (also called *Greedy Best-first*)

Explanation:

- Greedy search always expands the node that seems closest to a goal.

Informed Search Algorithms

The most basic informed search algorithms are:

Greedy $f(n) = h(n)$ (also called *Greedy Best-first*)

A^* $f(n) = g(n) + h(n)$

Explanation:

- Greedy search always expands the node that seems closest to a goal.
- A^* tries to find a cost-minimal solution while taking the heuristic into account during search.

Informed Search Algorithms

The most basic informed search algorithms are:

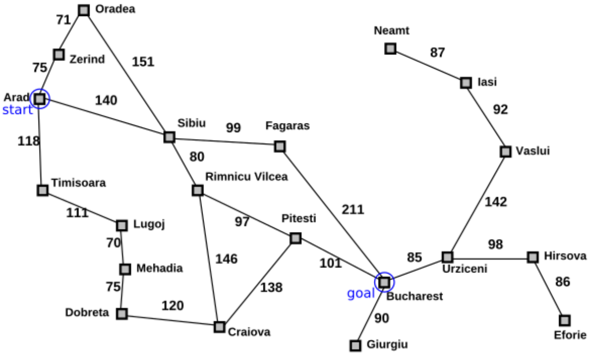
- **Greedy** $f(n) = h(n)$ (also called *Greedy Best-first*)
- **A^*** $f(n) = g(n) + h(n)$
- **Greedy A^*** $f(n) = c(n) + w * h(n), w > 1$ (also called *Weighted A^**)

Explanation:

- Greedy search always expands the node that seems closest to a goal.
- A^* tries to find a cost-minimal solution while taking the heuristic into account during search.
- Greedy A^* is a “more greedy” version of A^*, taking into account the heuristic to a larger extent.

The Search Problem

How to find a(n optimal/good) way from *Arad* to *Bucharest*?



Straight−line distance to Bucharest

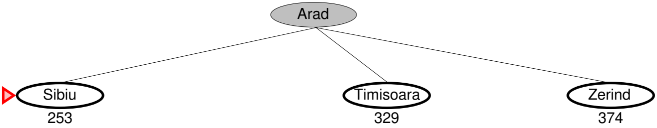
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

The Search Progress

Reminder:

- Always select a node with minimal $f(n) = h(n)$.
- Here, h is the linear distance.

The Search Progress

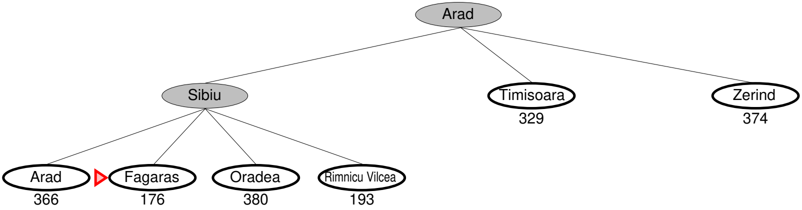


copyright: see slide 46[1] (modified)

Reminder:

- Always select a node with minimal $f(n) = h(n)$.
- Here, h is the linear distance.

The Search Progress

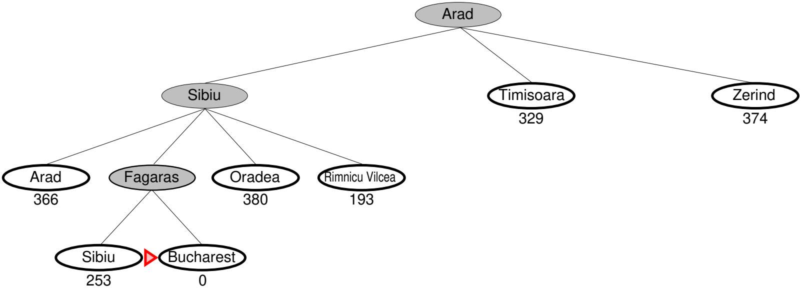


copyright: see slide 46[1] (modified)

Reminder:

- Always select a node with minimal $f(n) = h(n)$.
- Here, h is the linear distance.

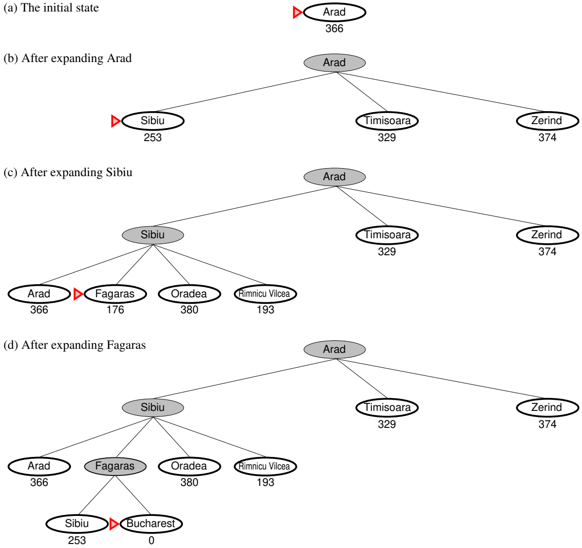
The Search Progress



Reminder:

- Always select a node with minimal $f(n) = h(n)$.
- Here, h is the linear distance.

The Search Progress, Overview



Properties of Greedy Search

Optimality

Properties of Greedy Search

Optimality No (even for admissible and consistent heuristics)

Properties of Greedy Search

Optimality No (even for admissible and consistent heuristics)

Completeness

Properties of Greedy Search

Optimality	No (even for admissible and consistent heuristics)
Completeness	No (can get stuck in loops)

Properties of Greedy Search

Optimality	No (even for admissible and consistent heuristics)
Completeness	No (can get stuck in loops)
Correctness	Yes

Properties of Greedy Search

Optimality	No (even for admissible and consistent heuristics)
Completeness	No (can get stuck in loops)
Correctness	Yes
Space	

Properties of Greedy Search

Optimality	No (even for admissible and consistent heuristics)
Completeness	No (can get stuck in loops)
Correctness	Yes
Space	$O(b^m)$

Properties of Greedy Search

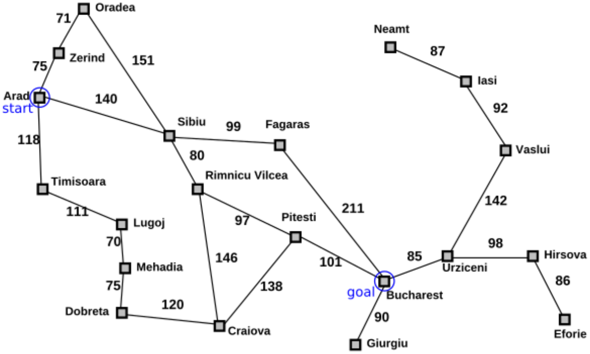
Optimality	No (even for admissible and consistent heuristics)
Completeness	No (can get stuck in loops)
Correctness	Yes
Space	$O(b^m)$
Time	

Properties of Greedy Search

Optimality	No (even for admissible and consistent heuristics)
Completeness	No (can get stuck in loops)
Correctness	Yes
Space	$O(b^m)$
Time	Just as space.

The Search Problem

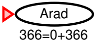
How to find a(n optimal/good) way from *Arad* to *Bucharest*?



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

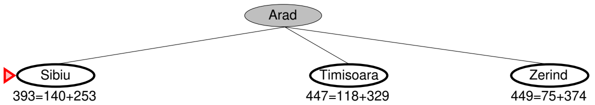
The Search Progress



Reminder:

- Always select a node with minimal $f(n) = g(n) + h(n)$.
- Here, h is the linear distance.

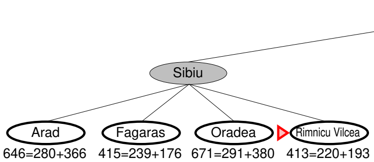
The Search Progress



Reminder:

- Always select a node with minimal $f(n) = g(n) + h(n)$.
- Here, h is the linear distance.

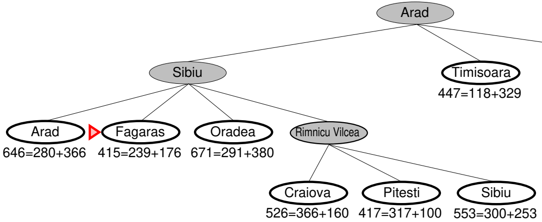
The Search Progress



Reminder:

- Always select a node with minimal $f(n) = g(n) + h(n)$.
- Here, h is the linear distance.

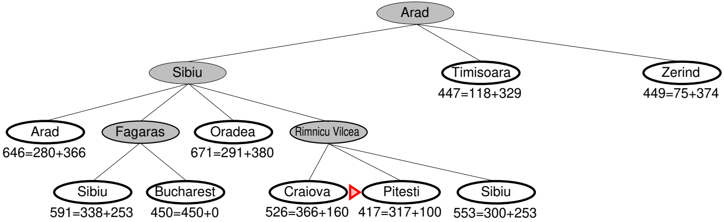
The Search Progress



Reminder:

- Always select a node with minimal $f(n) = g(n) + h(n)$.
- Here, h is the linear distance.

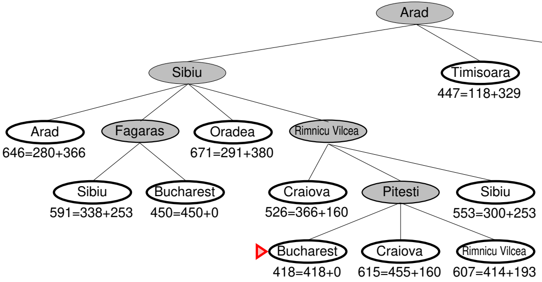
The Search Progress



Reminder:

- Always select a node with minimal $f(n) = g(n) + h(n)$.
- Here, h is the linear distance.

The Search Progress

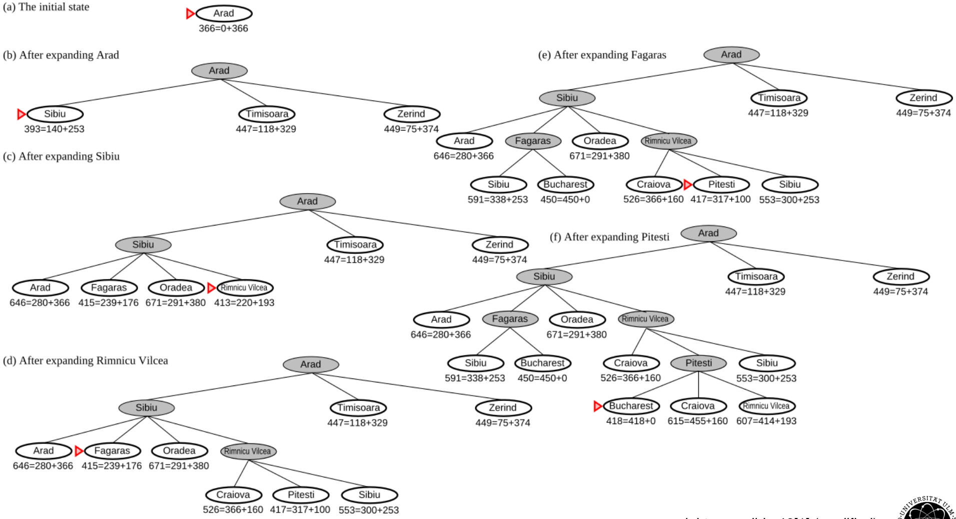


copyright: see slide 46[1] (modified)

Reminder:

- Always select a node with minimal $f(n) = g(n) + h(n)$.
- Here, h is the linear distance.

The Search Progress, Overview



Properties of A^*

Optimality

Properties of A^*

Optimality Depends on duplicate management and heuristics:

Properties of A^*

Optimality Depends on duplicate management and heuristics:

- Tree search: If the heuristic is admissible.

Properties of A^*

Optimality Depends on duplicate management and heuristics:

- Tree search: If the heuristic is admissible.
- Graph search: If the heuristic is consistent.

Properties of A^*

Optimality Depends on duplicate management and heuristics:

- Tree search: If the heuristic is admissible.
- Graph search: If the heuristic is consistent.
- Again, for infinite transition systems, the situation is more complicated.

Properties of A^*

Optimality Depends on duplicate management and heuristics:

- Tree search: If the heuristic is admissible.
- Graph search: If the heuristic is consistent.
- Again, for infinite transition systems, the situation is more complicated.

Completeness

Properties of A^*

Optimality Depends on duplicate management and heuristics:

- Tree search: If the heuristic is admissible.
- Graph search: If the heuristic is consistent.
- Again, for infinite transition systems, the situation is more complicated.

Completeness Depends on duplicate management and action costs:

Properties of A^*

Optimality Depends on duplicate management and heuristics:

- Tree search: If the heuristic is admissible.
- Graph search: If the heuristic is consistent.
- Again, for infinite transition systems, the situation is more complicated.

Completeness Depends on duplicate management and action costs:

- Tree search: If action costs are strictly larger than 0.

Properties of A^*

Optimality Depends on duplicate management and heuristics:

- Tree search: If the heuristic is admissible.
- Graph search: If the heuristic is consistent.
- Again, for infinite transition systems, the situation is more complicated.

Completeness Depends on duplicate management and action costs:

- Tree search: If action costs are strictly larger than 0.
- Graph search: Yes (except for the strongest form of completeness)

Properties of A^*

Optimality Depends on duplicate management and heuristics:

- Tree search: If the heuristic is admissible.
- Graph search: If the heuristic is consistent.
- Again, for infinite transition systems, the situation is more complicated.

Completeness Depends on duplicate management and action costs:

- Tree search: If action costs are strictly larger than 0.
- Graph search: Yes (except for the strongest form of completeness)
- Again, for infinite transition systems the situation is more complicated.

Properties of A^*

Optimality Depends on duplicate management and heuristics:

- Tree search: If the heuristic is admissible.
- Graph search: If the heuristic is consistent.
- Again, for infinite transition systems, the situation is more complicated.

Completeness Depends on duplicate management and action costs:

- Tree search: If action costs are strictly larger than 0.
- Graph search: Yes (except for the strongest form of completeness)
- Again, for infinite transition systems the situation is more complicated.

Correctness Yes

Properties of A^*

Optimality Depends on duplicate management and heuristics:
- Tree search: If the heuristic is admissible.
- Graph search: If the heuristic is consistent.
- Again, for infinite transition systems, the situation is more complicated.

Completeness Depends on duplicate management and action costs:
- Tree search: If action costs are strictly larger than 0.
- Graph search: Yes (except for the strongest form of completeness)
- Again, for infinite transition systems the situation is more complicated.

Correctness Yes

Space

Properties of A^*

Optimality Depends on duplicate management and heuristics:

- Tree search: If the heuristic is admissible.
- Graph search: If the heuristic is consistent.
- Again, for infinite transition systems, the situation is more complicated.

Completeness Depends on duplicate management and action costs:

- Tree search: If action costs are strictly larger than 0.
- Graph search: Yes (except for the strongest form of completeness)
- Again, for infinite transition systems the situation is more complicated.

Correctness Yes

Space $O(b^{1+\lfloor g^*/\varepsilon \rfloor})$, where g^* denotes the cost of an optimal solution, and ε the (positive) cost of the cheapest action.

Properties of A^*

Optimality Depends on duplicate management and heuristics:

- Tree search: If the heuristic is admissible.
- Graph search: If the heuristic is consistent.
- Again, for infinite transition systems, the situation is more complicated.

Completeness Depends on duplicate management and action costs:

- Tree search: If action costs are strictly larger than 0.
- Graph search: Yes (except for the strongest form of completeness)
- Again, for infinite transition systems the situation is more complicated.

Correctness Yes

Space $O(b^{1+\lfloor g^*/\varepsilon \rfloor})$, where g^* denotes the cost of an optimal solution, and ε the (positive) cost of the cheapest action.

Time

Properties of A^*

Optimality Depends on duplicate management and heuristics:

- Tree search: If the heuristic is admissible.
- Graph search: If the heuristic is consistent.
- Again, for infinite transition systems, the situation is more complicated.

Completeness Depends on duplicate management and action costs:

- Tree search: If action costs are strictly larger than 0.
- Graph search: Yes (except for the strongest form of completeness)
- Again, for infinite transition systems the situation is more complicated.

Correctness Yes

Space $O(b^{1+\lfloor g^*/\varepsilon \rfloor})$, where g^* denotes the cost of an optimal solution, and ε the (positive) cost of the cheapest action.

Time Just as space.

Properties of Greedy A^*

Optimality

Properties of Greedy A^*

Optimality Depends on duplicate management and heuristics:

Properties of Greedy A^*

Optimality Depends on duplicate management and heuristics:

- Tree search: for $w > 1$, it's bounded suboptimal if the heuristic is admissible.

- bounded suboptimal: the solutions returned are at most a factor w more costly than the optimal ones.

Properties of Greedy A^*

Optimality Depends on duplicate management and heuristics:

- Tree search: for $w > 1$, it's bounded suboptimal if the heuristic is admissible.
- Graph search: for $w > 1$, it's bounded suboptimal if the heuristic is consistent.
- bounded suboptimal: the solutions returned are at most a factor w more costly than the optimal ones.

Properties of Greedy A^*

Optimality Depends on duplicate management and heuristics:

- Tree search: for $w > 1$, it's bounded suboptimal if the heuristic is admissible.
- Graph search: for $w > 1$, it's bounded suboptimal if the heuristic is consistent.
- bounded suboptimal: the solutions returned are at most a factor w more costly than the optimal ones.
- Again, for infinite transition systems, the situation is more complicated.

Properties of Greedy A^*

Optimality Depends on duplicate management and heuristics:

- Tree search: for $w > 1$, it's bounded suboptimal if the heuristic is admissible.
- Graph search: for $w > 1$, it's bounded suboptimal if the heuristic is consistent.
- bounded suboptimal: the solutions returned are at most a factor w more costly than the optimal ones.
- Again, for infinite transition systems, the situation is more complicated.

Completeness

Properties of Greedy A^*

Optimality Depends on duplicate management and heuristics:

- Tree search: for $w > 1$, it's bounded suboptimal if the heuristic is admissible.
- Graph search: for $w > 1$, it's bounded suboptimal if the heuristic is consistent.
- bounded suboptimal: the solutions returned are at most a factor w more costly than the optimal ones.
- Again, for infinite transition systems, the situation is more complicated.

Completeness It depends... (see A^* – no difference)

Properties of Greedy A^*

Optimality Depends on duplicate management and heuristics:

- Tree search: for $w > 1$, it's bounded suboptimal if the heuristic is admissible.
- Graph search: for $w > 1$, it's bounded suboptimal if the heuristic is consistent.
- bounded suboptimal: the solutions returned are at most a factor w more costly than the optimal ones.
- Again, for infinite transition systems, the situation is more complicated.

Completeness It depends... (see A^* – no difference)

Correctness Yes

Properties of Greedy A^*

Optimality Depends on duplicate management and heuristics:

- Tree search: for $w > 1$, it's bounded suboptimal if the heuristic is admissible.
- Graph search: for $w > 1$, it's bounded suboptimal if the heuristic is consistent.
- bounded suboptimal: the solutions returned are at most a factor w more costly than the optimal ones.
- Again, for infinite transition systems, the situation is more complicated.

Completeness It depends... (see A^* – no difference)

Correctness Yes

Space

Properties of Greedy A^*

Optimality Depends on duplicate management and heuristics:

- Tree search: for $w > 1$, it's bounded suboptimal if the heuristic is admissible.
- Graph search: for $w > 1$, it's bounded suboptimal if the heuristic is consistent.
- bounded suboptimal: the solutions returned are at most a factor w more costly than the optimal ones.
- Again, for infinite transition systems, the situation is more complicated.

Completeness It depends... (see A^* – no difference)

Correctness Yes

Space $O(b^m)$

Properties of Greedy A^*

Optimality Depends on duplicate management and heuristics:

- Tree search: for $w > 1$, it's bounded suboptimal if the heuristic is admissible.
- Graph search: for $w > 1$, it's bounded suboptimal if the heuristic is consistent.
- bounded suboptimal: the solutions returned are at most a factor w more costly than the optimal ones.
- Again, for infinite transition systems, the situation is more complicated.

Completeness It depends... (see A^* – no difference)

Correctness Yes

Space $O(b^m)$

Time

Properties of Greedy A^*

Optimality Depends on duplicate management and heuristics:

- Tree search: for $w > 1$, it's bounded suboptimal if the heuristic is admissible.
- Graph search: for $w > 1$, it's bounded suboptimal if the heuristic is consistent.
- bounded suboptimal: the solutions returned are at most a factor w more costly than the optimal ones.
- Again, for infinite transition systems, the situation is more complicated.

Completeness It depends... (see A^* – no difference)

Correctness Yes

Space $O(b^m)$

Time Just as space.

Notes about Greedy A^*

- Greedy A^*, $f(n) = g(n) + w * h(h)$, allows to interpolate between greedy search and A^* search.
- Greedy A^* trades off plan quality against computational effort. (Normally, Greedy A^* is much more efficient than A^*.)

Notes about Greedy A^*

- Greedy A^*, $f(n) = g(n) + w * h(h)$, allows to interpolate between greedy search and A^* search.
- Greedy A^* trades off plan quality against computational effort. (Normally, Greedy A^* is much more efficient than A^*.)

How does Greedy A^* behave for different weights $w \in \mathbb{R}_0^+$?

- How does it behave for $w = 0$?
- How does it behave for $w = 1$?
- How does it behave for $w = 10^{101010}$?

Notes about Greedy A^*

- Greedy A^*, $f(n) = g(n) + w * h(h)$, allows to interpolate between greedy search and A^* search.
- Greedy A^* trades off plan quality against computational effort. (Normally, Greedy A^* is much more efficient than A^*.)

How does Greedy A^* behave for different weights $w \in \mathbb{R}_0^+$?

- How does it behave for $w = 0$? Uniform-cost Search
- How does it behave for $w = 1$?
- How does it behave for $w = 10^{101010}$?

Notes about Greedy A^*

- Greedy A^*, $f(n) = g(n) + w * h(h)$, allows to interpolate between greedy search and A^* search.
- Greedy A^* trades off plan quality against computational effort. (Normally, Greedy A^* is much more efficient than A^*.)

How does Greedy A^* behave for different weights $w \in \mathbb{R}_0^+$?

- How does it behave for $w = 0$? Uniform-cost Search
- How does it behave for $w = 1$? A^* Search
- How does it behave for $w = 10^{101010}$?

Notes about Greedy A^*

- Greedy A^*, $f(n) = g(n) + w * h(h)$, allows to interpolate between greedy search and A^* search.
- Greedy A^* trades off plan quality against computational effort. (Normally, Greedy A^* is much more efficient than A^*.)

How does Greedy A^* behave for different weights $w \in \mathbb{R}_0^+$?

- How does it behave for $w = 0$? Uniform-cost Search
- How does it behave for $w = 1$? A^* Search
- How does it behave for $w = 10^{101010}$? Greedy Search

Summary

- Search is a method to systematically (and with known properties) find solutions in state transition systems, i.e., a sequence of transitions from the initial state to a goal state.

Summary

- Search is a method to systematically (and with known properties) find solutions in state transition systems, i.e., a sequence of transitions from the initial state to a goal state.
- Interesting properties of search algorithms are:

Summary

- Search is a method to systematically (and with known properties) find solutions in state transition systems, i.e., a sequence of transitions from the initial state to a goal state.
- Interesting properties of search algorithms are:
  - Space and time requirement.

Summary

- Search is a method to systematically (and with known properties) find solutions in state transition systems, i.e., a sequence of transitions from the initial state to a goal state.
- Interesting properties of search algorithms are:
  - Space and time requirement.
  - Optimality, completeness.

Summary

- Search is a method to systematically (and with known properties) find solutions in state transition systems, i.e., a sequence of transitions from the initial state to a goal state.
- Interesting properties of search algorithms are:
  - Space and time requirement.
  - Optimality, completeness.
- Two important distinctions in search are:

Summary

- Search is a method to systematically (and with known properties) find solutions in state transition systems, i.e., a sequence of transitions from the initial state to a goal state.
- Interesting properties of search algorithms are:
  - Space and time requirement.
  - Optimality, completeness.
- Two important distinctions in search are:
  - Tree search vs. graph search.

Summary

- Search is a method to systematically (and with known properties) find solutions in state transition systems, i.e., a sequence of transitions from the initial state to a goal state.
- Interesting properties of search algorithms are:
  - Space and time requirement.
  - Optimality, completeness.
- Two important distinctions in search are:
  - Tree search vs. graph search.
  - Uninformed search vs. informed search.

Summary

- Search is a method to systematically (and with known properties) find solutions in state transition systems, i.e., a sequence of transitions from the initial state to a goal state.
- Interesting properties of search algorithms are:
  - Space and time requirement.
  - Optimality, completeness.
- Two important distinctions in search are:
  - Tree search vs. graph search.
  - Uninformed search vs. informed search.
- In informed search, heuristics play a central role. Important properties are:

Summary

- Search is a method to systematically (and with known properties) find solutions in state transition systems, i.e., a sequence of transitions from the initial state to a goal state.
- Interesting properties of search algorithms are:
  - Space and time requirement.
  - Optimality, completeness.
- Two important distinctions in search are:
  - Tree search vs. graph search.
  - Uninformed search vs. informed search.
- In informed search, heuristics play a central role. Important properties are:
  - Admissibility.

Summary

- Search is a method to systematically (and with known properties) find solutions in state transition systems, i.e., a sequence of transitions from the initial state to a goal state.
- Interesting properties of search algorithms are:
  - Space and time requirement.
  - Optimality, completeness.
- Two important distinctions in search are:
  - Tree search vs. graph search.
  - Uninformed search vs. informed search.
- In informed search, heuristics play a central role. Important properties are:
  - Admissibility.
  - Consistency.

Copyright Notes and Licenses

[1] Title: *Artificial Intelligence: A Modern Approach (Third Edition)*
Url: `https://aima.cs.berkeley.edu/`
Authors: *Stuart Russel* and *Peter Norvig*