State-Space Search and the STRIPS Planner

Searching for a Path through a Graph of Nodes Representing World States

Literature

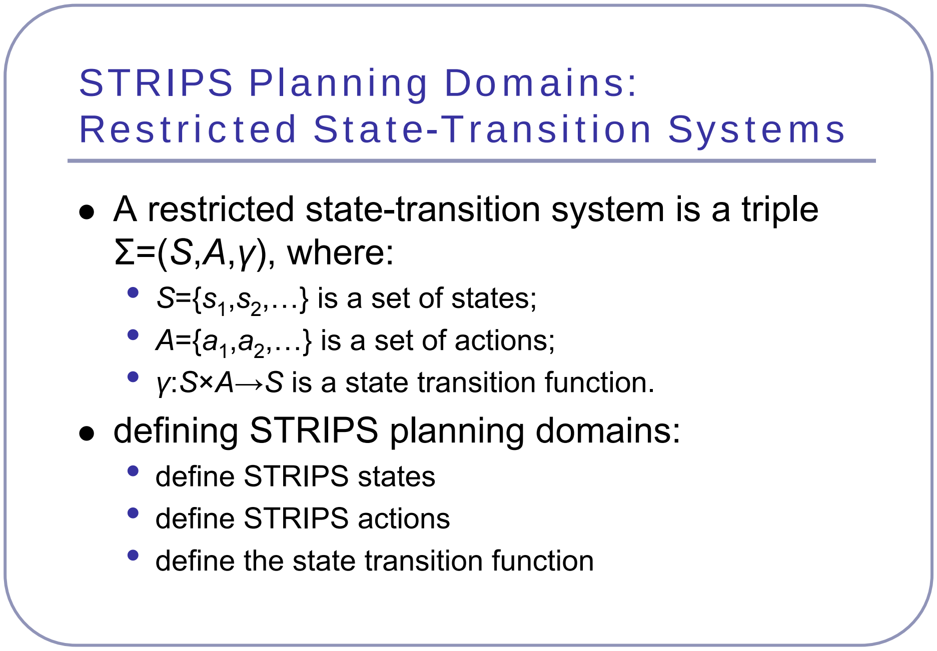
- Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning – Theory and Practice*, chapter 2 and 4. Elsevier/Morgan Kaufmann, 2004.
- Malik Ghallab, *et al*. PDDL–The Planning Domain Definition Language, Version 1.x. ftp://ftp.cs.yale.edu/pub/mcdermott/software/pddl.tar.gz
- S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*, chapters 3-4. Prentice Hall, 2nd edition, 2003.
- J. Pearl. *Heuristics*, chapters 1-2. Addison-Wesley, 1984.

Classical Representations

- propositional representation
  - world state is set of propositions
  - action consists of precondition propositions, propositions to be added and removed
- STRIPS representation
  - like propositional representation, but first-order literals instead of propositions
- state-variable representation
  - state is tuple of state variables $\{x_1,\ldots,x_n\}$
  - action is partial function over states

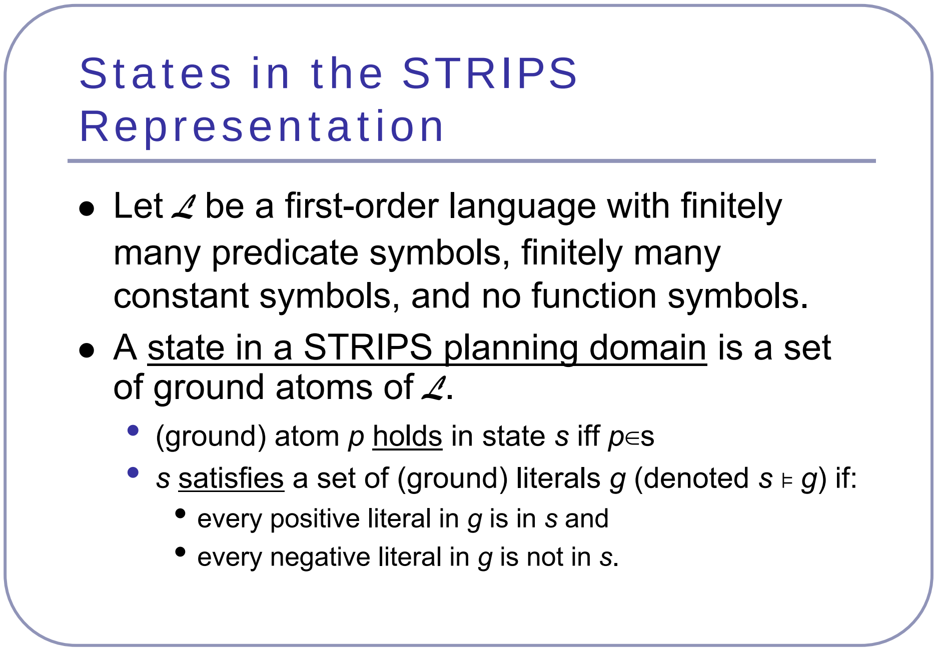
Overview

- ➡ The STRIPS Representation
- The Planning Domain Definition Language (PDDL)
- Problem-Solving by Search
- Heuristic Search
- Forward State-Space Search
- Backward State-Space Search
- The STRIPS Planner



STRIPS Planning Domains: Restricted State-Transition Systems

- A restricted state-transition system is a triple $\Sigma=(S,A,\gamma)$, where:
  - $S=\{s_1,s_2,\ldots\}$ is a set of states;
  - $A=\{a_1,a_2,\ldots\}$ is a set of actions;
  - $\gamma{:}S\times A\rightarrow S$ is a state transition function.
- defining STRIPS planning domains:
  - define STRIPS states
  - define STRIPS actions
  - define the state transition function

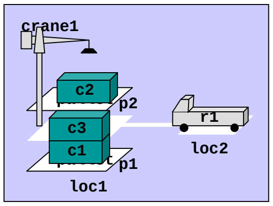


States in the STRIPS Representation

- Let $\mathcal{L}$ be a first-order language with finitely many predicate symbols, finitely many constant symbols, and no function symbols.
- A state in a STRIPS planning domain is a set of ground atoms of $\mathcal{L}$.
  - (ground) atom p holds in state s iff $p\in s$
  - s satisfies a set of (ground) literals g (denoted $s \models g$) if:
    - every positive literal in g is in s and
    - every negative literal in g is not in s.

DWR Example: STRIPS States

state = {attached(p1,loc1),
attached(p2,loc1),
in(c1,p1),in(c3,p1),
top(c3,p1), on(c3,c1),
on(c1,pallet), in(c2,p2),
top(c2,p2), on(c2,pallet),
belong(crane1,loc1),
empty(crane1),
adjacent(loc1,loc2),
adjacent(loc2, loc1),
at(r1,loc2), occupied(loc2),
unloaded(r1)}



Fluent Relations

- Predicates that represent relations, the truth value of which can change from state to state, are called a <u>fluent</u> or <u>flexible relations</u>.
  - example: at
- A state-invariant predicate is called a <u>rigid relation</u>.
  - example: adjacent

Operators and Actions in STRIPS Planning Domains

- A planning operator in a STRIPS planning domain is a triple
  o = (name(*o*), precond(*o*), effects(*o*)) where:
  - the name of the operator name(*o*) is a syntactic expression of the form $n(x_1,\ldots,x_k)$ where *n* is a (unique) symbol and $x_1,\ldots,x_k$ are all the variables that appear in *o*, and
  - the preconditions precond(*o*) and the effects effects(*o*) of the operator are sets of literals.
- An action in a STRIPS planning domain is a ground instance of a planning operator.

DWR Example: STRIPS Operators

- move(*r*,*l*,*m*)
  - precond: adjacent(*l*,*m*), at(*r*,*l*), ¬occupied(*m*)
  - effects: at(*r*,*m*), occupied(*m*), ¬occupied(*l*), ¬at(*r*,*l*)

- load(*k*,*l*,*c*,*r*)
  - precond: belong(*k*,*l*), holding(*k*,*c*), at(*r*,*l*), unloaded(*r*)
  - effects: empty(*k*), ¬holding(*k*,*c*), loaded(*r*,*c*), ¬unloaded(*r*)

- put(*k*,*l*,*c*,*d*,*p*)
  - precond: belong(*k*,*l*), attached(*p*,*l*), holding(*k*,*c*), top(*d*,*p*)
  - effects: ¬holding(*k*,*c*), empty(*k*), in(*c*,*p*), top(*c*,*p*), on(*c*,*d*), ¬top(*d*,*p*)

Applicability and State Transitions

- Let L be a set of literals.
  - $\underline{L^+}$ is the set of atoms that are positive literals in L and
  - $\underline{L^-}$ is the set of all atoms whose negations are in L.
- Let a be an action and s a state. Then a is <u>applicable</u> in s iff:
  - precond$^+(a) \subseteq s$; and
  - precond$^-(a) \cap s = \{\}$.
- The state transition function γ for an applicable action a in state s is defined as:
  - $\underline{\gamma(s,a)} = (s - \text{effects}^-(a)) \cup \text{effects}^+(a)$

STRIPS Planning Domains

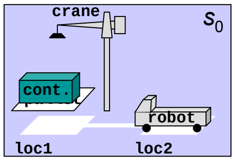
- Let $\mathcal{L}$ be a function-free first-order language. A <u>STRIPS planning domain on $\mathcal{L}$</u> is a restricted state-transition system $\Sigma=(S,A,\gamma)$ such that:
  - S is a set of STRIPS states, i.e. sets of ground atoms
  - A is a set of ground instances of some STRIPS planning operators O
  - γ:$S \times A \rightarrow S$ where
    - $\gamma(s,a)=(s - \text{effects}^-(a)) \cup \text{effects}^+(a)$ if a is applicable in s
    - $\gamma(s,a)$=undefined otherwise
  - S is closed under γ

STRIPS Planning Problems

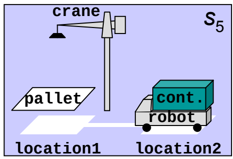
- A STRIPS planning problem is a triple $\mathcal{P}=(\Sigma,s_i,g)$ where:
  - $\Sigma=(S,A,\gamma)$ is a STRIPS planning domain on some first-order language $\mathcal{L}$
  - $s_i \in S$ is the initial state
  - g is a set of ground literals describing the goal such that the set of goal states is: $S_g=\{s \in S \mid s \text{ satisfies } g\}$

DWR Example: STRIPS Planning Problem

- Σ: STRIPS planning domain for DWR domain
- s_i: any state
  - example: s_0 = {attached(pile,loc1), in(cont,pile), top(cont,pile), on(cont,pallet), belong(crane,loc1), empty(crane), adjacent(loc1,loc2), adjacent(loc2,loc1), at(robot,loc2), occupied(loc2), unloaded(robot)}



- g: any subset of L
  - example: g = {¬unloaded(robot), at(robot,loc2)}, i.e. $S_g=\{s_5\}$



Statement of a STRIPS Planning Problem

- A statement of a STRIPS planning problem is a triple P=(O,s_i,g) where:
  - O is a set of planning operators in an appropriate STRIPS planning domain Σ=(S,A,γ) on $\mathcal{L}$
  - s_i is the initial state in an appropriate STRIPS planning problem $\mathcal{P}$=(Σ,s_i,g)
  - g is a goal (set of ground literals) in the same STRIPS planning problem $\mathcal{P}$

Classical Plans

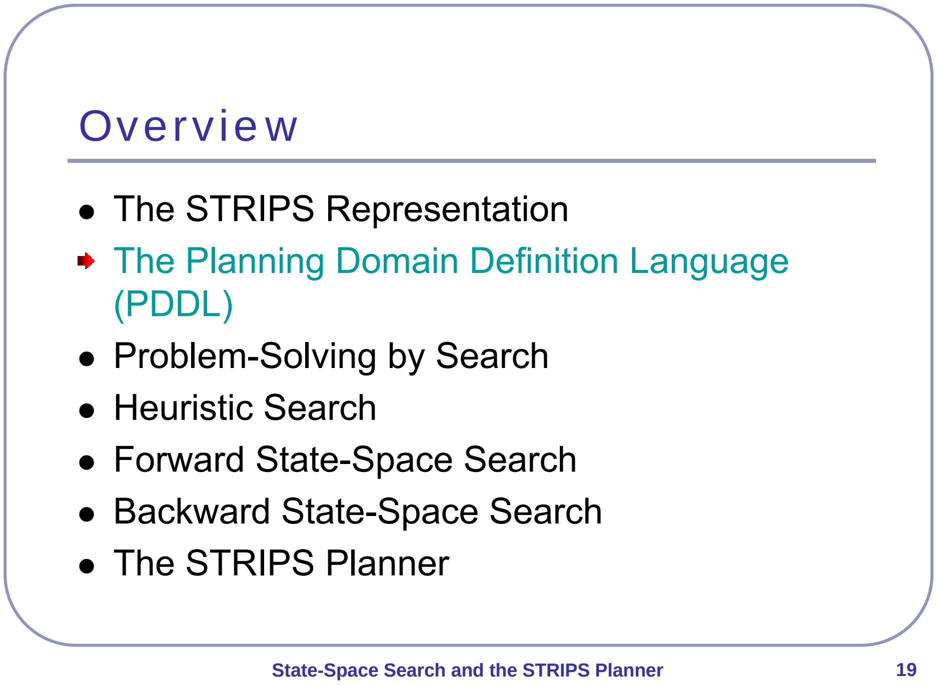
- A plan is any sequence of actions $\pi=\langle a_1,\ldots,a_k\rangle$, where $k\geq0$.
  - The length of plan π is $|\pi|=k$, the number of actions.
  - If $\pi_1=\langle a_1,\ldots,a_k\rangle$ and $\pi_2=\langle a'_1,\ldots,a'_j\rangle$ are plans, then their concatenation is the plan $\pi_1\bullet\pi_2=\langle a_1,\ldots,a_k,a'_1,\ldots,a'_j\rangle$.
  - The extended state transition function for plans is defined as follows:
    - $\gamma(s,\pi)=s$ if k=0 (π is empty)
    - $\gamma(s,\pi)=\gamma(\gamma(s,a_1),\langle a_2,\ldots,a_k\rangle)$ if k>0 and a_1 applicable in s
    - $\gamma(s,\pi)$=undefined otherwise

Classical Solutions

- Let $\mathcal{P}$=(Σ,s_i,g) be a planning problem. A plan π is a <u>solution</u> for $\mathcal{P}$ if $\gamma(s_i,\pi)$ satisfies g.
  - A solution π is <u>redundant</u> if there is a proper subsequence of π is also a solution for $\mathcal{P}$.
  - π is <u>minimal</u> if no other solution for $\mathcal{P}$ contains fewer actions than π.

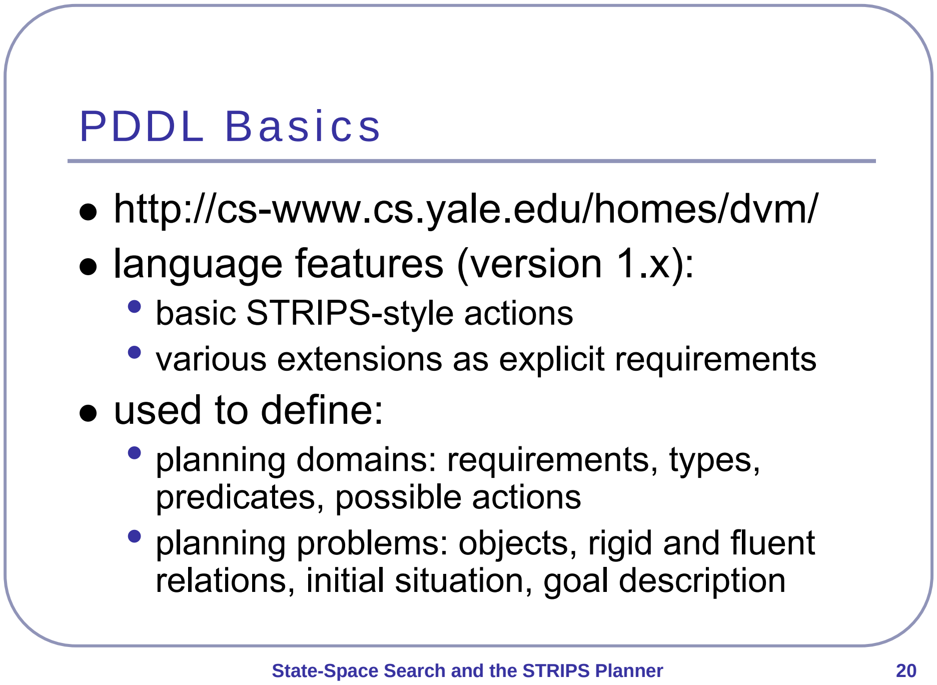
DWR Example: Solution Plan

- plan π_1 =
  - ⟨ move(robot,loc2,loc1),
  - take(crane,loc1,cont,pallet,pile),
  - load(crane,loc1,cont,robot),
  - move(robot,loc1,loc2) ⟩
- $|\pi_1|$=4
- π_1 is a minimal, non-redundant solution



Overview

- The STRIPS Representation
- The Planning Domain Definition Language (PDDL)
- Problem-Solving by Search
- Heuristic Search
- Forward State-Space Search
- Backward State-Space Search
- The STRIPS Planner



PDDL Basics

- http://cs-www.cs.yale.edu/homes/dvm/
- language features (version 1.x):
  - basic STRIPS-style actions
  - various extensions as explicit requirements
- used to define:
  - planning domains: requirements, types, predicates, possible actions
  - planning problems: objects, rigid and fluent relations, initial situation, goal description

PDDL 1.x Domains

<domain> ::=
    (define (domain <name>)
    [<extension-def>]
    [<require-def>]
    [<types-def>]$^{\text{:typing}}$
    [<constants-def>]
    [<domain-vars-def>]$^{\text{:expression−evaluation}}$
    [<predicates-def>]
    [<timeless-def>]
    [<safety-def>]$^{\text{:safety−constraints}}$
    <structure-def>*)

<extension-def> ::=
    (:extends <domain name>+)
<require-def> ::=
    (:requirements <require-key>+)
<require-key> ::=
    :strips | :typing | …

<types-def> ::= (:types <typed list (name)>)
<constants-def> ::=
    (:constants <typed list (name)>)
<domain-vars-def> ::=
    (:domain-variables
    <typed list(domain-var-declaration)>)
<predicates-def> ::=
    (:predicates <atomic formula skeleton>+)
<atomic formula skeleton> ::=
    (<predicate> <typed list (variable)>)
<predicate> ::= <name>
<variable> ::= ?<name>
<timeless-def> ::=
    (:timeless <literal (name)>+)
<structure-def> ::= <action-def>
<structure-def> ::=$^{\text{:domain−axioms}}$ <axiom-def>
<structure-def> ::=$^{\text{:action−expansions}}$ <method-def>

PDDL Types

- PDDL types syntax

    <typed list (x)> ::= x*

    <typed list (x)> ::=$^{\text{:typing}}$
        x^{+} - <type> <typed list(x)>

    <type> ::= <name>

    <type> ::= (either <type>$^{+}$)

    <type> ::=$^{\text{:fluents}}$ (fluent <type>)

Example: DWR Types

```
(define (domain dock-worker-robot)

    (:requirements :strips :typing )

    (:types
        location     ;there are several connected locations
        pile         ;is attached to a location,
                     ;it holds a pallet and a stack of containers
        robot        ;holds at most 1 container,
                     ;only 1 robot per location
        crane        ;belongs to a location to pickup containers
        container )

…)
```

Example: DWR Predicates

```
(:predicates
    (adjacent ?l1 ?l2 - location)          ;location ?l1 is adjacent to ?l2
    (attached ?p - pile ?l - location)     ;pile ?p attached to location ?l
    (belong ?k - crane ?l - location)      ;crane ?k belongs to location ?l

    (at ?r - robot ?l - location)          ;robot ?r is at location ?l
    (occupied ?l - location)               ;there is a robot at location ?l
    (loaded ?r - robot ?c - container )    ;robot ?r is loaded with container ?c
    (unloaded ?r - robot)                  ;robot ?r is empty

    (holding ?k - crane ?c - container)    ;crane ?k is holding a container ?c
    (empty ?k - crane)                     ;crane ?k is empty

    (in ?c - container ?p - pile)          ;container ?c is within pile ?p
    (top ?c - container ?p - pile)         ;container ?c is on top of pile ?p
    (on ?c1 - container ?c2 - container)   ;container ?c1 is on container ?c2
)
```

PDDL Actions

<action-def> ::=
    (:action <action functor>
        :parameters (<typed list (variable)>)
        <action-def body>)
<action functor> ::= <name>
<action-def body> ::=
    [:vars (<typed list(variable)>)]$^{\text{:existential-preconditions :conditional-effects}}$
    [:precondition <GD>]
    [:expansion <action spec>]$^{\text{:action-expansions}}$
    [:expansion :methods]$^{\text{:action-expansions}}$
    [:maintain <GD>]$^{\text{:action-expansions}}$
    [:effect <effect>]
    [:only-in-expansions <boolean>]$^{\text{:action-expansions}}$

PDDL Goal Descriptions

<GD> ::= <atomic formula(term)>
<GD> ::= (and <GD>$^+$)
<GD> ::= <literal(term)>
<GD> ::=$^{\text{:disjunctive-preconditions}}$ (or <GD>$^+$)
<GD> ::=$^{\text{:disjunctive-preconditions}}$ (not <GD>)
<GD> ::=$^{\text{:disjunctive-preconditions}}$ (imply <GD> <GD>)
<GD> ::=$^{\text{:existential-preconditions}}$ (exists (<typed list(variable)>) <GD>)
<GD> ::=$^{\text{:universal-preconditions}}$ (forall (<typed list(variable)>) <GD>)
<literal(t)> ::= <atomic formula(t)>
<literal(t)> ::= (not <atomic formula(t)>)
<atomic formula(t)> ::= (<predicate> t*)
<term> ::= <name>

PDDL Effects

<effect> ::= (and <effect>$^{+}$)

<effect> ::= <atomic formula(term)>

<effect> ::= (not <atomic formula(term)>)

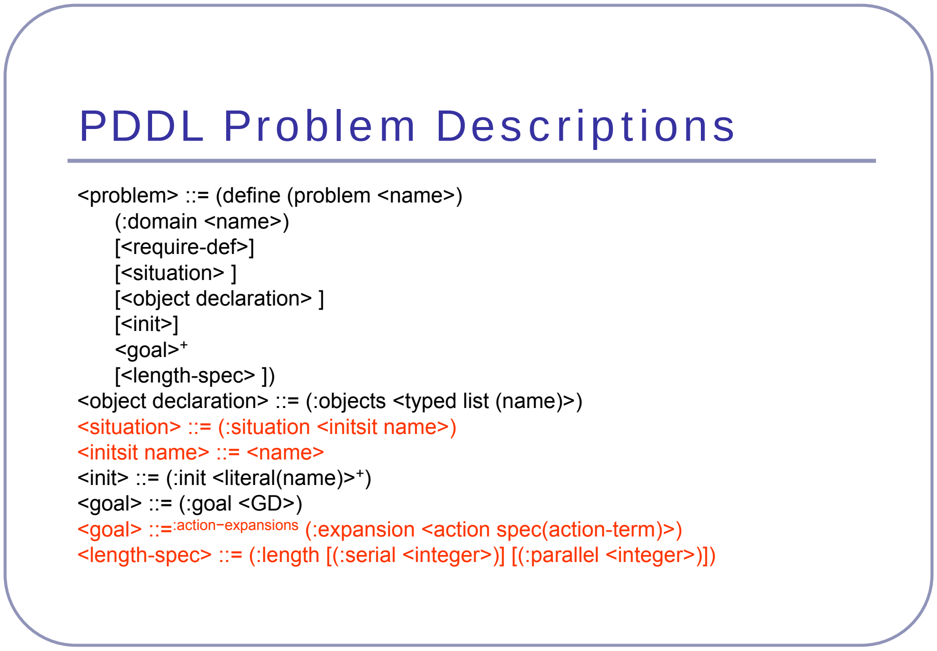
<effect> ::=$^{\text{:conditional−effects}}$
(forall (<variable>*) <effect>)

<effect> ::=$^{\text{:conditional−effects}}$
(when <GD> <effect>)

<effect> ::=$^{\text{:fluents}}$ (change <fluent> <expression>)

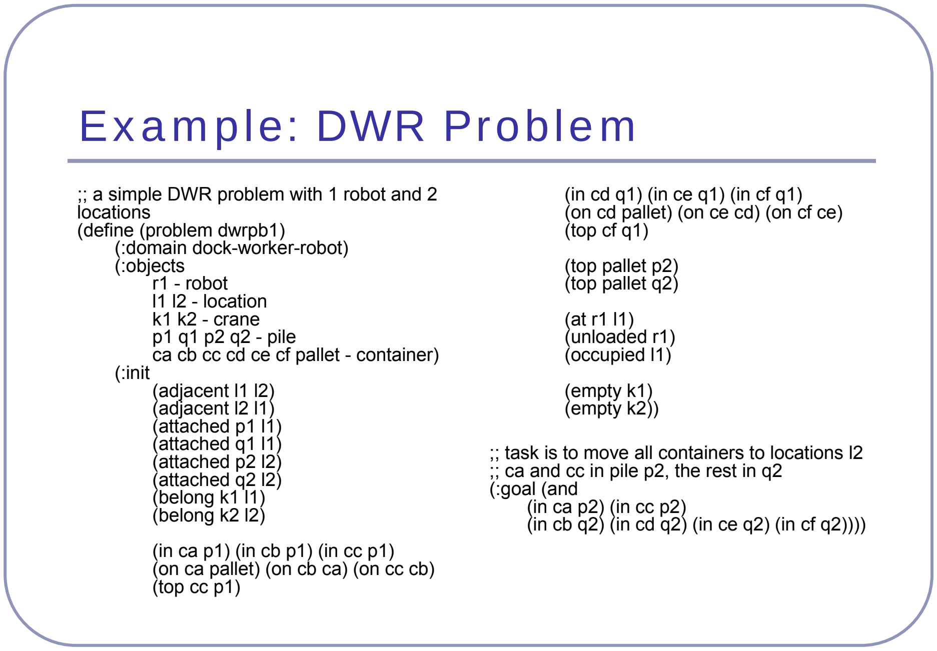
Example: DWR Action

```
;; moves a robot between two adjacent locations
(:action move
   :parameters (?r - robot ?from ?to - location)
   :precondition (and
      (adjacent ?from ?to) (at ?r ?from)
      (not (occupied ?to)))
   :effect (and
      (at ?r ?to) (occupied ?to)
      (not (occupied ?from)) (not (at ?r ?from)) ))
```



PDDL Problem Descriptions

<problem> ::= (define (problem <name>)
  (:domain <name>)
  [<require-def>]
  [<situation>]
  [<object declaration>]
  [<init>]
  <goal>$^+$
  [<length-spec>])
<object declaration> ::= (:objects <typed list (name)>)
<situation> ::= (:situation <initsit name>)
<initsit name> ::= <name>
<init> ::= (:init <literal(name)>$^+$)
<goal> ::= (:goal <GD>)
<goal> ::=$^{:action-expansions}$ (:expansion <action spec(action-term)>)
<length-spec> ::= (:length [(:serial <integer>)] [(:parallel <integer>)])



Example: DWR Problem

```
;; a simple DWR problem with 1 robot and 2
locations
(define (problem dwrpb1)
    (:domain dock-worker-robot)
    (:objects
        r1 - robot
        l1 l2 - location
        k1 k2 - crane
        p1 q1 p2 q2 - pile
        ca cb cc cd ce cf pallet - container)
    (:init
        (adjacent l1 l2)
        (adjacent l2 l1)
        (attached p1 l1)
        (attached q1 l1)
        (attached p2 l2)
        (attached q2 l2)
        (belong k1 l1)
        (belong k2 l2)

        (in ca p1) (in cb p1) (in cc p1)
        (on ca pallet) (on cb ca) (on cc cb)
        (top cc p1)

        (in cd q1) (in ce q1) (in cf q1)
        (on cd pallet) (on ce cd) (on cf ce)
        (top cf q1)

        (top pallet p2)
        (top pallet q2)

        (at r1 l1)
        (unloaded r1)
        (occupied l1)

        (empty k1)
        (empty k2))

    ;; task is to move all containers to locations l2
    ;; ca and cc in pile p2, the rest in q2
    (:goal (and
        (in ca p2) (in cc p2)
        (in cb q2) (in cd q2) (in ce q2) (in cf q2))))
```

Overview

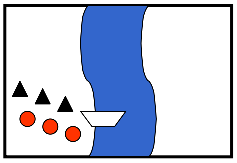
- The STRIPS Representation
- The Planning Domain Definition Language (PDDL)
- ➧ Problem-Solving by Search
- Heuristic Search
- Forward State-Space Search
- Backward State-Space Search
- The STRIPS Planner

Search Problems

- <u>initial state</u>
- set of possible <u>actions</u>/applicability conditions
  - successor function: *state* → set of <*action*, *state*>
  - successor function + initial state = state space
  - path (solution)
- <u>goal</u>
  - goal state or goal test function
- <u>path cost</u> function
  - for optimality
  - assumption: path cost = sum of step costs

Missionaries and Cannibals: Initial State and Actions

- initial state:
  - all missionaries, all cannibals, and the boat are on the left bank

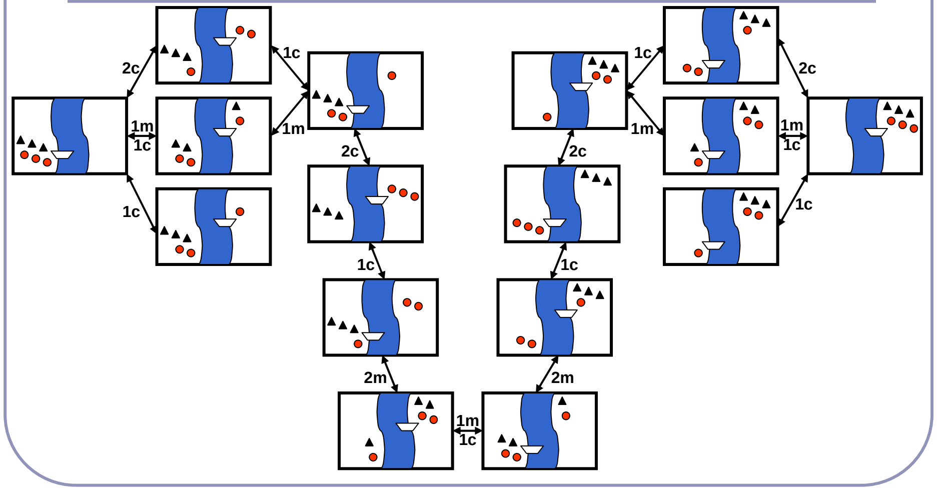


- 5 possible actions:
  - one missionary crossing
  - one cannibal crossing
  - two missionaries crossing
  - two cannibals crossing
  - one missionary and one cannibal crossing

Missionaries and Cannibals: Successor Function

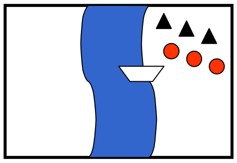
state	set of <*action*, *state*>
(L:3m,3c,b-R:0m,0c) →	{<2c, (L:3m,1c-R:0m,2c,b)>, <1m1c, (L:2m,2c-R:1m,1c,b)>, <1c, (L:3m,2c-R:0m,1c,b)>}
(L:3m,1c-R:0m,2c,b) →	{<2c, (L:3m,3c,b-R:0m,0c)>, <1c, (L:3m,2c,b-R:0m,1c)>}
(L:2m,2c-R:1m,1c,b) →	{<1m1c, (L:3m,3c,b-R:0m,0c)>, <1m, (L:3m,2c,b-R:0m,1c)>}
⋮	⋮

Missionaries and Cannibals: State Space



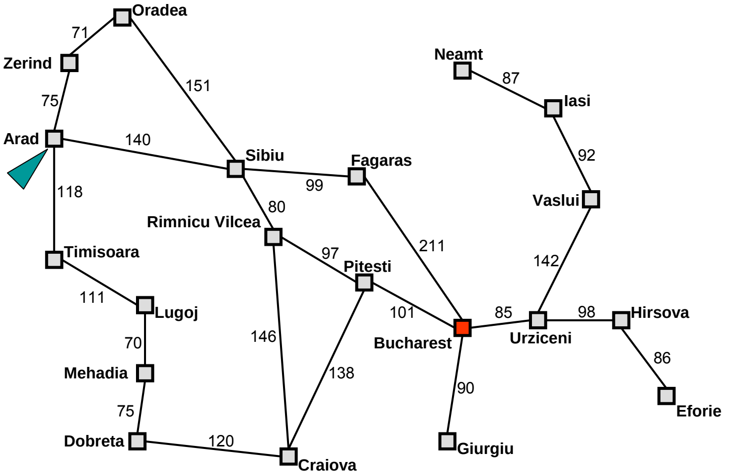
Missionaries and Cannibals: Goal State and Path Cost

- goal state:
  - all missionaries, all cannibals, and the boat are on the right bank



- path cost
  - step cost: 1 for each crossing
  - path cost: number of crossings = length of path
- solution path:
  - 4 optimal solutions
  - cost: 11

Real-World Problem: Touring in Romania

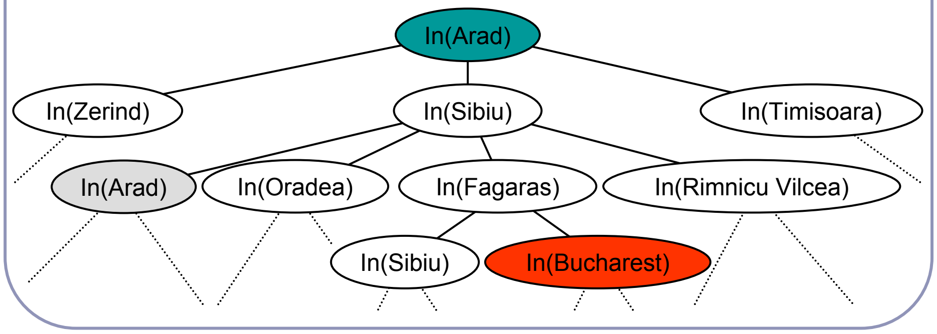


Touring Romania: Search Problem Definition

- initial state:
  - In(Arad)
- possible Actions:
  - DriveTo(Zerind), DriveTo(Sibiu), DriveTo(Timisoara), etc.
- goal state:
  - In(Bucharest)
- step cost:
  - distances between cities

Search Trees

- search tree: tree structure defined by initial state and successor function
- Touring Romania (partial search tree):



In(Arad)

In(Zerind)

In(Sibiu)

In(Timisoara)

In(Arad)

In(Oradea)

In(Fagaras)

In(Rimnicu Vilcea)

In(Sibiu)

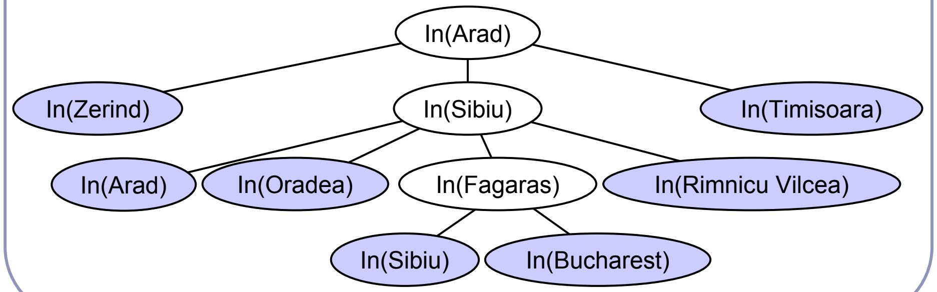
In(Bucharest)

Search Nodes

- search nodes: the nodes in the search tree
- data structure:
  - *state*: a state in the state space
  - *parent node*: the immediate predecessor in the search tree
  - *action*: the action that, performed in the parent node’s state, leads to this node’s state
  - *path cost*: the total cost of the path leading to this node
  - *depth*: the depth of this node in the search tree

Fringe Nodes in Touring Romania Example

fringe nodes: nodes that have not been expanded



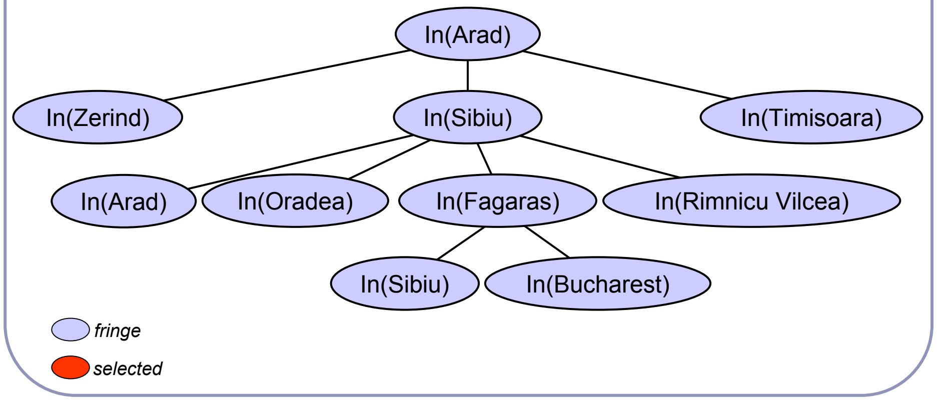
Search (Control) Strategy

- search or control strategy: an effective method for scheduling the application of the successor function to expand nodes
  - selects the next node to be expanded from the fringe
  - determines the order in which nodes are expanded
  - aim: produce a goal state as quickly as possible
- examples:
  - LIFO/FIFO-queue for fringe nodes
  - alphabetical ordering

General Tree Search Algorithm

```
function treeSearch(problem, strategy)
   fringe ← { new
         searchNode(problem.initialState) }
   loop
      if empty(fringe) then return failure
      node ← selectFrom(fringe, strategy)
      if problem.goalTest(node.state) then
         return pathTo(node)
      fringe ← fringe + expand(problem, node)
```

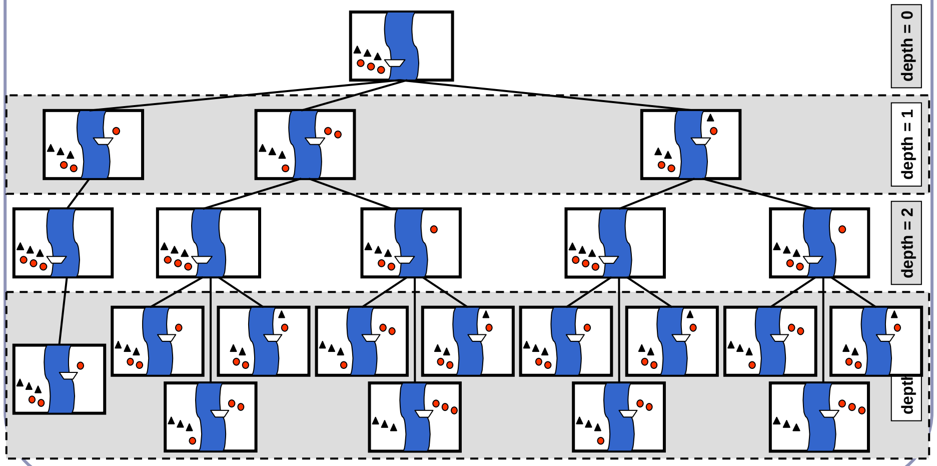
General Search Algorithm: Touring Romania Example



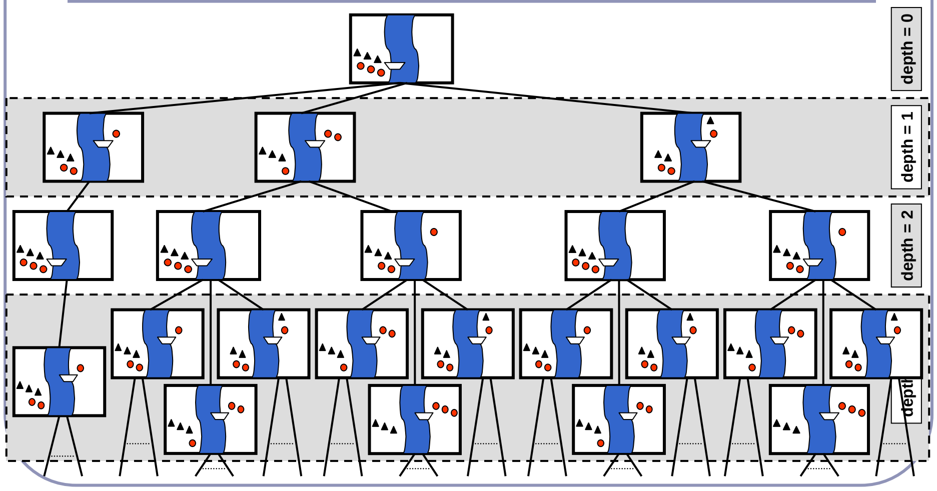
Uninformed vs. Informed Search

- uninformed search (blind search)
  - no additional information about states beyond problem definition
  - only goal states and non-goal states can be distinguished
- informed search (heuristic search)
  - additional information about how “promising” a state is available

Breadth-First Search: Missionaries and Cannibals



Depth-First Search: Missionaries and Cannibals



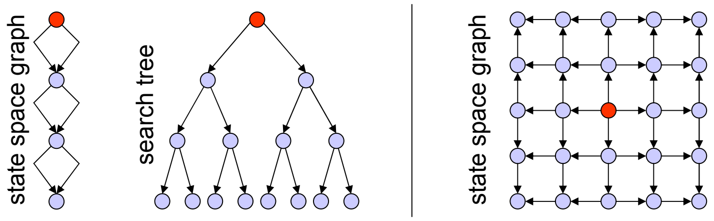
Iterative Deepening Search

- *strategy*:
  - based on depth-limited (depth-first) search
  - repeat search with gradually increasing depth limit until a goal state is found
- *implementation*:

```
for depth ← 0 to ∞ do
    result ← depthLimitedSearch(problem, depth)
    if result ≠ cutoff then return result
```

Discovering Repeated States: Potential Savings

- sometimes repeated states are unavoidable, resulting in infinite search trees
- checking for repeated states:
  - infinite search tree ⇒ finite search tree
  - finite search tree ⇒ exponential reduction



Overview

- The STRIPS Representation
- The Planning Domain Definition Language (PDDL)
- Problem-Solving by Search
- ➨ Heuristic Search
- Forward State-Space Search
- Backward State-Space Search
- The STRIPS Planner

Best-First Search

- an instance of the general tree search or graph search algorithm
  - strategy: select next node based on an evaluation function f: state space $\rightarrow \mathbb{R}$
  - select node with lowest value $f(n)$
- implementation:
  selectFrom(*fringe*, *strategy*)
  - priority queue: maintains fringe in ascending order of f-values

Heuristic Functions

- heuristic function h: state space $\rightarrow \mathbb{R}$
- $h(n)$ = estimated cost of the cheapest path from node n to a goal node
- if n is a goal node then $h(n)$ must be 0
- heuristic function encodes problem-specific knowledge in a problem-independent way

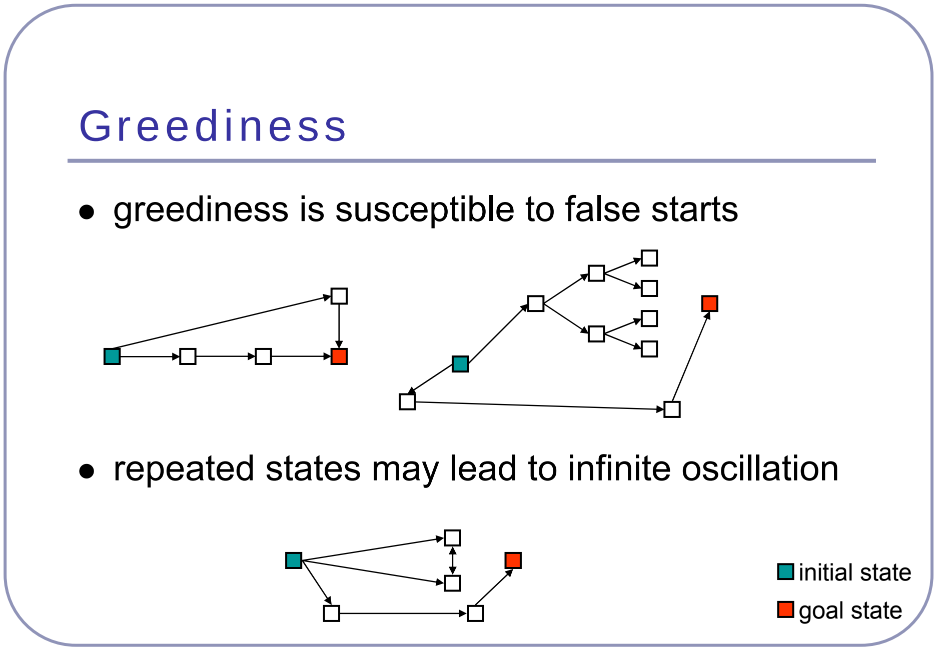
Greedy Best-First Search

- use heuristic function as evaluation function: $f(n) = h(n)$
  - always expands the node that is closest to the goal node
  - eats the largest chunk out of the remaining distance, hence, “greedy”

Touring in Romania: Heuristic

- $h_{SLD}(n)$ = straight-line distance to Bucharest

Arad	366	Hirsova	151	Rimnicu Vilcea	193
Bucharest	0	Iasi	226		
Craiova	160	Lugoj	244	Sibiu	253
Dobreta	242	Mehadia	241	Timisoara	329
Eforie	161	Neamt	234	Urziceni	80
Fagaras	176	Oradea	380	Vaslui	199
Giurgiu	77	Pitesti	100	Zerind	374

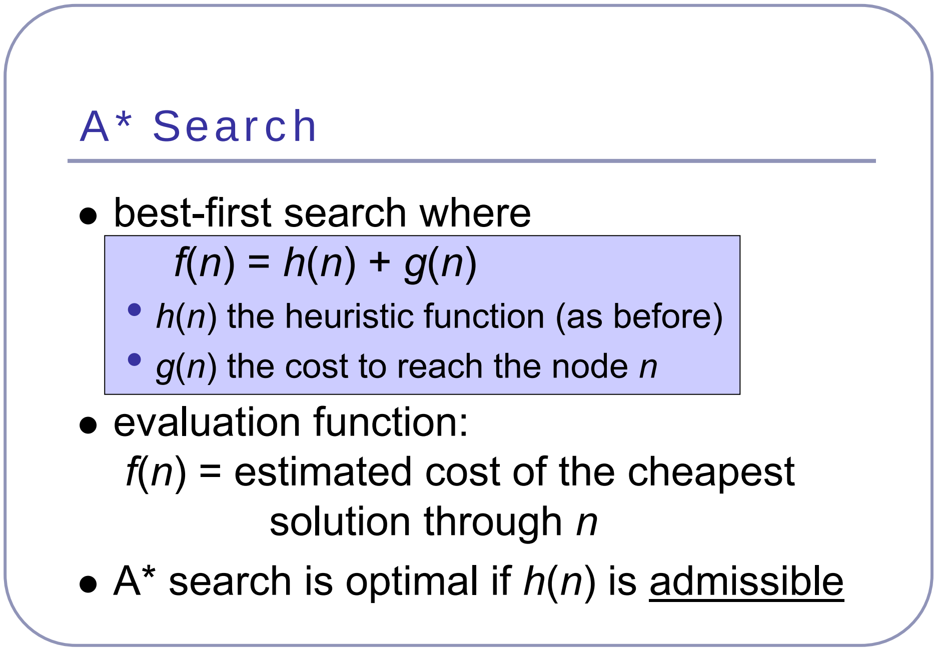


Greediness

- greediness is susceptible to false starts
- repeated states may lead to infinite oscillation

🟩 initial state

🟥 goal state



A* Search

- best-first search where

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

  - $h(n)$ the heuristic function (as before)
  - $g(n)$ the cost to reach the node n
- evaluation function:

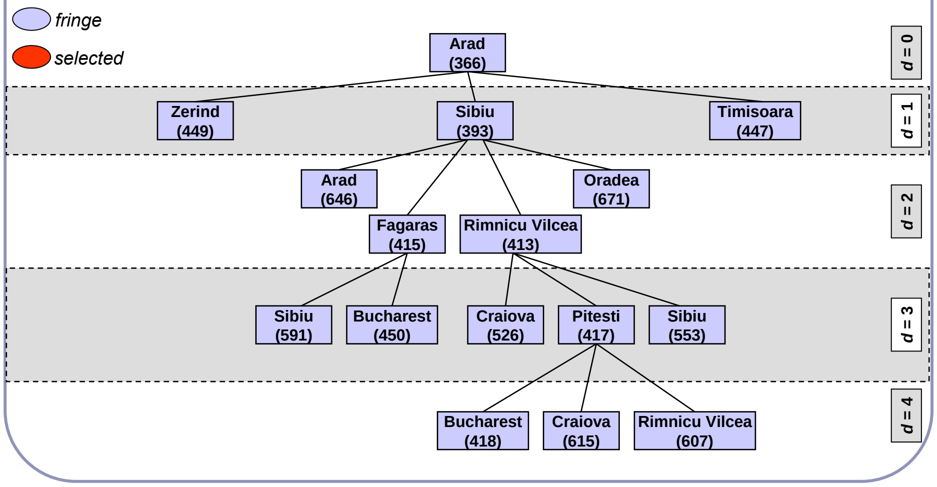
  $f(n)$ = estimated cost of the cheapest solution through n
- A* search is optimal if $h(n)$ is admissible

Admissible Heuristics

A heuristic $h(n)$ is admissible if it *never overestimates* the distance from n to the nearest goal node.

- example: h_{SLD}
- A* search: If $h(n)$ is admissible then $f(n)$ never overestimates the true cost of a solution through n.

A* (Best-First) Search: Touring Romania



Optimality of A* (Tree Search)

Theorem:

A* using tree search is optimal if the heuristic *h*(*n*) is admissible.

A*: Optimally Efficient

- A* is optimally efficient for a given heuristic function:
  no other optimal algorithm is guaranteed to expand fewer nodes than A*.
- any algorithm that does not expand all nodes with $f(n) < C^*$ runs the risk of missing the optimal solution

A* and Exponential Space

- A* has worst case time and space complexity of $O(b^l)$
- exponential growth of the fringe is normal
  - exponential time complexity may be acceptable
  - exponential space complexity will exhaust any computer’s resources all too quickly

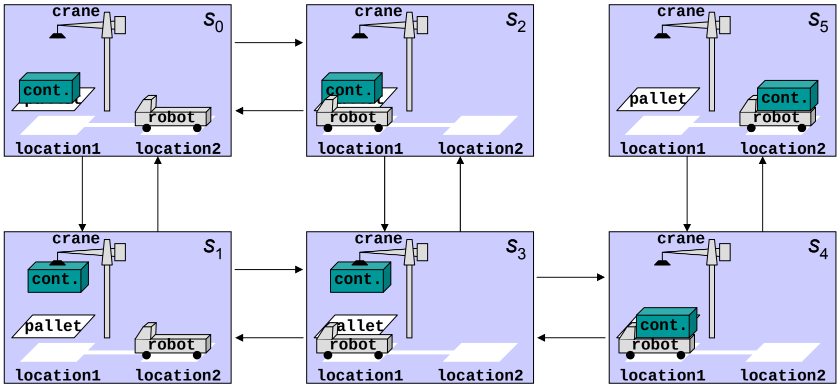
Overview

- The STRIPS Representation
- The Planning Domain Definition Language (PDDL)
- Problem-Solving by Search
- Heuristic Search
- ➡ Forward State-Space Search
- Backward State-Space Search
- The STRIPS Planner

State-Space Search

- idea: apply standard search algorithms (breadth-first, depth-first, A*, etc.) to planning problem:
  - search space is subset of state space
  - nodes correspond to world states
  - arcs correspond to state transitions
  - path in the search space corresponds to plan

DWR Example: State Space



Search Problems

- initial state
- set of possible actions/applicability conditions
  - successor function: *state* → set of <*action*, *state*>
  - successor function + initial state = state space
  - path (solution)
- goal
  - goal state or goal test function
- path cost function
  - for optimality
  - assumption: path cost = sum of step costs

State-Space Planning as a Search Problem

- given: statement of a planning problem $P=(O,s_i,g)$
- define the search problem as follows:
  - initial state: s_i
  - goal test for state s: s satisfies g
  - path cost function for plan π: $|\pi|$
  - successor function for state s: $\Gamma(s)$

Reachable Successor States

- The successor function $\Gamma^m{:}2^S \rightarrow 2^S$ for a STRIPS domain $\Sigma=(S,A,\gamma)$ is defined as:
  - $\Gamma(s)=\{\gamma(s,a) \mid a\in A \text{ and } a \text{ applicable in } s\}$ for $s\in S$

$$\left.\begin{array}{l} \bullet\ \Gamma(\{s_1,\dots,s_n\})= \cup_{(k\in[1,n])}\Gamma(s_k) \\ \bullet\ \Gamma^0(\{s_1,\dots,s_n\})= \{s_1,\dots,s_n\} \\ \bullet\ \Gamma^m(\{s_1,\dots,s_n\})= \Gamma(\Gamma^{m-1}(\{s_1,\dots,s_n\})) \end{array}\right\} \quad s_1,\dots,s_n\in S$$

- The transitive closure of Γ defines the set of all reachable states:
  - $\Gamma^>(s)= \cup_{(k\in[0,\infty])}\Gamma^k(\{s\})$ for $s\in S$

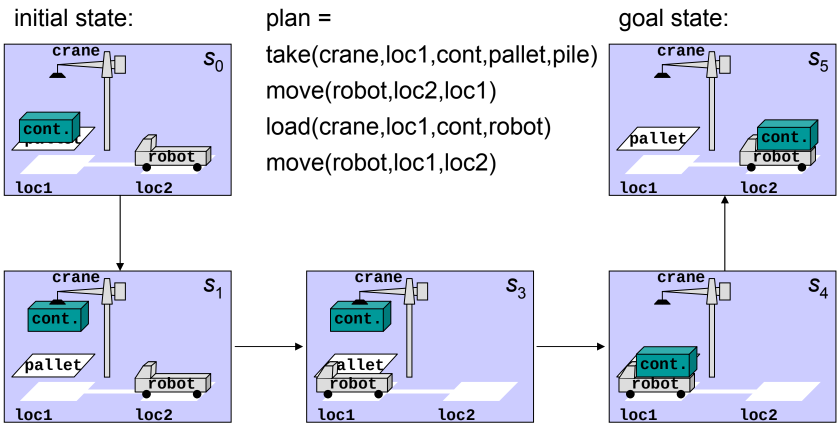
Solution Existence

- **Proposition**: A STRIPS planning problem $\mathcal{P}=(\Sigma,s_i,g)$ (and a statement of such a problem $P=(O,s_i,g)$) has a solution iff $S_g \cap \Gamma^>(\{s_i\}) \neq \{\}$.

Forward State-Space Search Algorithm

```
function fwdSearch($O$,$s_i$,$g$)
    state ← $s_i$
    plan ← ⟨⟩
    loop
        if state.satisfies($g$) then return plan
        applicables ←
            {ground instances from $O$ applicable in state}
        if applicables.isEmpty() then return failure
        action ← applicables.chooseOne()
        state ← $\gamma$(state,action)
        plan ← plan • ⟨action⟩
```

DWR Example: Forward Search



initial state:

plan =
take(crane,loc1,cont,pallet,pile)
move(robot,loc2,loc1)
load(crane,loc1,cont,robot)
move(robot,loc1,loc2)

goal state:

Finding Applicable Actions: Algorithm

```
function addApplicables(A, op, precs, σ, s)
    if $precs^+$.isEmpty() then
        for every np in $precs^-$ do
            if s.falsifies(σ(np)) then return
        A.add(σ(op))
    else
        pp ← $precs^+$.chooseOne()
        for every sp in s do
            σ' ← σ.extend(sp, pp)
            if σ'.isValid() then
                addApplicables(A, op, (precs - pp), σ', s)
```

Properties of Forward Search

- **Proposition**: fwdSearch is sound, i.e. if the function returns a plan as a solution then this plan is indeed a solution.
  - proof idea: show (by induction) *state*=γ(s_i,*plan*) at the beginning of each iteration of the loop

- **Proposition**: fwdSearch is complete, i.e. if there exists solution plan then there is an execution trace of the function that will return this solution plan.
  - proof idea: show (by induction) there is an execution trace for which *plan* is a prefix of the sought plan

Making Forward Search Deterministic

- idea: use depth-first search
  - problem: infinite branches
  - solution: prune repeated states
- pruning: cutting off search below certain nodes
  - safe pruning: guaranteed not to prune every solution
  - strongly safe pruning: guaranteed not to prune every optimal solution
  - example: prune below nodes that have a predecessor that is an equal state (no repeated states)

Overview

- The STRIPS Representation
- The Planning Domain Definition Language (PDDL)
- Problem-Solving by Search
- Heuristic Search
- Forward State-Space Search
- ➡ Backward State-Space Search
- The STRIPS Planner

The Problem with Forward Search

- number of actions applicable in any given state is usually very large
- branching factor is very large
- forward search for plans with more than a few steps not feasible

- idea: search backwards from the goal
- problem: many goal states

Relevance and Regression Sets

- Let $\mathcal{P}=(\Sigma,s_i,g)$ be a STRIPS planning problem. An action $a \in A$ is relevant for g if
  - $g \cap \text{effects}(a) \neq \{\}$ and
  - $g^+ \cap \text{effects}^-(a) = \{\}$ and $g^- \cap \text{effects}^+(a) = \{\}$.
- The regression set of g for a relevant action $a \in A$ is:
  - $\gamma^{-1}(g,a)=(g - \text{effects}(a)) \cup \text{precond}(a)$

Regression Function

- The regression function Γ^{-m} for a STRIPS domain $\Sigma=(S,A,\gamma)$ on L is defined as:
  - $\Gamma^{-1}(g)=\{\gamma^{-1}(g,a) \mid a\in A$ is relevant for $g\}$ for $g\in 2^L$
  - $\Gamma^{0}(\{g_1,\ldots,g_n\})= \{g_1,\ldots,g_n\}$
  - $\Gamma^{-1}(\{g_1,\ldots,g_n\})= \cup_{(k\in[1,n])}\Gamma^{-1}(g_k)$
  - $\Gamma^{-m}(\{g_1,\ldots,g_n\})= \Gamma^{-1}(\Gamma^{-(m-1)}(\{g_1,\ldots,g_n\}))$

  $\}$ $g_1,\ldots,g_n\in 2^L$
- The transitive closure of Γ^{-1} defines the set of all regression sets:
  - $\Gamma^{<}(g)= \cup_{(k\in[0,\infty])}\Gamma^{-k}(\{g\})$ for $g\in 2^L$

State-Space Planning as a Search Problem

- given: statement of a planning problem $P=(O,s_i,g)$
- define the search problem as follows:
  - initial search state: g
  - goal test for state s: s_i satisfies s
  - path cost function for plan π: $|\pi|$
  - successor function for state s: $\Gamma^{-1}(s)$

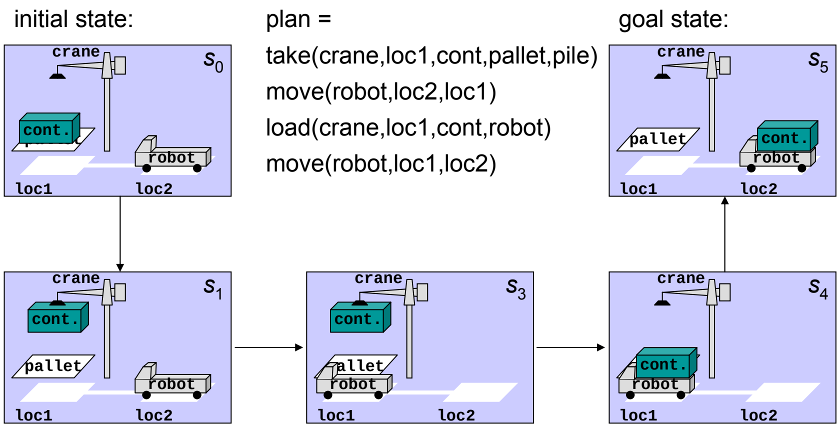
Solution Existence

- **Proposition**: A propositional planning problem $\mathcal{P}$=(Σ,s_i,g) (and a statement of such a problem P=(O,s_i,g)) has a solution iff $\exists s \in \Gamma^{<}(\{g\})$: s_i satisfies s.

Ground Backward State-Space Search Algorithm

```
function groundBwdSearch($O$,$s_i$,$g$)
    $subgoal \leftarrow g$
    plan $\leftarrow \langle\rangle$
    loop
        if $s_i$.satisfies($subgoal$) then return $plan$
        $applicables \leftarrow$
            {ground instances from $O$ relevant for $subgoal$}
        if $applicables$.isEmpty() then return failure
        $action \leftarrow applicables$.chooseOne()
        $subgoal \leftarrow \gamma^{-1}(subgoal, action)$
        $plan \leftarrow \langle action \rangle \bullet plan$
```

DWR Example: Backward Search



plan =
take(crane,loc1,cont,pallet,pile)
move(robot,loc2,loc1)
load(crane,loc1,cont,robot)
move(robot,loc1,loc2)

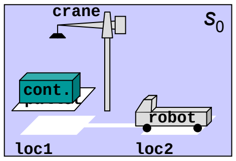
Example: Regression with Operators

- goal: at(robot,loc1)
- operator: move(*r*,*l*,*m*)
  - precond: adjacent(*l*,*m*), at(*r*,*l*), ¬occupied(*m*)
  - effects: at(*r*,*m*), occupied(*m*), ¬occupied(*l*), ¬at(*r*,*l*)
- actions: move(robot,*l*,loc1)
  - *l*=?
  - many options increase branching factor
- lifted backward search: use partially instantiated operators instead of actions

Lifted Backward State-Space Search Algorithm

```
function liftedBwdSearch($O$,$s_i$,$g$)
    $subgoal \leftarrow g$
    plan $\leftarrow \langle\rangle$
    loop
        if $\exists\sigma$:$s_i$.satisfies($\sigma$($subgoal$)) then return $\sigma$($plan$)
        $applicables \leftarrow$
            {($o$,$\sigma$) | $o \in O$ and $\sigma$($o$) relevant for $subgoal$}
        if $applicables$.isEmpty() then return failure
        $action \leftarrow applicables$.chooseOne()
        $subgoal \leftarrow \gamma^{-1}$($\sigma$($subgoal$), $\sigma$($o$))
        $plan \leftarrow \sigma$($\langle action\rangle$) • $\sigma$($plan$)
```

DWR Example: Lifted Backward Search



- initial state: s_0 = {attached(pile,loc1), in(cont,pile), top(cont,pile), on(cont,pallet), belong(crane,loc1), empty(crane), adjacent(loc1,loc2), adjacent(loc2,loc1), at(robot,loc2), occupied(loc2), unloaded(robot)}
- operator:move(r,l,m)
  - precond: adjacent(l,m), at(r,l), ¬occupied(m)
  - effects: at(r,m), occupied(m), ¬occupied(l), ¬at(r,l)

- liftedBwdSearch({move(r,l,m)}, s_0, {at(robot,loc1)})

- $\exists\sigma$:s_i.satisfies(σ($subgoal$)): no
- $applicables$ = {(move(r_1,l_1,m_1),{$r_1 \leftarrow$robot, $m_1 \leftarrow$loc1})}
- $subgoal$ = {adjacent(l_1,loc1), at(robot,l_1), ¬occupied(loc1)}
- $plan$ = ⟨move(robot,l_1,loc1)⟩

- $\exists\sigma$:s_i.satisfies(σ($subgoal$)): yes
  σ = {$l_1 \leftarrow$loc1}

Properties of Backward Search

- **Proposition**: liftedBwdSearch is sound, i.e. if the function returns a plan as a solution then this plan is indeed a solution.
  - proof idea: show (by induction) $\textit{subgaol}=\gamma^{-1}(g,\textit{plan})$ at the beginning of each iteration of the loop

- **Proposition**: liftedBwdSearch is complete, i.e. if there exists solution plan then there is an execution trace of the function that will return this solution plan.
  - proof idea: show (by induction) there is an execution trace for which *plan* is a suffix of the sought plan

Avoiding Repeated States

- search space:
  - let g_i and g_k be sub-goals where g_i is an ancestor of g_k in the search tree
  - let σ be a substitution such that $\sigma(g_i) \subseteq g_k$
- pruning:
  - then we can prune all nodes below g_k

Overview

- The STRIPS Representation
- The Planning Domain Definition Language (PDDL)
- Problem-Solving by Search
- Heuristic Search
- Forward State-Space Search
- Backward State-Space Search
- ➡ The STRIPS Planner

Problems with Backward Search

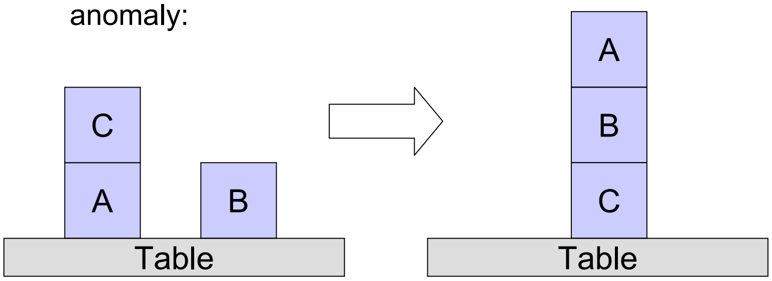
- state space still too large to search efficiently
- STRIPS idea:
  - only work on preconditions of the last operator added to the plan
  - if the current state satisfies all of an operator’s preconditions, commit to this operator

Ground-STRIPS Algorithm

```
function groundStrips(O,s,g)
    plan ← ⟨⟩
    loop
        if s.satisfies(g) then return plan
        applicables ←
            {ground instances from O relevant for g-s}
        if applicables.isEmpty() then return failure
        action ← applicables.chooseOne()
        subplan ← groundStrips(O,s,action.preconditions())
        if subplan = failure then return failure
        s ← $\gamma$(s, subplan • ⟨action⟩)
        plan ← plan • subplan • ⟨action⟩
```

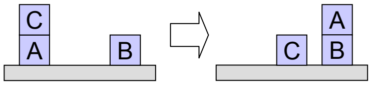
Problems with STRIPS

- STRIPS is incomplete:
  - cannot find solution for some problems, e.g. interchanging the values of two variables
  - cannot find optimal solution for others, e.g. Sussman anomaly:

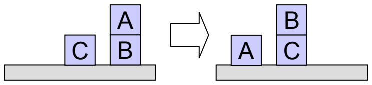


STRIPS and the Sussman Anomaly (1)

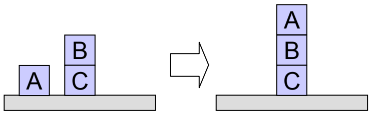
- achieve on(A,B)
  - put C from A onto table
  - put A onto B



- achieve on(B,C)
  - put A from B onto table
  - put B onto C

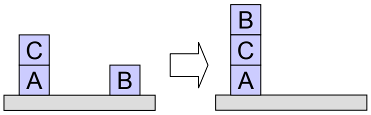


- re-achieve on(A,B)
  - put A onto B

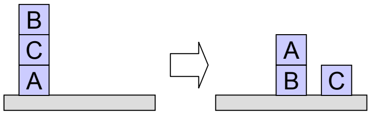


STRIPS and the Sussman Anomaly (2)

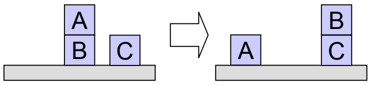
- achieve on(B,C)
  - put B onto C



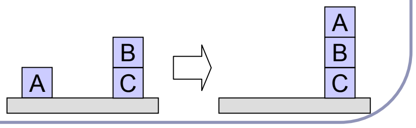
- achieve on(A,B)
  - put B from C onto table
  - put C from A onto table
  - put A onto B



- re-achieve on(B,C)
  - put A from B onto table
  - put B onto C



- re-achieve on(A,B)
  - put A onto B



Interleaving Plans for an Optimal Solution

- shortest solution achieving on(A,B):

put C from A onto table

put A onto B

- shortest solution achieving on(B,C):

put B onto C

- shortest solution for on(A,B) and on(B,C):

Overview

- The STRIPS Representation
- The Planning Domain Definition Language (PDDL)
- Problem-Solving by Search
- Heuristic Search
- Forward State-Space Search
- Backward State-Space Search
- The STRIPS Planner