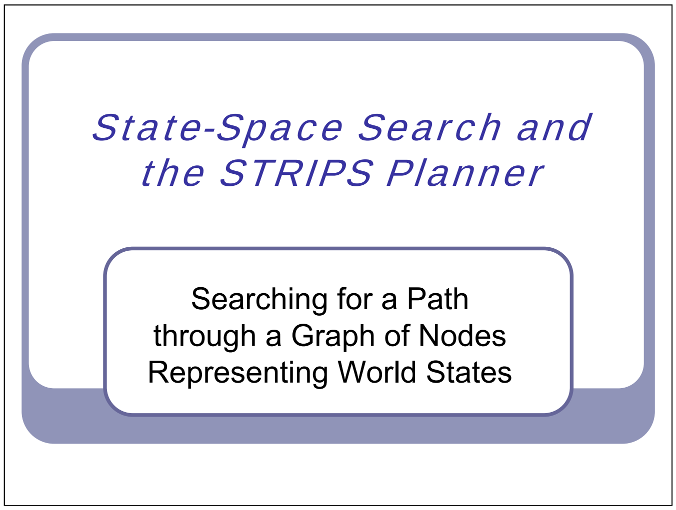
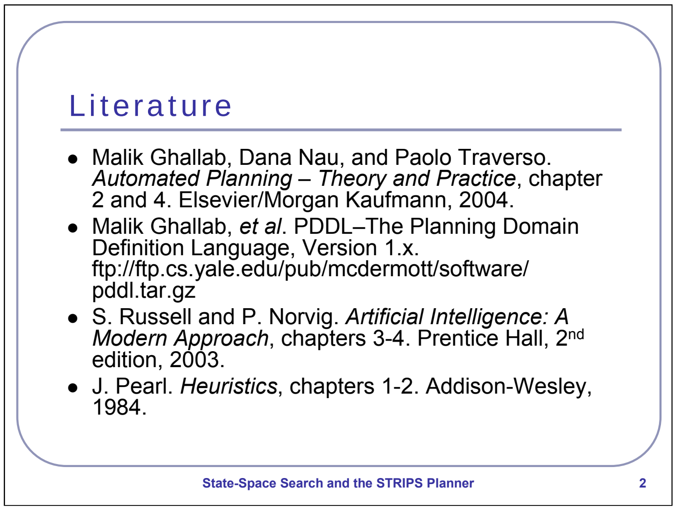


State-Space Search and the STRIPS Planner

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



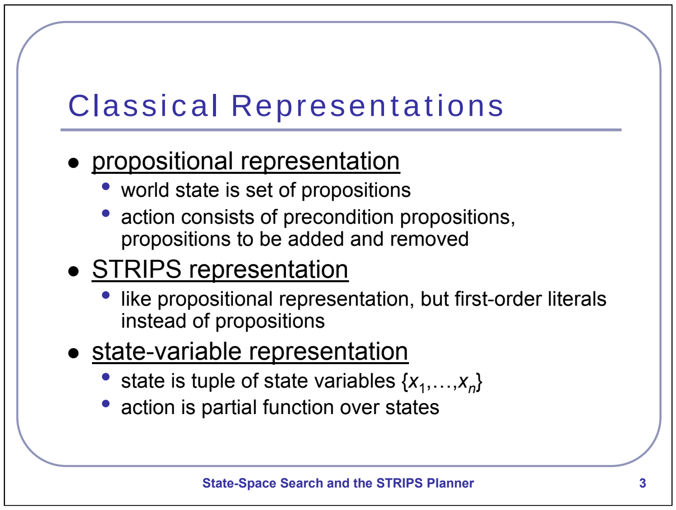
Literature

•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

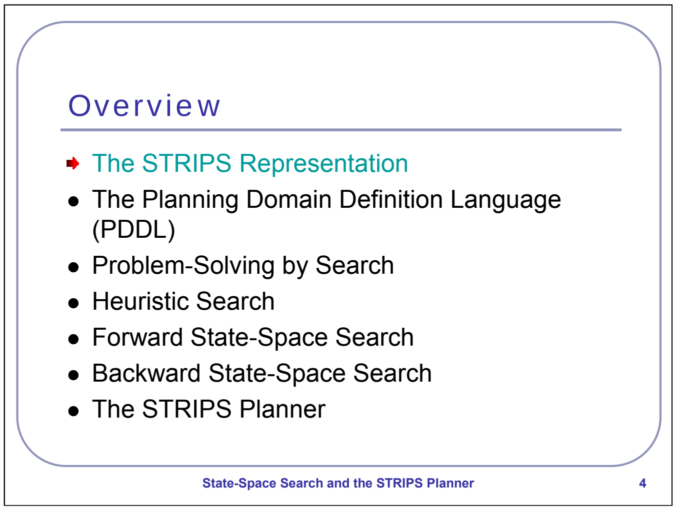
•S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*, chapters 3-4. Prentice Hall, 2nd edition, 2003.

•J. Pearl. *Heuristics*, chapters 1-2. Addison-Wesley, 1984.



Classical Representations

- **<u>propositional representation</u>**
  - **world state is set of propositions**
  - **action consists of precondition propositions, propositions to be added and removed**
- **<u>STRIPS representation</u>**
  - named after STRIPS planner
  - **like propositional representation, but first-order literals instead of propositions**
  - most popular for restricted state-transitions systems
- **<u>state-variable representation</u>**
  - **state is tuple of state variables $\{x_1,\ldots,x_n\}$**
  - **action is partial function over states**
  - useful where state is characterized by attributes over finite domains
- equally expressive: planning domain in one representation can also be represented in the others



Overview

➧The STRIPS Representation

- ➧now: the best-known knowledge representation formalism for reasoning about actions

•The Planning Domain Definition Language (PDDL)

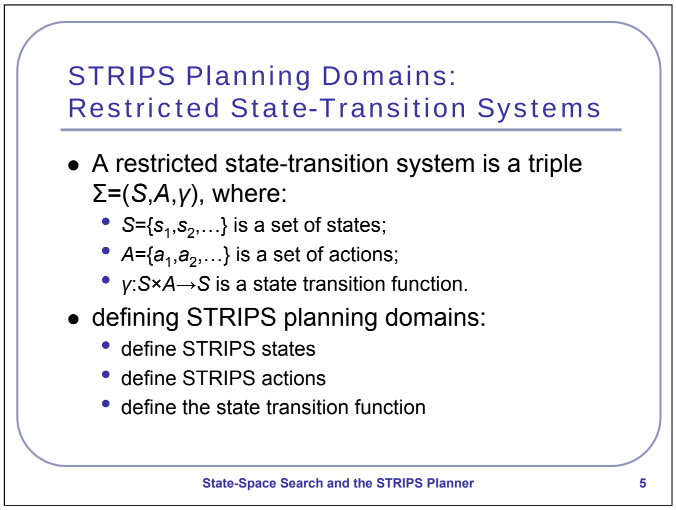
•Problem-Solving by Search

•Heuristic Search

•Forward State-Space Search

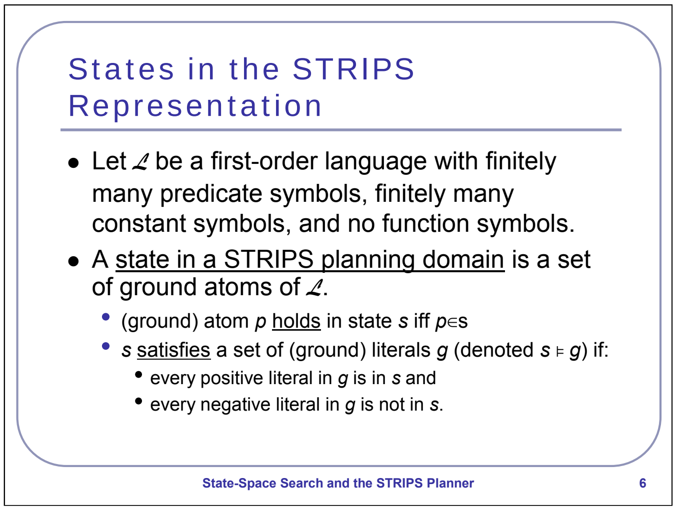
•Backward State-Space Search

•The STRIPS Planner



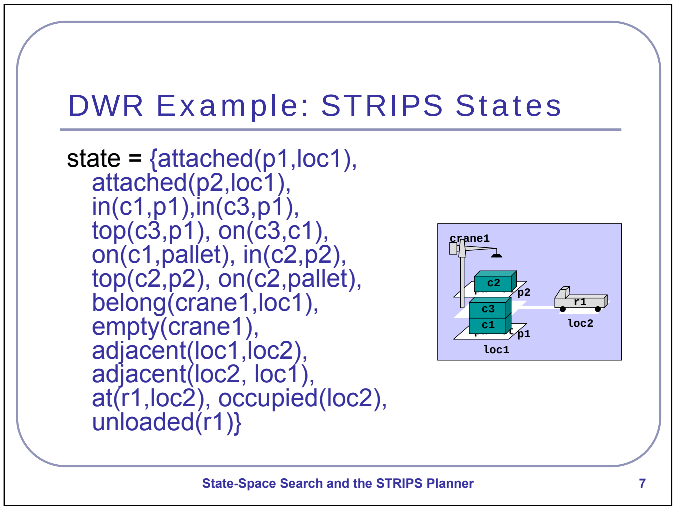
STRIPS Planning Domains: Restricted State-Transition Systems

- **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:**
    - to do to define the representation:
    - **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.**
  - terms in L are either constants or a variables
  - extensions of L will follow later
- **A <u>state in a STRIPS planning domain</u> is a set of ground atoms of $\mathcal{L}$.**
  - note: number of different states is finite
  - **(ground) atom *p* <u>holds</u> in state *s* iff $p \in s$**
    - closed-world assumption
  - *s* <u>satisfies</u> a set of (ground) literals *g* (denoted $s \models g$) if:
    - literals: atoms and negated atoms
    - every positive literal in *g* is in *s* and
    - every negative literal in *g* is not in *s*.
  - definitions for “holds” and “satisfies” may be generalized using substitutions

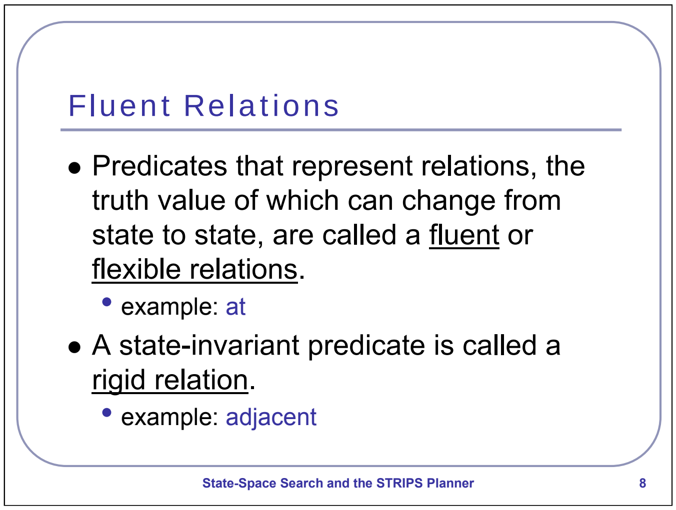


DWR Example: STRIPS States

•predicate symbols: relations for DWR domain

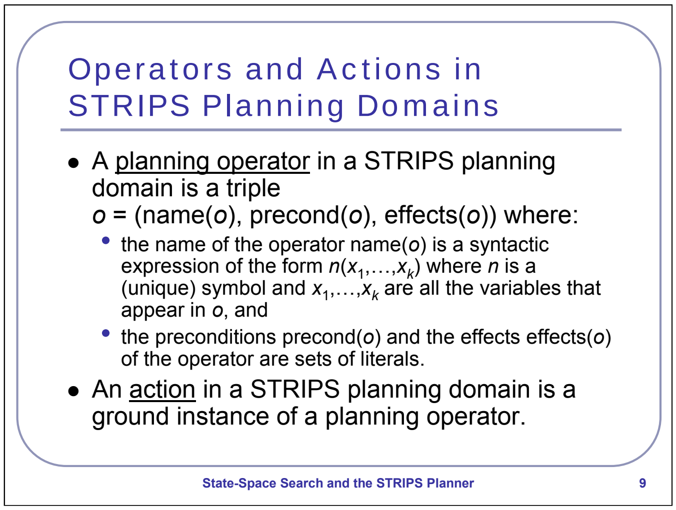
•constant symbols: for objects in the domain {loc1, loc2, r1, crane1, p1, p2, c1, c2, c3, pallet}

•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)}



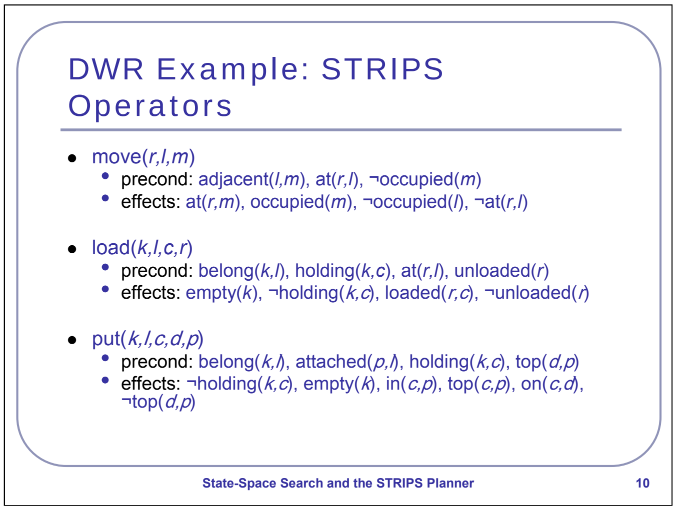
Fluent Relations

- note: whether an atom holds in a state may or may not depend on the state
- **Predicates that represent relations, the truth value of which can change from state to state, are called a fluent or flexible relations.**
  - **example: at**
    - changes when the robot moves
- **A state-invariant predicate is called a rigid relation.**
  - **example: adjacent**
    - cannot be changed by any of the actions in the domain
    - atoms involving this relation do not have a state or situation argument



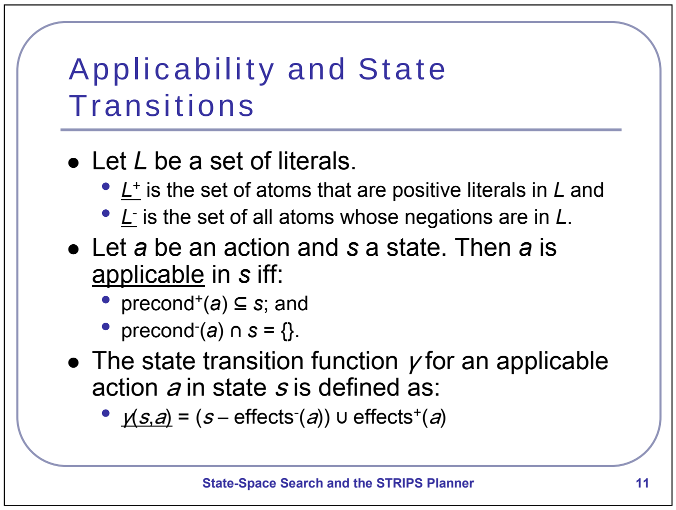
Operators and Actions in STRIPS Planning Domains

- **A <u>planning operator</u> 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**
    - unique: no two operators in the same domain must have the same name symbol
  - **the preconditions precond(*o*) and the effects effects(*o*) of the operator are sets of literals.**
    - only variables mentioned in the name are allowed to appear in these literals
- **An <u>action</u> in a STRIPS planning domain is a ground instance of a planning operator.**
  - actions are also called operator instances
- note: rigid relation must not appear in the effects of an operator, only in the preconditions



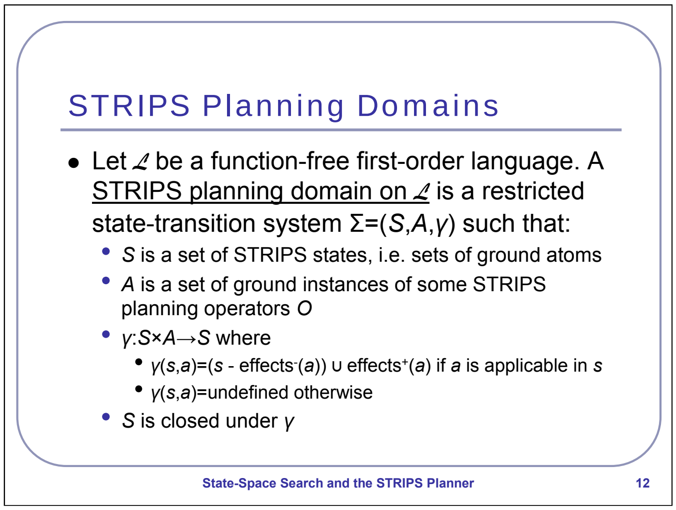
DWR Example: STRIPS Operators

- **move(*r,l,m*)**
  - robot *r* moves from location *l* to an adjacent location *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*)**
  - crane *k* at location *l* loads container *c* onto robot *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*)
  - crane *k* at location *l* puts container *c* onto *d* in pile *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*)
- similar: unload and take operators
- action: just substitute variables with values consistently



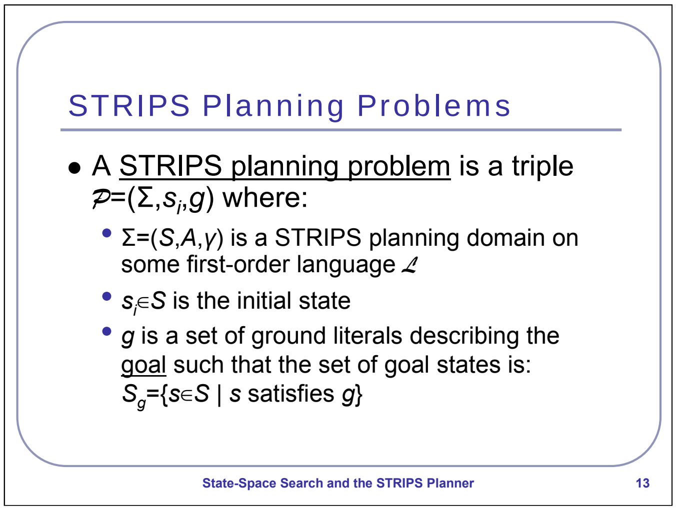
Applicability and State Transitions

- **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.**
    - specifically, for operators: $\text{precond}^+(a)$, $\text{precond}^-(a)$, $\text{effects}^+(a)$, and $\text{effects}^-(a)$ are defined in this way
- **Let a be an action and s a state. Then a is <u>applicable</u> in s iff:**
    - **$\text{precond}^+(a) \subseteq s$;** and
    - $\text{precond}^-(a) \cap s = \{\}$.
- The state transition function γ for an applicable action a in state s is defined as:
    - $\underline{\gamma(s,a)} = (s - \text{effects}^-(a)) \cup \text{effects}^+(a)$
        - note implicit frame axioms: what is not mentioned as an effect persists



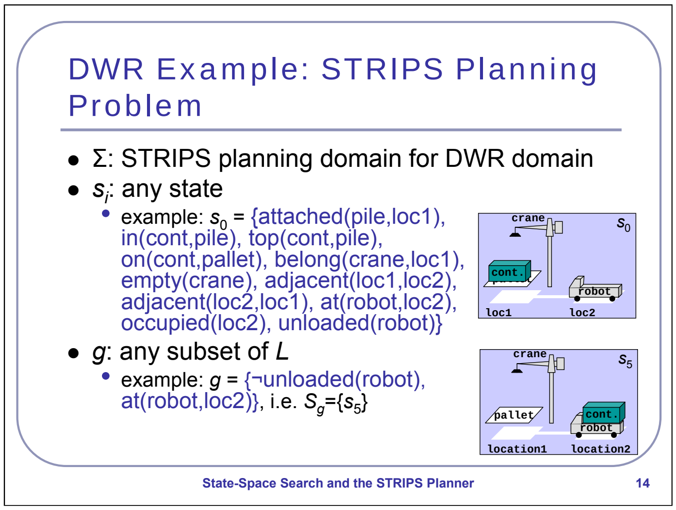
STRIPS Planning Domains

- **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
    - STRIPS vs. propositional domains: ground atoms instead of propositions
  - **A is a set of ground instances of some STRIPS planning operators O**
    - abstraction in operator descriptions due to variables; action effectively same as propositional actions
  - **$\gamma: S \times A \rightarrow S$ where**
    - **$\gamma(s,a)=(s - \mathrm{effects}^-(a)) \cup \mathrm{effects}^+(a)$ if a is applicable in s**
    - **$\gamma(s,a)$=undefined otherwise**
  - **S is closed under γ**



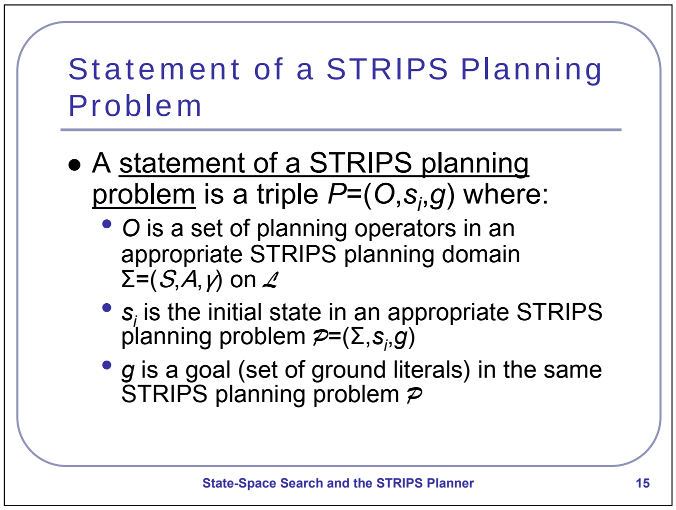
STRIPS Planning Problems

- **A <u>STRIPS planning problem</u> 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 <u>goal</u> such that the set of goal states is: $S_g=\{s \in S \mid s \text{ satisfies } g\}$**
    - note: g may contain positive and negated ground atoms (no closed world assumption for goals)



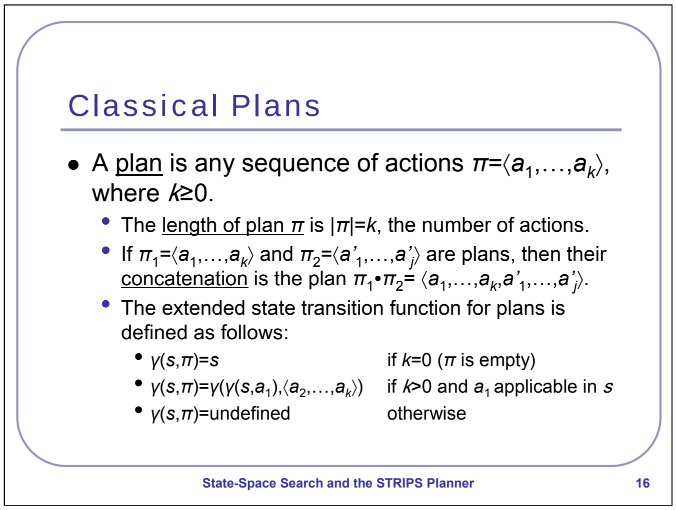
DWR Example: STRIPS Planning Problem

- **Σ: STRIPS planning domain for DWR domain**
  - see previous slides
- ***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)}**
  - note: s_0 is not necessarily initial state
- ***g*: any subset of *L***
  - **example: *g* = {¬unloaded(robot), at(robot,loc2)}, i.e. S_g={s_5}**
    - other relations will hold, but they are not mentioned in the goal = partial specification of a state



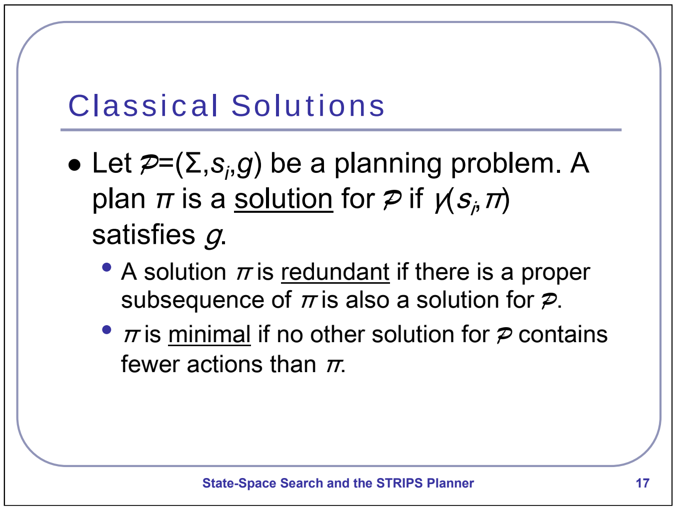
Statement of a STRIPS Planning Problem

- **A <u>statement of a STRIPS planning problem</u> is a triple $P=(O,s_i,g)$ where:**
  - **O is a set of planning operators in an appropriate STRIPS planning domain $\Sigma=(S,A,\gamma)$ on $\mathcal{L}$**
    - note: statement based on operators rather than actions = operator instances
  - **s_i is the initial state in an appropriate STRIPS planning problem $\mathcal{P}=(\Sigma,s_i,g)$**
  - ***g*** is a goal (set of ground literals) **in the same STRIPS planning problem $\mathcal{P}$**
- *statement is syntactic specification of STRIPS planning problem*
- *if two STRIPS planning problems have same statement, they will have same reachable states and solutions*



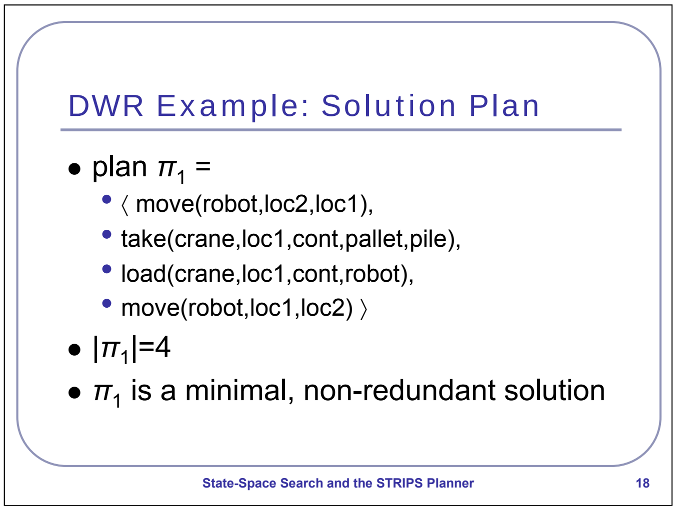
Classical Plans

- note: classical definitions apply to all representations
- **A plan is any sequence of actions $\pi=\langle a_1,\ldots,a_k\rangle$**, where $k\geq0$.
  - $k=0$ means no actions in the empty plan
  - 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**
- plan corresponds to a path through the state space



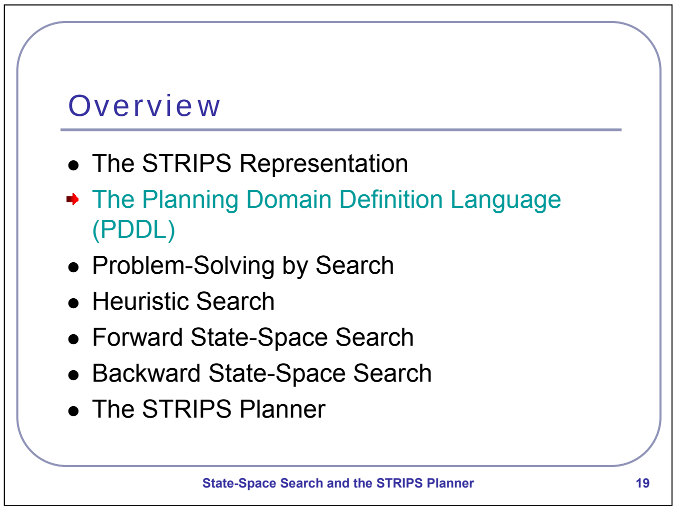
Classical Solutions

- **Let $\mathcal{P}=(\Sigma,s_i,g)$ be a propositional planning problem. A plan π is a <u>solution</u> for $\mathcal{P}$ if $g \subseteq \gamma(s_i,\pi)$.**
  - 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.
  - note: a minimal solution cannot be redundant
- solution is a path through the state space that leads from the initial state to a state that satisfies the goal



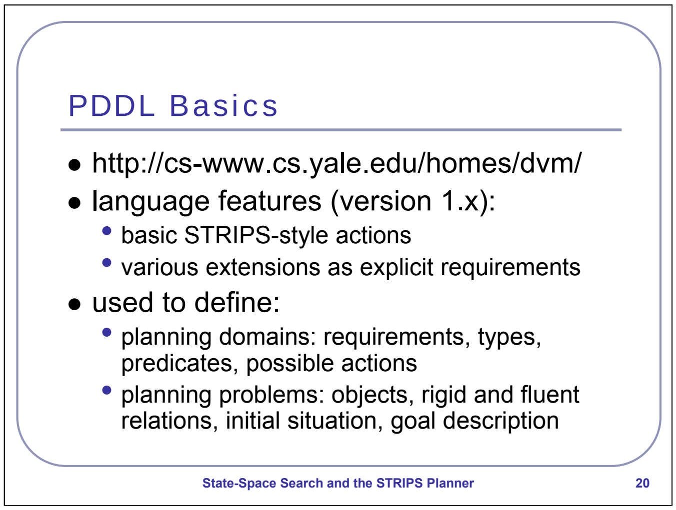
DWR Example: Solution Plan

- **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**
  - to the problem discussed previously



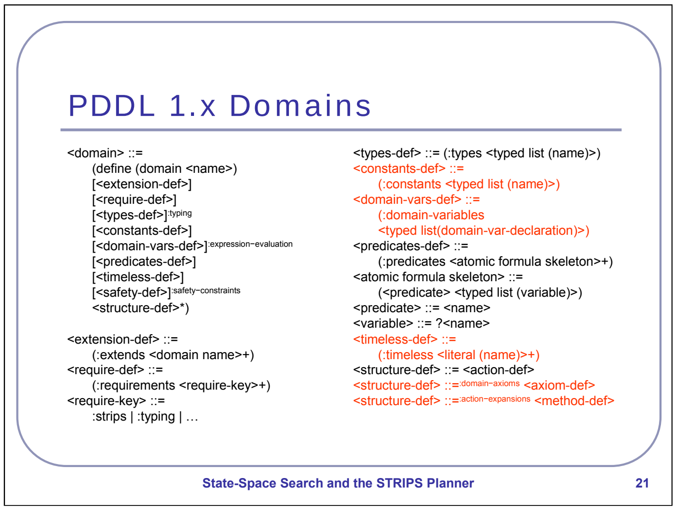
Overview

- **The STRIPS Representation**
  - just done: the best-known knowledge representation formalism for reasoning about actions
- **The Planning Domain Definition Language (PDDL)**
  - now: a syntax for the STRIPS representation (and extensions)
- **Problem-Solving by Search**
- **Heuristic Search**
- **Forward State-Space Search**
- **Backward State-Space Search**
- **The STRIPS Planner**



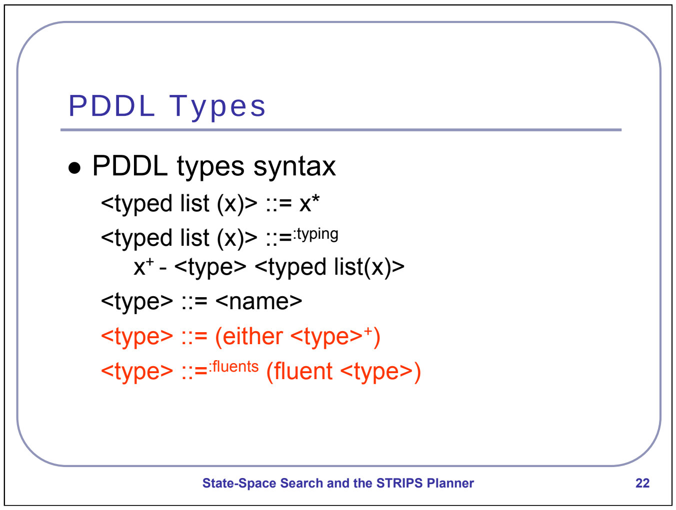
PDDL Basics

- **http://cs-www.cs.yale.edu/homes/dvm/**
  - Drew McDermott's home page; PDDL 1.7 available (contains documentation version 1.2)
  - developed for planning competition 1998; current version 3.0
- **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**



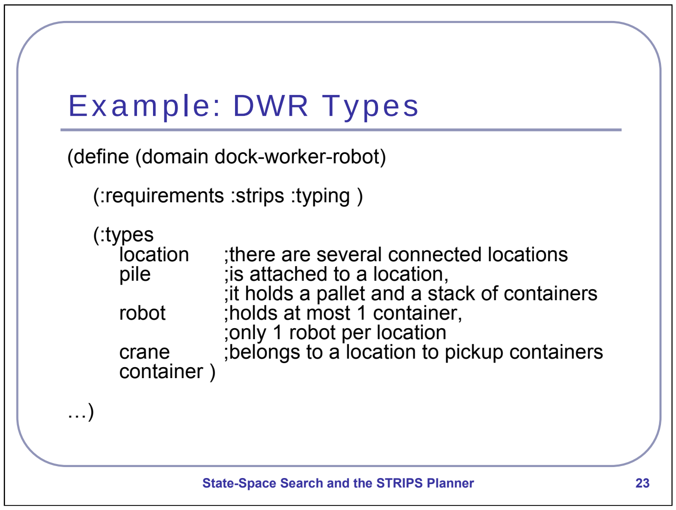
PDDL 1.x Domains

- **<domain> ::= (define (domain <name>)**
    - defines a (statement of a) planning domain
- **[<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>*)**
    - various optional components (in any order); only structure definitions (actions) required
- **<extension-def> ::= (:extends <domain name>+)**
    - possibility to “inherit” definitions from other domain
- **<require-def> ::= (:requirements <require-key>+)**
- **<require-key> ::= :strips | :typing | …**
    - language extensions required by the domain must be stated explicitly
- **<types-def> ::= (:types <typed list (name)>)**
    - allows for typing of objects and variables
- **<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>**
    - used to define domain relations for state descriptions; arguments may be typed
- **<timeless-def> ::= (:timeless <literal (name)>+)**
- **<structure-def> ::= <action-def>**
    - the basic STRIPS actions
- **<structure-def> ::=$^{\text{:domain-axioms}}$ <axiom-def>**
- **<structure-def> ::=$^{\text{:action-expansions}}$ <method-def>**



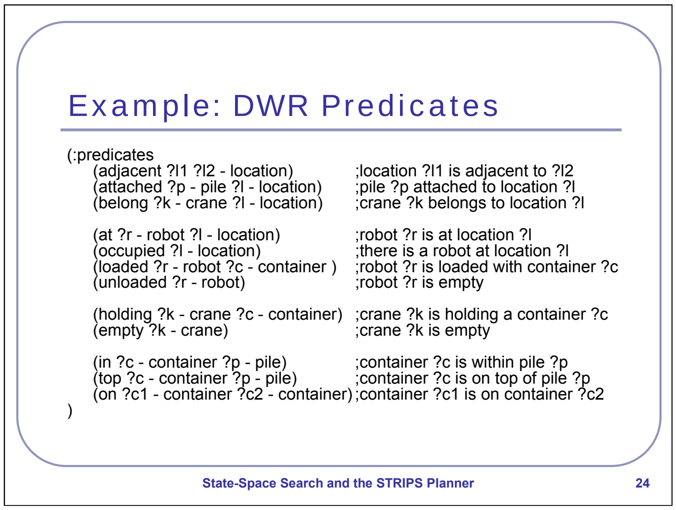
PDDL Types

- **PDDL types syntax**
  - **<typed list (x)> ::= x***
    - untyped version is always part of the syntax
  - **<typed list (x)> ::=$^{:typing}$ x^+ - <type> <typed list(x)>**
    - multiple objects can be declared to have the same type
    - last element for recursion
  - **<type> ::= <name>**
  - **<type> ::= (either <type>$^+$)**
  - **<type> ::=$^{:fluents}$ (fluent <type>)**



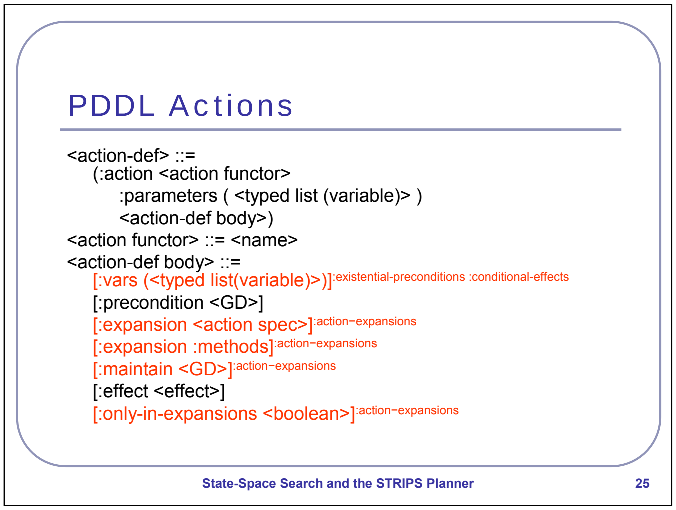
Example: DWR Types

- **(define (domain dock-worker-robot)**
  - defines a named domain (running example)
  - **(:requirements :strips :typing)**
    - simple requirements: STRIPS actions and typing (to make domain more readable)
  - **(:types**
    - **location ;there are several connected locations**
      - first type: set of objects that belong to this type
      - note: semicolon is beginning of comment
    - **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)**
- **…)**
  - remaining domain omitted here



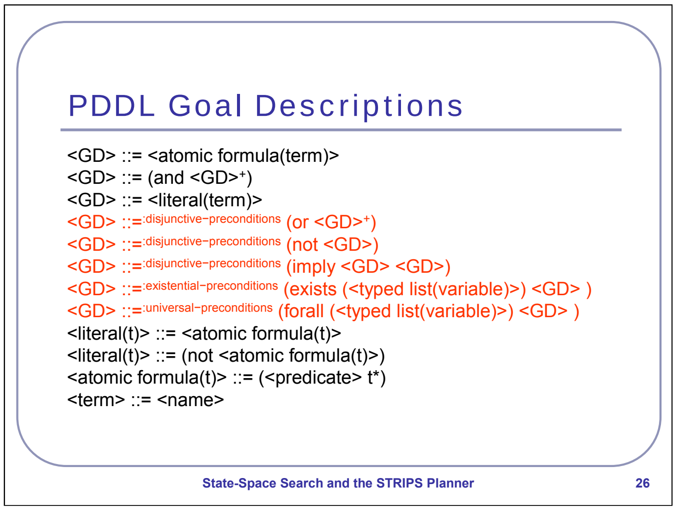
Example: DWR Predicates

- **(:predicates**
  - **(adjacent ?l1 ?l2 - location) ;location ?l1 is adjacent to ?l2**
    - predicate name: adjacent
    - two arguments represented by variables: ?l1 and ?l2
    - type of both variables must be location
  - **(attached ?p - pile ?l - location) ;pile ?p attached to location ?l**
    - arguments of two different types
  - **(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**
    - always use comments!
- **)**



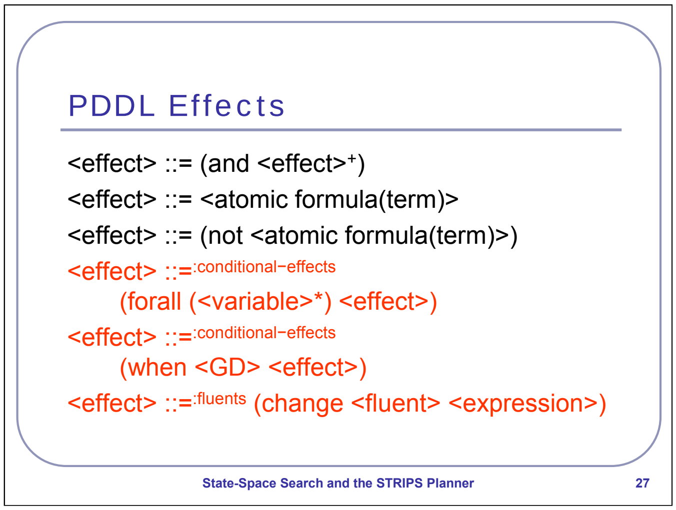
PDDL Actions

- **<action-def> ::= (:action <action functor>**
  - **:parameters (<typed list (variable)>)**
    - list of variables representing parameters
    - typed for readability and reduced search space size
  - **<action-def body>)**
- **<action functor> ::= <name>**
- **<action-def body> ::= [:vars (<typed list(variable)>)][:existential-preconditions :conditional-effects] [:precondition <GD>] [:expansion <action spec>][:action−expansions] [:expansion :methods][:action−expansions] [:maintain <GD>][:action−expansions] [:effect <effect>] [:only-in-expansions <boolean>][:action−expansions]**
  - preconditions: GD = goal description; sub-goal for making this action applicable



PDDL Goal Descriptions

- **<GD> ::= <atomic formula(term)>**
  - simples case: positive or negative atom (predicate with arguments)
- **<GD> ::= (and <GD>$^+$)**
  - conjunction made explicit
- **<GD> ::= <literal(term)>**
- **<GD> ::=:disjunctive−preconditions (or <GD>$^+$)**
- **<GD> ::=:disjunctive−preconditions (not <GD>)**
- **<GD> ::=:disjunctive−preconditions (imply <GD> <GD>)**
- **<GD> ::=:existential−preconditions (exists (<typed list(variable)>) <GD>)**
- **<GD> ::=: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

- note: for basic STRIPS representation, goals and effects are syntactically identical

•<effect> ::= (and <effect>$^+$)

- again, conjunction is explicit (but no disjunctive extension)

•<effect> ::= <atomic formula(term)>

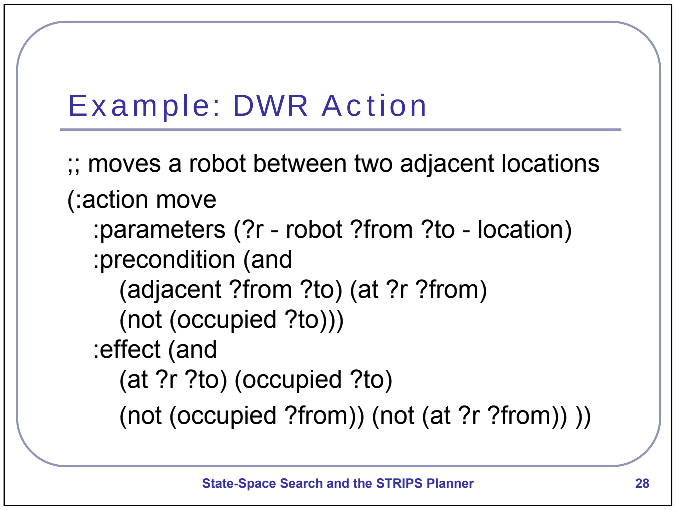
•<effect> ::= (not <atomic formula(term)>)

- positive and negative literals

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

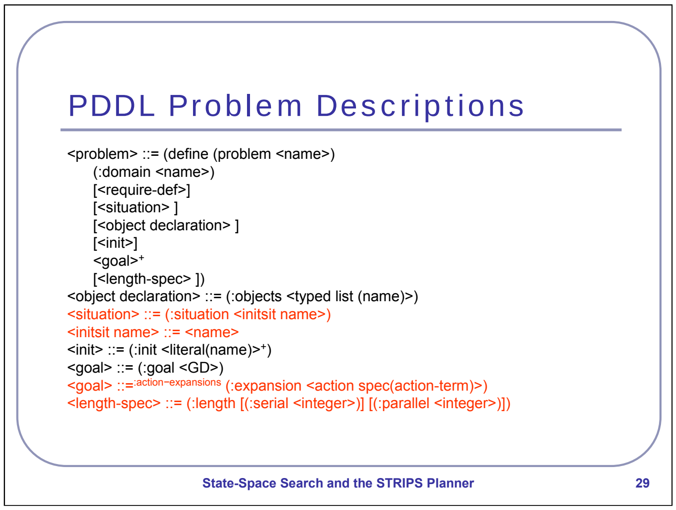
•<effect> ::=$^{\text{:conditional-effects}}$ (when <GD> <effect>)

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



Example: DWR Action

- **;; moves a robot between two adjacent locations**
  - Lisp convention: double semicolon not strictly necessary
- **(:action move**
  - **:parameters (?r - robot ?from ?to - location)**
    - typed parameters: “?r” of type robot and “?from” and “?to” of type location
  - **:precondition (and**
    - conjunction
    - **(adjacent ?from ?to) (at ?r ?from)**
    - **(not (occupied ?to)))**
  - **:effect (and**
    - **(at ?r ?to) (occupied ?to)**
    - **(not (occupied ?from)) (not (at ?r ?from))))**
  - note: common to find negated fluent preconditions as effects, but not always



PDDL Problem Descriptions

•<problem> ::= (define (problem <name>)

- **(:domain <name>)**

  •problem must be defined wrt. a domain, i.e. a set of action definitions

- **[<require-def>] [<situation>] [<object declaration>] [<init>]**

  •situation vs. init: used named situation (re-usable) or define initial state explicitly

- **<goal>$^{+}$**

  •at least one goal description

- **[<length-spec>])**

•<object declaration> ::= (:objects <typed list (name)>)

  •list of (typed) objects that exist in this problem (logically: constant terms)

•<situation> ::= (:situation <initsit name>)

•<initsit name> ::= <name>

  •named situation

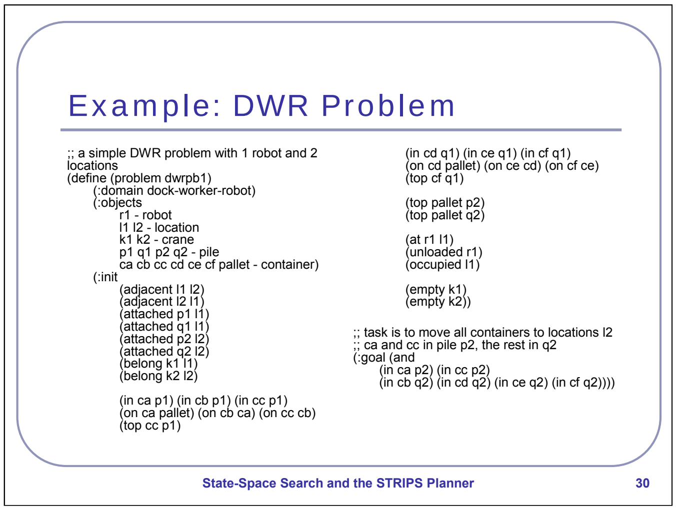
•<init> ::= (:init <literal(name)>$^{+}$)

  •list of literals (note: includes negative literals)

•<goal> ::= (:goal <GD>)

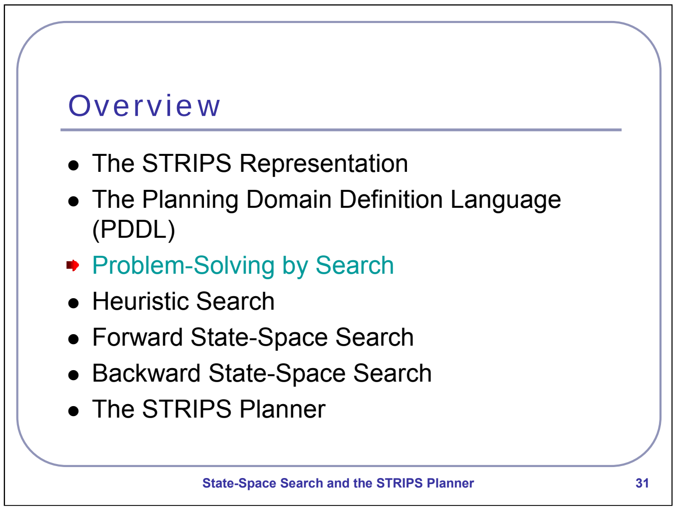
•<goal> ::=$^{\text{:action−expansions}}$ (:expansion <action spec(action-term)>)

•<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)**
      - rigid relations
    - **(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)**
      - the two piles of containers at location l1
    - **(top pallet p2)**
    - **(top pallet q2)**
      - no containers at location l2
    - **(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))))**
  - note: many solutions as order of containers is undefined



Overview

➧The STRIPS Representation

•The Planning Domain Definition Language (PDDL)

•just done : a syntax for the STRIPS representation (and extensions)

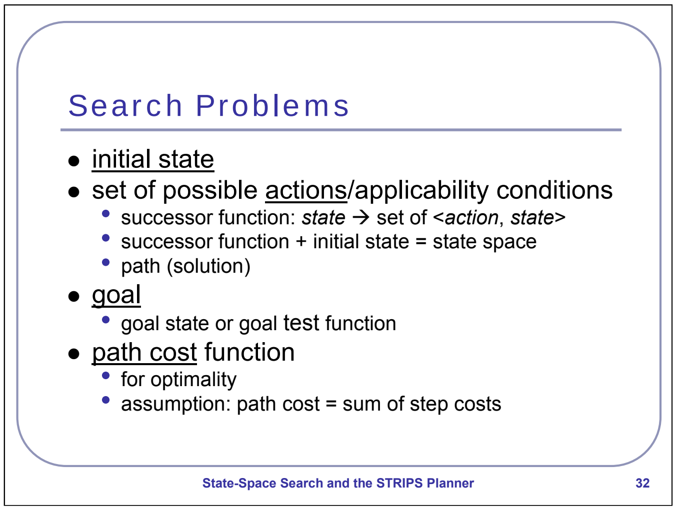
•Problem-Solving by Search

•Heuristic Search

•Forward State-Space Search

•Backward State-Space Search

•The STRIPS Planner



Search Problems

•**initial state**: current state the world is in (state = situation)

- states: symbol structures representing real world objects and relations → physical symbols systems

•finite **set of possible actions** (aka. operators or production rules (problem formulation))**/applicability conditions**

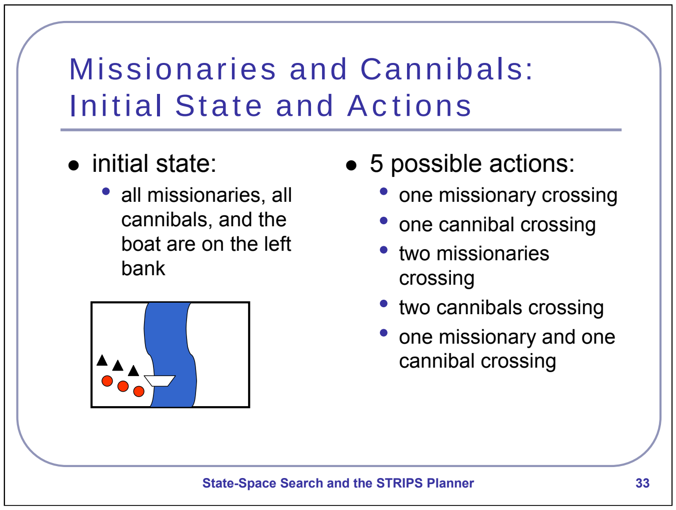
- **successor function: *state* → set of <*action*, *state*>**: action is applicable in given state; result of applying action in given state is paired state
- **successor function + initial state = state space**: directed graph with states as nodes and actions as arcs
- **path** (in the graph) **(solution)**

•**goal** (goal formulation)

- **goal state** (for unique goal state) **or goal test function** (for multiple goal states (e.g. in chess))
  - Solution: path in state space from initial state to goal state

•**path cost function**

- **for optimality**: find solution path with minimal path cost
- **assumption: path cost = sum of step costs** (cost of applying a given action in a given state)



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**
  - note: not every action applicable in every state
    - example: first action not applicable in initial state

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: Successor Function

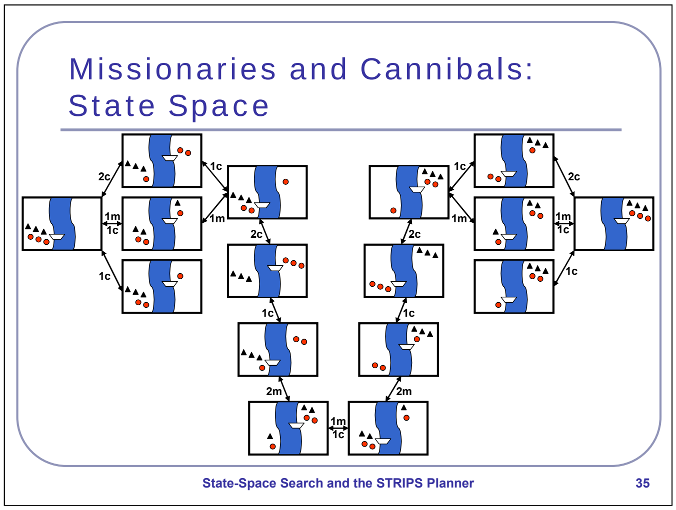
•***state* → set of <*action*, *state*>** (domain and range: set of pairs)

•**(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)>}**

- •states:
  - •L/R: on left/right bank
  - •m/c: missionaries/cannibals (example: 3 missionaries and three cannibals on left bank, none on right bank)
  - •b: boat (example: boat on left bank)
- •actions:
  - •m/c: missionaries/cannibals crossing (example(s): 2 cannibals crossing (L to R), 1m and 1c crossing; 1c crossing)

•**(L:3m,1c-R:0m,2c,b) → {<2c, (L:3m,3c,b-R:0m,0c)>, <1c, (L:3m,2c,b-R:0m,1c)>}** (note: only two actions applicable)

•**(L:2m,2c-R:1m,1c,b) → {<1m1c, (L:3m,3c,b-R:0m,0c)>, <1m, (L:3m,2c,b-R:0m,1c)>}**

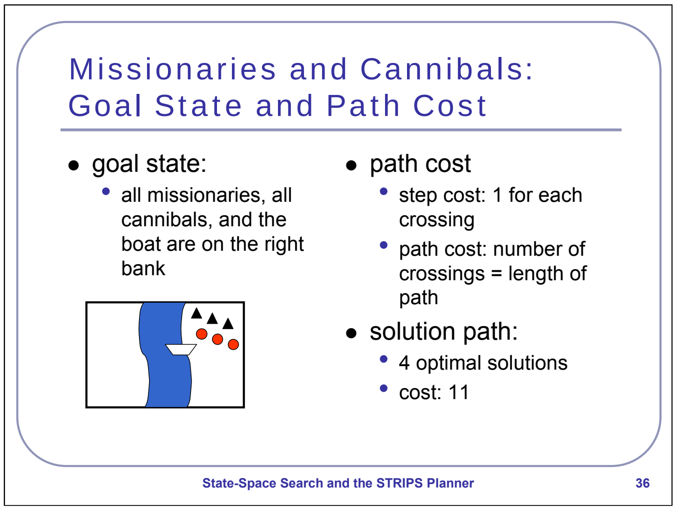


Missionaries and Cannibals: State Space

•(only) 16 possible world states

•arcs represent possible actions with action as label

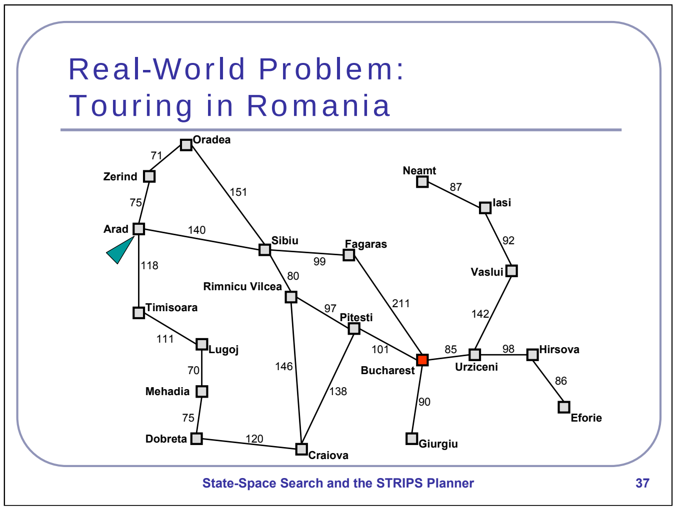
  •actions reversible and reversing action is same action; hence bidirectional arcs



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** (alternatives weigh missionaries and cannibals crossing differently)
  - **path cost: number of crossings = length of path**
- **solution path:**
  - **4 optimal solutions**
  - **cost: 11**

- search problem now complete: initial state, actions (successor function), goal state, and path cost function

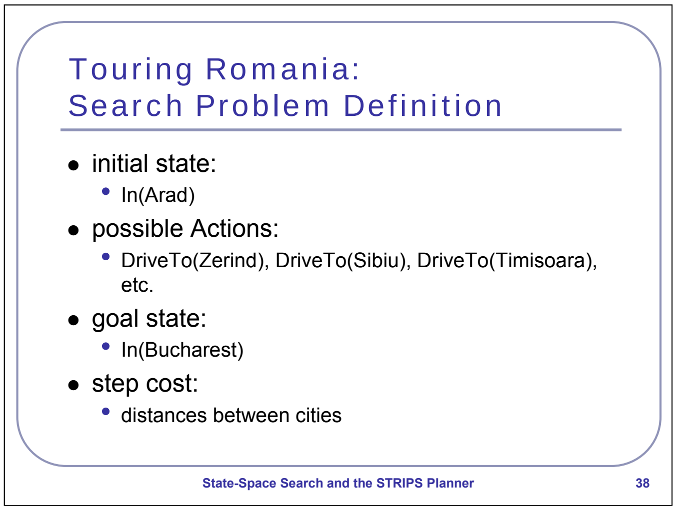


Real-World Problem: Touring in Romania

•shown: rough map of Romania

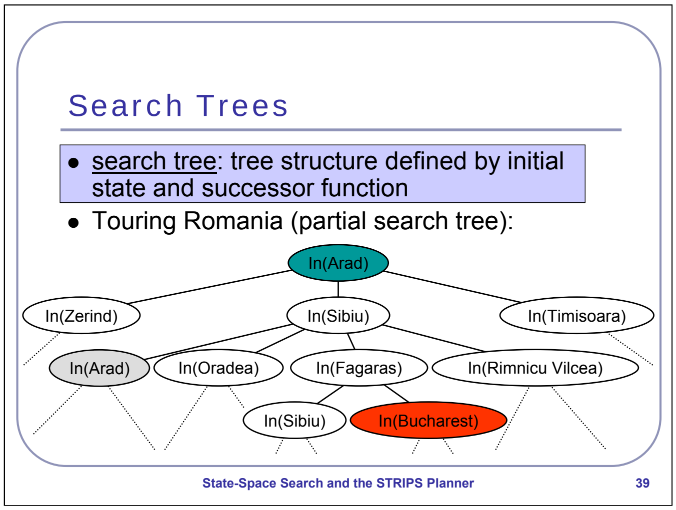
•initial state: on vacation in Arad, Romania

•goal? actions? -- “Touring Romania” cannot readily be described in terms of possible actions, goals, and path cost



Touring Romania: Search Problem Definition

- **initial state:**
  - **In(Arad)**
  - all states: current location only (abstraction)
- **possible Actions:**
  - **DriveTo(Zerind), DriveTo(Sibiu), DriveTo(Timisoara), etc.**
  - actions: applicable if there is a direct road from the current location to the destination
- **goal state:**
  - **In(Bucharest)**
  - goal state: here single state
- **step cost:**
  - **distances between cities**
  - path cost = sum of step costs; step cost is distance on map (abstraction)



Search Trees

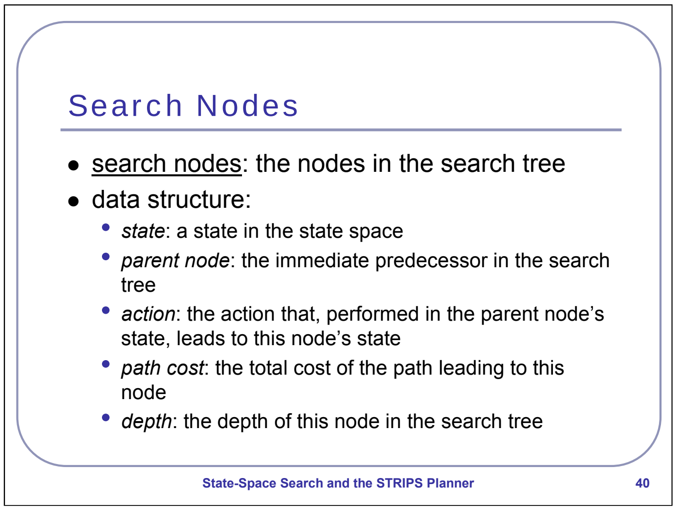
•**<u>search tree</u>: tree structure defined by initial state and successor function**

•**Touring Romania (partial search tree):**

- •initial state: root of tree (green)
- •children of any node: states reachable via a single action
  - •note: repeated states possible (e.g. grey state)
  - •note: tree may be infinite; infinite path: Arad – Sibiu - Arad – Sibiu - …
- •goal state (red)

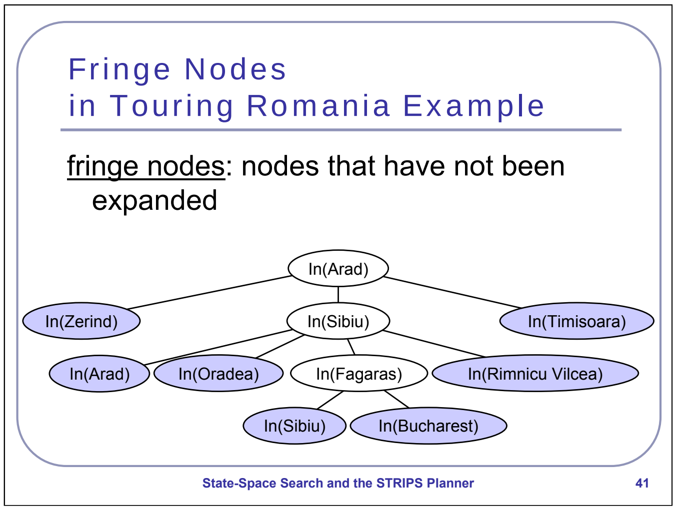
•search graph vs. search tree

- •graph: if nodes can be reached through multiple paths
- •corresponds to state space



Search Nodes

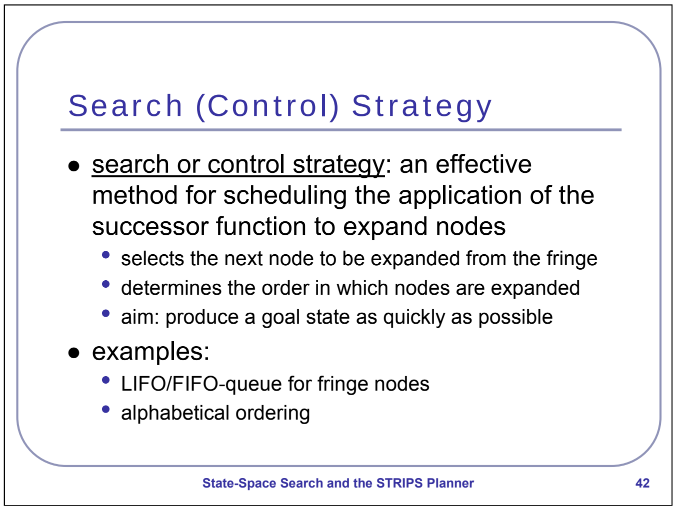
- **search nodes: the nodes in the search tree**
  - node is a bookkeeping structure in a search tree
- **data structure:**
  - ***state*: a state in the state space**
    - state (vs. node) corresponds to a configuration of the world
    - two nodes may contain equal states
  - ***parent node*: the immediate predecessor in the search tree**
    - nodes are on paths (defined by parent nodes)
  - ***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**
- alternative: representing paths only (sequences of actions):
  - possible, but state provides direct access to valuable information that might be expensive to regenerate all the time



Fringe Nodes in Touring Romania Example

- **<u>fringe nodes</u>: nodes that have not been expanded**
- shown: partial search tree for TR example
  - three expanded nodes (white)
  - seven (unexpanded) fringe nodes (blue)
    - fringe nodes are leaves in the search tree, but not necessarily vice versa

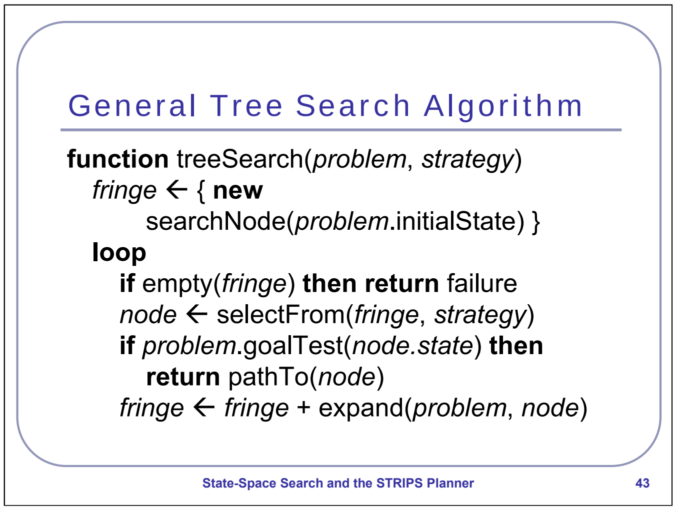
- remark: fringe nodes also called open nodes (vs. closed)



Search (Control) Strategy

- **<u>search or control strategy</u>: an effective method for scheduling the application of the successor function to expand nodes**
  - removes non-determinism from search method
  - **selects the next node to be expanded from the fringe**
    - closed nodes never need to be expanded again
  - **determines the order in which nodes are expanded**
    - exact order makes method deterministic
  - **aim: produce a goal state as quickly as possible**
    - strategy that produces goal state quicker is usually considered better
- **examples:**
  - **LIFO/FIFO-queue for fringe nodes** (two fundamental search strategies)
  - **alphabetical ordering**

•remark: complete search tree is usually too large to fit into memory, strategy determines which part to generate



General Tree Search Algorithm

function treeSearch(*problem*, *strategy*)

•find a solution to the given problem while expanding nodes according to the given strategy

***fringe* ← { new searchNode(*problem*.initialState) }**

•*fringe*: set of known states; initially just initial state

loop

•possibly infinite loop expands nodes

if empty(*fringe*) then return failure

•complete tree explored; no goal state found

***node* ← selectFrom(*fringe*, *strategy*)**

•select node from fringe according to search control strategy; the node will not be selected again

if *problem*.goalTest(*node.state*) then

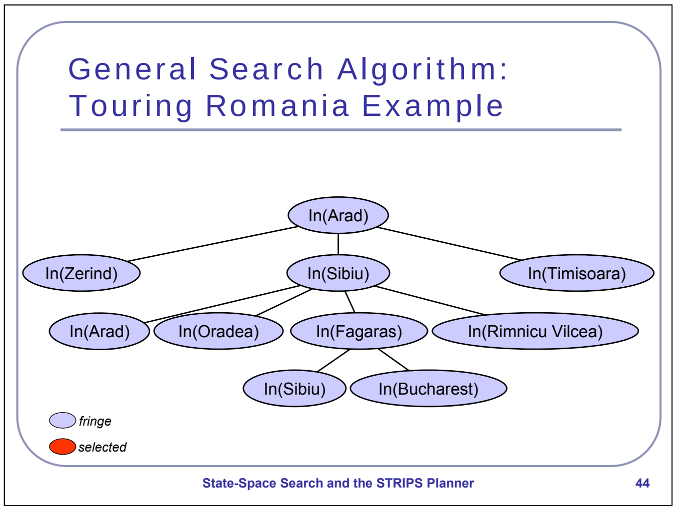
•goal test before expansion: to avoid trick problem like “get from Arad to Arad”

return pathTo(*node*)

•success: goal node found

***fringe* ← *fringe* + expand(*problem*, *node*)**

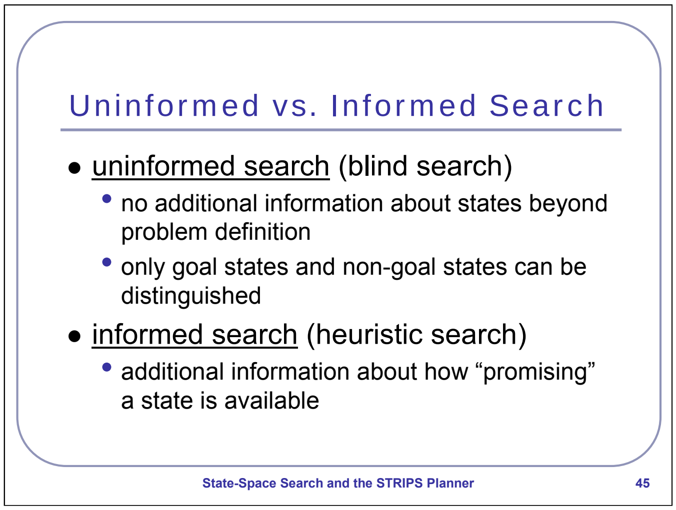
•otherwise: add new nodes to the fringe and continue loop



General Search Algorithm: Touring Romania Example

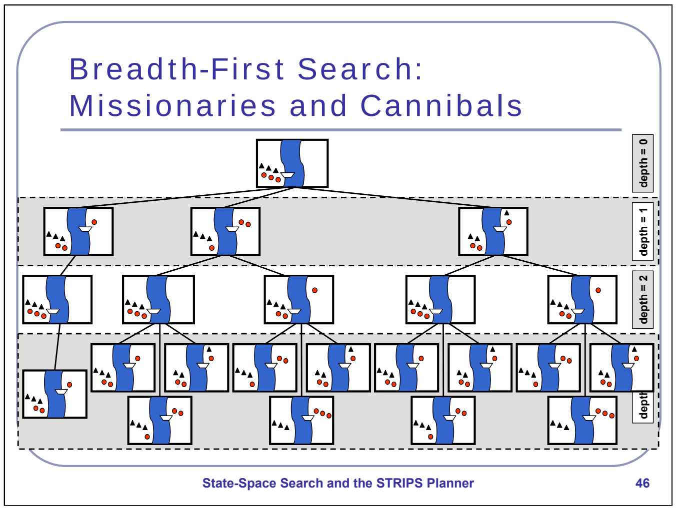
•algorithm: select and expand cycle until goal node is about to be expanded

•strategy: expand node on path to the goal – how do we know which node this is? (generally, we don’t!)



Uninformed vs. Informed Search

- **<u>uninformed search</u> (blind search)**
    - **no additional information about states beyond problem definition**
    - **only goal states and non-goal states can be distinguished**
    - the order of node expansion does not depend on the location of the goal state
- **<u>informed search</u> (heuristic search)**
    - **additional information about how “promising” a state is available**



Breadth-First Search: Missionaries and Cannibals

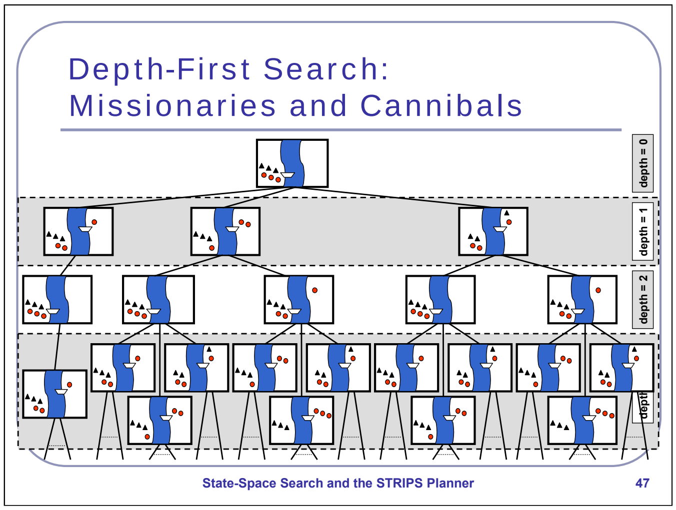
•first expand root node

•expand all nodes at depth 1 left to right (i.e. order depends on order in which successors have been generated)

•expand all nodes at depth 2, again left to right

•etc.

•no nodes beyond depth 3 shown but breadth-first search would continue



Depth-First Search: Missionaries and Cannibals

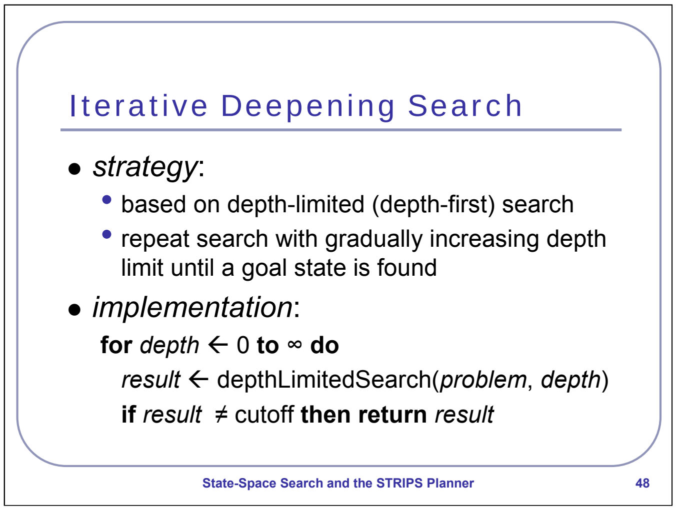
•expand left-most sub-tree

- •this constitutes an infinite sub-tree and the algorithm would never return, therefore the rest of the animation is wrong for this example!

•back up to depth 1; memory is freed up

•expand the remaining sub-trees

- •note: only one path including all siblings in memory at any one time



Iterative Deepening Search

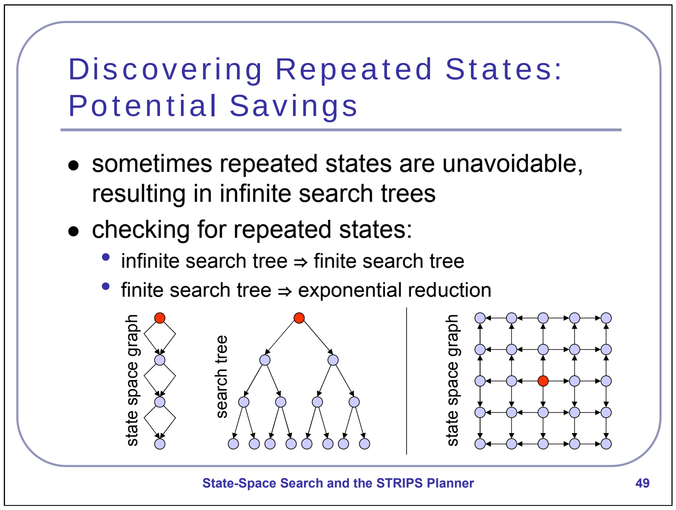
•***strategy*:**

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

•***implementation*:**

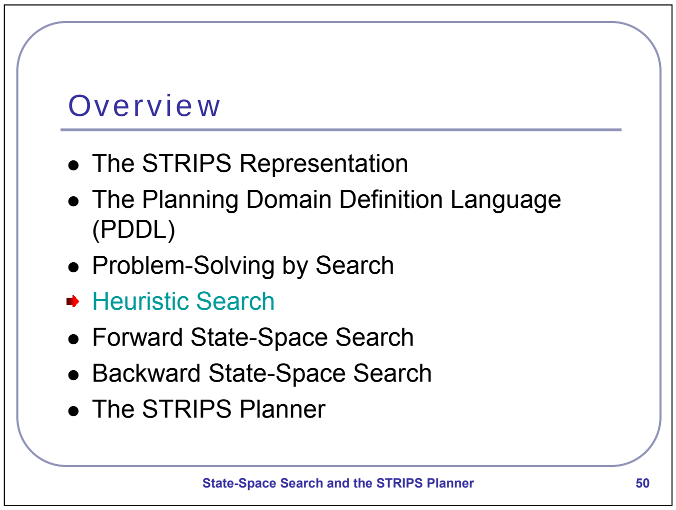
- **for *depth* ← 0 to ∞ do**
- loop over increasing depth limit
- ***result* ← depthLimitedSearch(*problem*, *depth*)**
- perform depth-limited search with current depth limit
- **if *result* ≠ cutoff then return *result***
- terminate search when no cut-off occurred (we have a solution or failure)

•iterative deepening search finds shallowest goal node



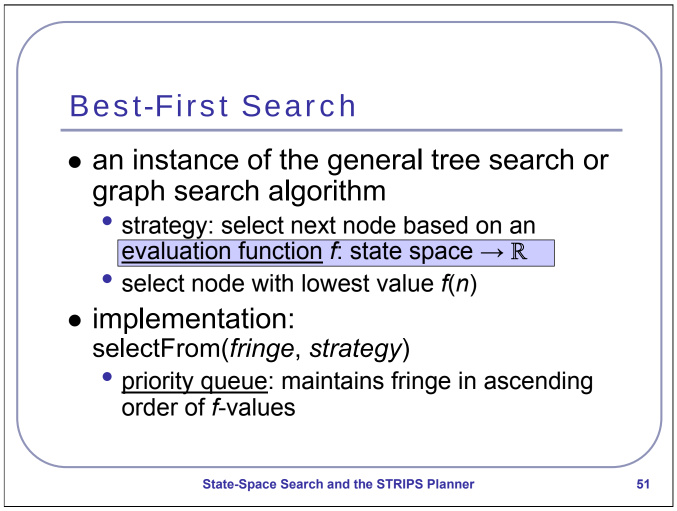
Discovering Repeated States: Potential Savings

- **sometimes repeated states are unavoidable, resulting in infinite search trees**
  - e.g. when actions are reversible; search graph rather than search tree
- **checking for repeated states:** (during the search process)
  - **infinite search tree ⇒ finite search tree**
    - reduces the search tree to the part that is necessary to span the state space graph (e.g. M&C, Touring Romania problem)
  - finite search tree ⇒ exponential reduction
    - example left: worst case scenario; true exponential reduction (reduction from exponential to linear function)
    - example right: more realistic example; still exponential reduction (exponential to polynomial)



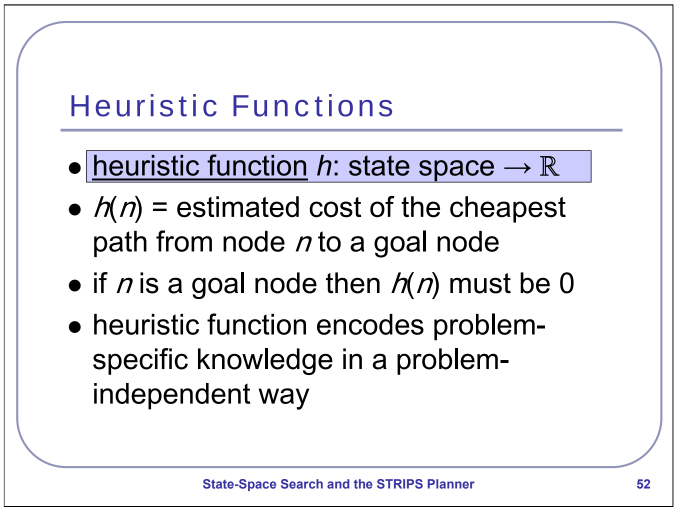
Overview

- **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**
    - tree or graph search: both possible; difference only lies in test for repeated states
    - **strategy: select next node based on an <u>evaluation function</u> *f*: state space → ℝ**
        - evaluation function: determines the search strategy
        - intuition: choose function that estimates the distance to the goal
    - **select node with lowest value *f*(*n*)**
        - lowest *f*-value means best node: hence best-first search
- **implementation: selectFrom(*fringe*, *strategy*)**
    - **<u>priority queue</u>: maintains fringe in ascending order of *f*-values**
        - implementation as binary tree: nodes can be added/retrieved in log-time (still expensive)



Heuristic Functions

•**<u>heuristic function</u> *h*: state space → $\mathbb{R}$**

•*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

•difference between evaluation function and heuristic function:

- •good evaluation function makes sure nodes are expanded in an order that leads straight to the optimal solution
- •good heuristic function always gives the correct distance to the nearest goal node
- •evaluation function is not problem-specific, but uses heuristic function which is problem-specific

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”

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

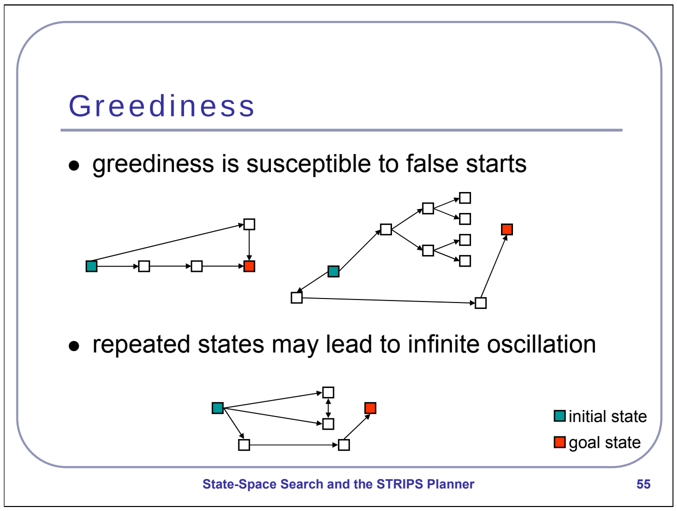
Touring in Romania: Heuristic

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

- •straight-line distance: Euclidean distance
- •distance to Bucharest because our goal is to be in Bucharest

•[table]

- •h_{SLD}(Bucharest) = 0
- •h_{SLD}(Fagaras) = 176 < 211 driving distance
- •$h_{SLD}(n)$ cannot be computed from the problem description, it represents additional information



Greediness

•greediness is susceptible to false starts

•[left figure]

- •GBFS will go to node at top first because this is closest to the goal node
- •solution path is sub-optimal

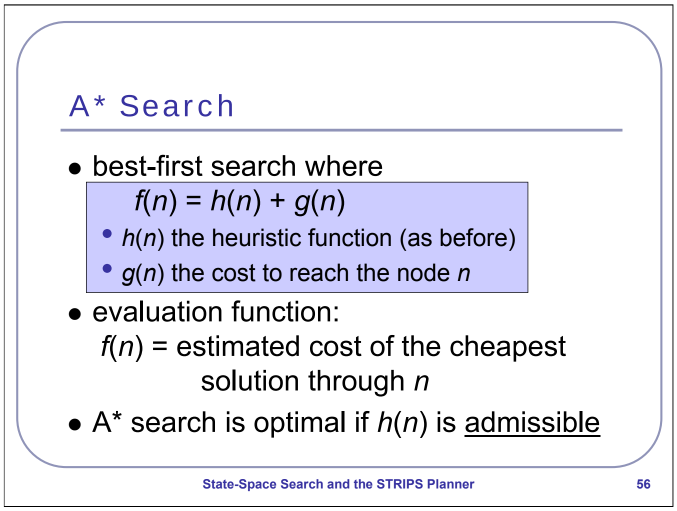
•[right figure]

- •GBFS will first explore the complete tree at the top that is not connected to the goal node
- •finally, it will go further away from the goal node and discover the (optimal) solution path
- •a lot of wasted search effort

•repeated states may lead to infinite oscillation

•[bottom figure]

- •algorithm may go back and forth between “close” nodes, never exploring node on way to goal



A* Search

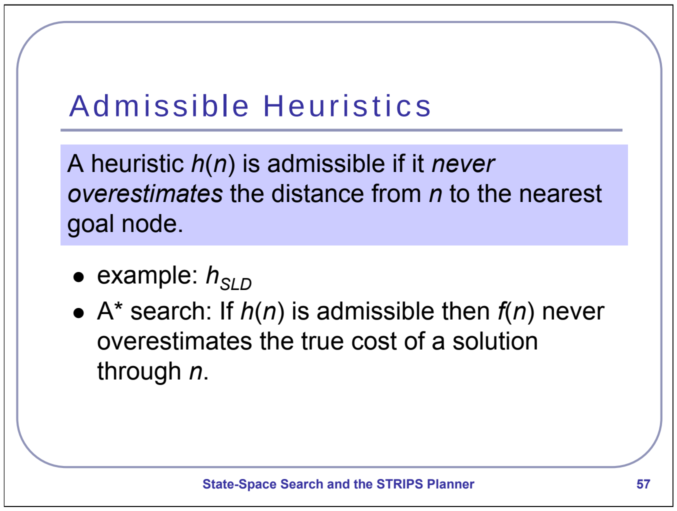
•best-first search where *f*(*n*) = *h*(*n*) + *g*(*n*)

- **•*h*(*n*) the heuristic function (as before)**
- **•*g*(*n*) the cost to reach the node *n***
  - •adds a breadth-first component to GBFS

•evaluation function: *f*(*n*) = estimated cost of the cheapest solution through *n*

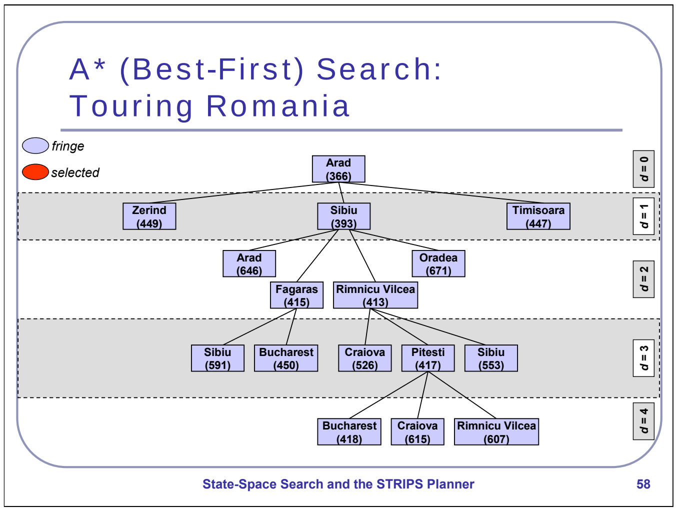
- •expand that node next which is on the cheapest path to a goal node

•A* search is optimal if *h*(*n*) is <u>admissible</u>



Admissible Heuristics

- **A heuristic $h(n)$ is admissible if it *never overestimates* the distance from n to the nearest goal node.**
  - admissible heuristics usually think the nearest goal node is closer than it actually is
  - **example: h_{SLD}**
    - h_{SLD}: shortest distance between two point is straight line, hence h_{SLD} is admissible
  - **A* search: If $h(n)$ is admissible then $f(n)$ never overestimates the true cost of a solution through n.**
    - since $f(n) = h(n) + g(n)$ and $g(n)$ is the exact cost of reaching n, $f(n)$ cannot overestimate the true cost of a solution through n



A* (Best-First) Search: Touring Romania

- •initial state: in Arad; values shown are evaluation function $f(n) = h(n) + g(n)$

•select Arad; expand Arad

- •lowest f-value: Sibiu (393); means: possible path through Sibiu with cost 393

•select Sibiu; expand Sibiu

- •lowest f-value: Rimnicu Vilcea (413); means: possible path through Rimnicu Vilcea with cost 413

•select Rimnicu Vilcea; expand Rimnicu Vilcea

- •lowest f-value: Fagaras (415); expanding Rimnicu Vilcea showed f-value too optimistic

•select Fagaras; expand Fagaras

- •lowest f-value: Pitesti (417); expanding Fagaras showed f-value too optimistic

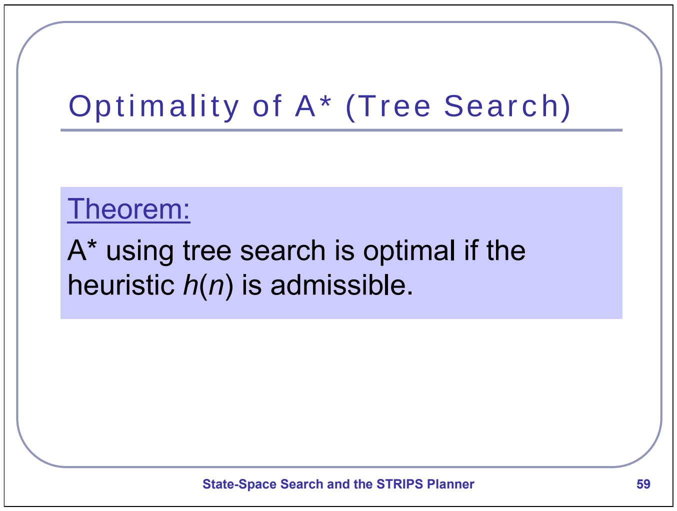
•select Pitesti; expand Pitesti

- •lowest f-value: Bucharest (418)

•select Bucharest

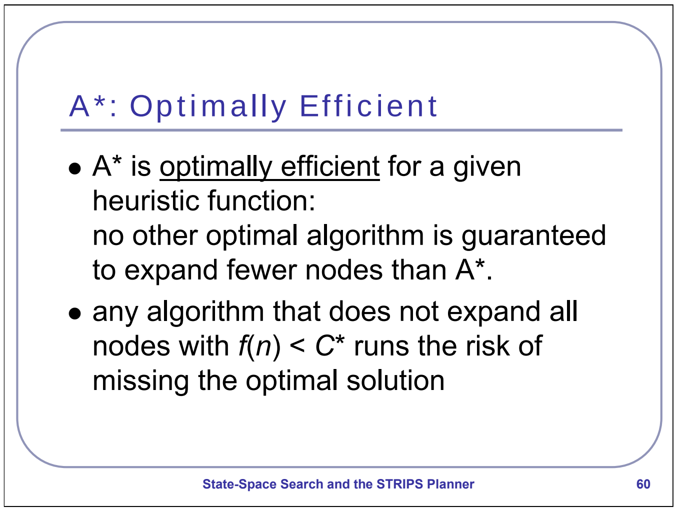
- •goal node test succeeds

•note: search cost not minimal as for GBFS but solution is optimal



Optimality of A* (Tree Search)

- **<u>Theorem:</u> A* using tree search is optimal if the heuristic *h*(*n*) is admissible.**
  - reminder: optimal means finds a minimal-path cost solution



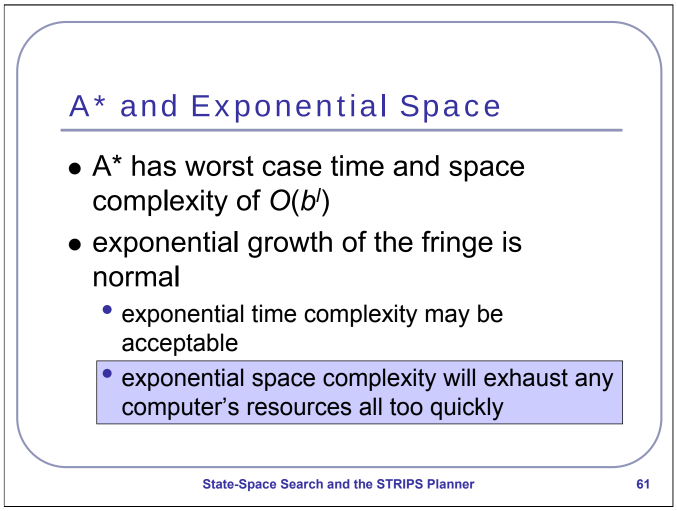
A*: Optimally Efficient

•A* is <u>optimally efficient</u> for a given heuristic function: no other optimal algorithm is guaranteed to expand fewer nodes than A*.

- •efficiency can still be increased with a different, more accurate heuristic for a given problem
- •but: efficiency does not only depend on number of nodes expanded

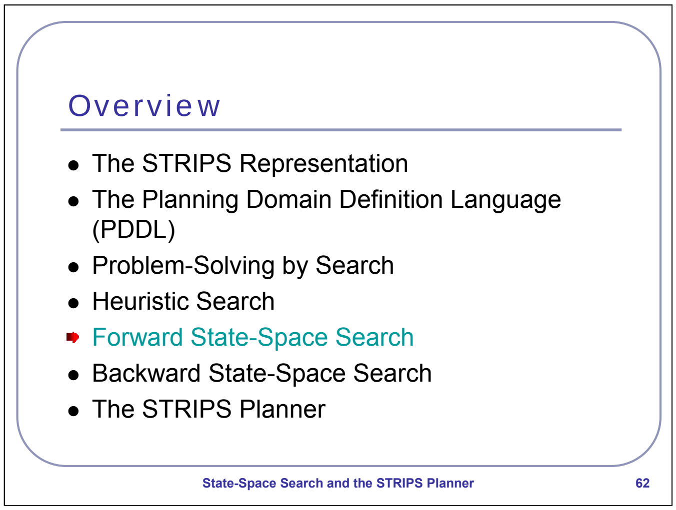
•any algorithm that does not expand all nodes with $f(n) < C^*$ runs the risk of missing the optimal solution

- •suppose there is a node with $f(n) < C^*$ that is not expanded before a goal node
- •then there could be a path of cost with $f(n) < C^*$ through that node which would be better than the goal node found



A* and Exponential Space

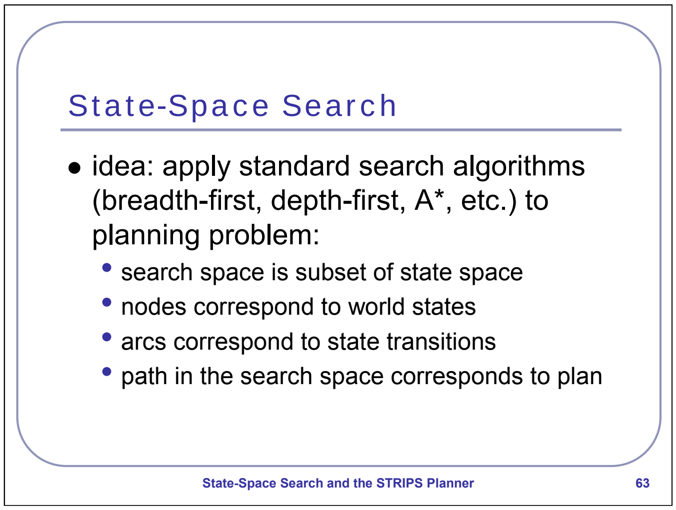
- **A* has worst case time and space complexity of $O(b^l)$**
- **exponential growth of the fringe is normal**
    - **exponential time complexity may be acceptable**
    - **exponential space complexity will exhaust any computer’s resources all too quickly**
        - and with the memory exhausted A* cannot continue and fails – no solution will be found



Overview

The STRIPS Representation

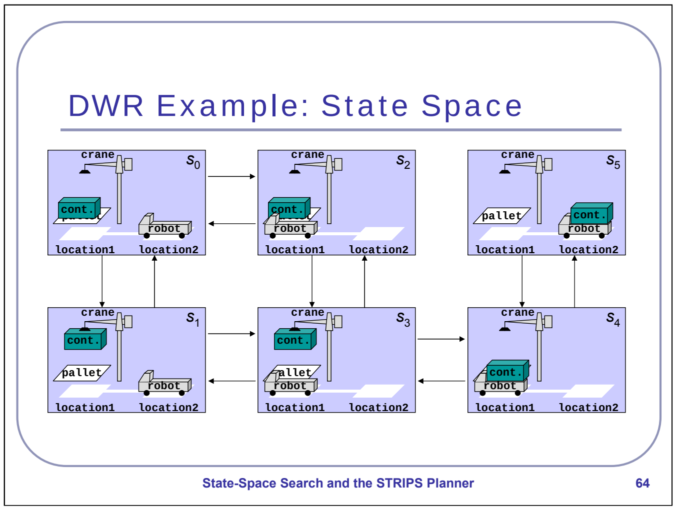
- **The Planning Domain Definition Language (PDDL)**
- **Problem-Solving by Search**
- **Heuristic Search**
- **Forward State-Space Search**
    - now: using standard search algorithms to perform a forward search for a goal state
- **Backward State-Space Search**
- **The STRIPS Planner**



State-Space Search

•idea: apply standard search algorithms (breadth-first, depth-first, A*, etc.) to planning problem:

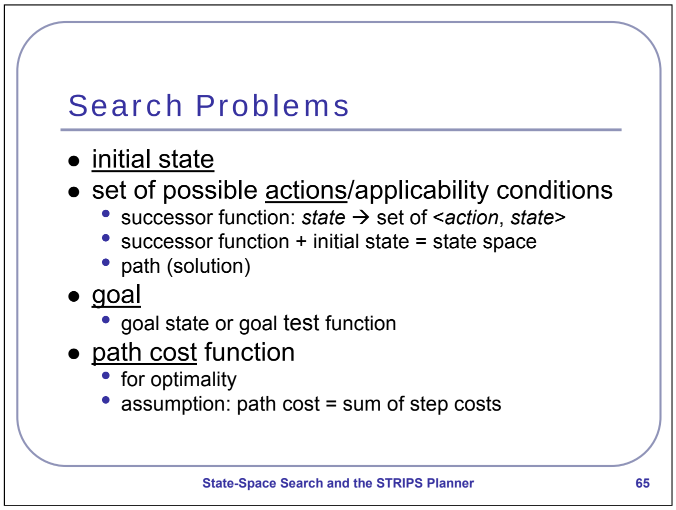
- **•search space is subset of state space**
  - •subset: generate only reachable states until a goal state has been found
- **•nodes correspond to world states**
- **•arcs correspond to state transitions**
  - •arcs are labelled with actions
- **•path in the search space corresponds to plan**
  - •path from initial state to goal state is solution



DWR Example: State Space

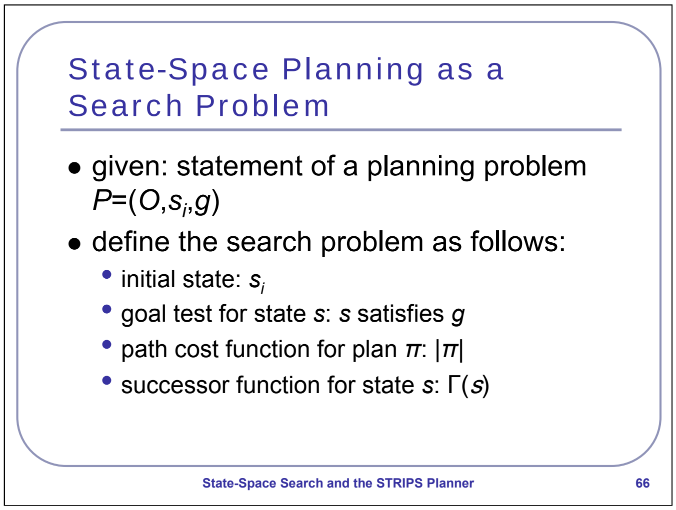
•from introduction

- •nodes are sets of ground atoms (shown here as 3D visualisations)
- •transitions should be labelled with ground operator instances (actions), e.g. move(robot,location1,location2)



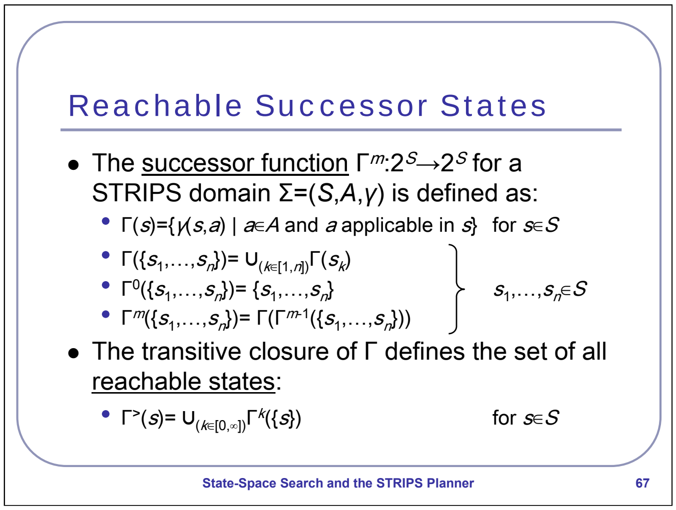
Search Problems

- **initial state**: current state the world is in (state = situation)
  - STRIPS states: sets of ground atoms
- finite **set of possible actions** with **applicability conditions**
  - **successor function: *state* → set of <*action*, *state*>**: corresponds to state transition function as defined for STRIPS actions
  - **successor function + initial state = state space**: directed graph with states as nodes and actions as arcs
  - **path** (in the graph) **(solution)**
- **goal**
  - **goal state** (not applicable) **or goal test function**: for multiple goal states; states in which goal holds
- **path cost function**
  - **for optimality**
  - **assumption: path cost = sum of step costs** (cost of applying a given action in a given state)



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: $|\pi|$**
        - simplification: plan length = path cost
    - **successor function for state s: $\Gamma(s)$**
        - to be defined next



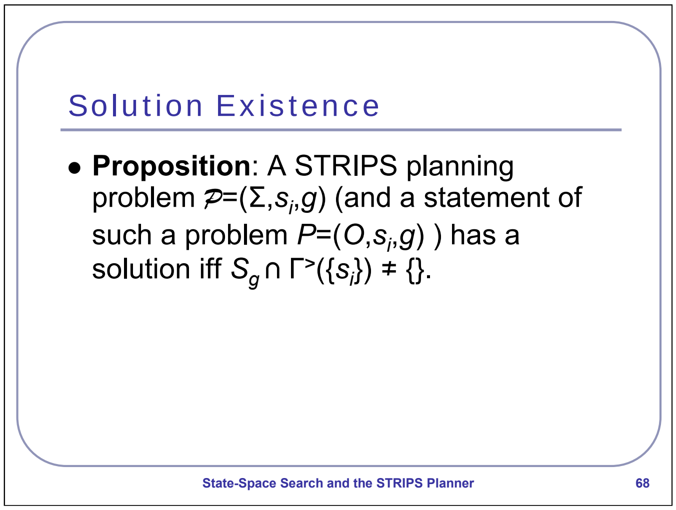
Reachable Successor States

•The <u>successor function</u> Γ^m:$2^S \rightarrow 2^S$ for a STRIPS domain $\Sigma=(S,A,\gamma)$ is defined as:

- •$\Gamma(s)=\{\gamma(s,a) \mid a\in A$ and a applicable in $s\}$ for $s\in S$
  - •all states that can be reached by applying exactly one applicable action
- •$\Gamma(\{s_1,\ldots,s_n\})= \cup_{(k\in[1,n])}\Gamma(s_k)$
  - •union of all states that can be reached by applying exactly one applicable action
- •$\Gamma^0(\{s_1,\ldots,s_n\})= \{s_1,\ldots,s_n\}$
  - •identity function; the states themselves
- •$\Gamma^m(\{s_1,\ldots,s_n\})= \Gamma(\Gamma^{m-1}(\{s_1,\ldots,s_n\}))$
  - •union of all states that can be reached by applying exactly *m* applicable actions

•The transitive closure of Γ defines the set of all <u>reachable states</u>:

- •$\Gamma^>(s)= \cup_{(k\in[0,\infty])}\Gamma^k(\{s\})$ for $s\in S$
  - •pronounce: gamma forward
  - •all states that can be reached by applying any number of applicable actions

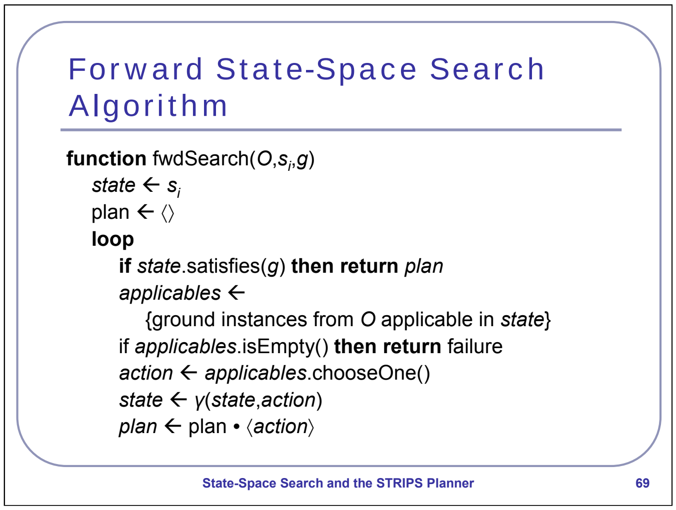


Solution Existence

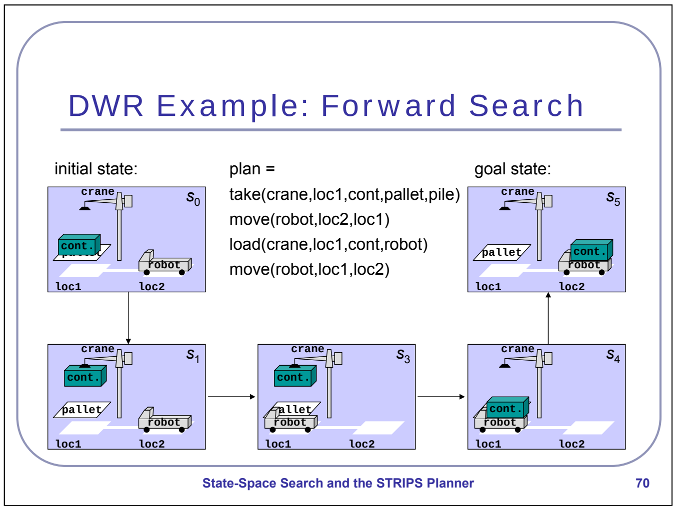
•Proposition: A STRIPS planning problem $\mathcal{P}=(\Sigma,s_i,g)$ (and a statement of such a problem $P=(O,s_i,g)$) has a solution iff $S_g \cap \Gamma^{>}(\{s_i\}) \neq \{\}$.

  •… iff there is a goal state that is also a reachable state

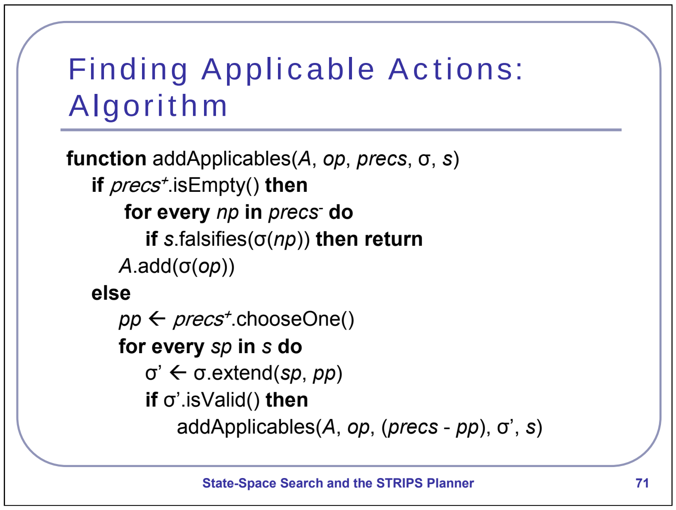
•enumerate all reachable states from the initial state (in some good order) and we will generate a goal state eventually = forward search



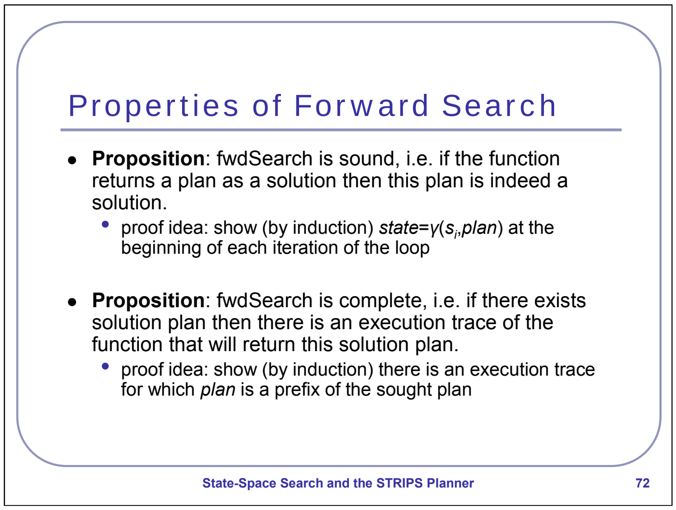
- **function fwdSearch(*O*,s_i,*g*)**
    - given: statement of a STRIPS planning problem; return a solution plan (or failure)
    - non-deterministic version
- ***state* ← s_i**
    - start with the initial state
- **plan ← ⟨⟩**
    - initialize solution with empty plan (partial plan: prefix of the solution)
- **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()**
    - non-deterministically choose an applicable action
- ***state* ← γ(*state*,*action*)**
- ***plan* ← plan • ⟨*action*⟩**



- **DWR Example: Forward Search**
  - goal state available at start
  - choose action; (non-deterministic; alternative would be "move" action)
  - compute successor state
  - chose action; (again non-deterministic; alternative would be "put" returning to s_0)
  - compute successor state
  - chose action
  - compute successor state
  - chose action
  - compute successor state; goal state!

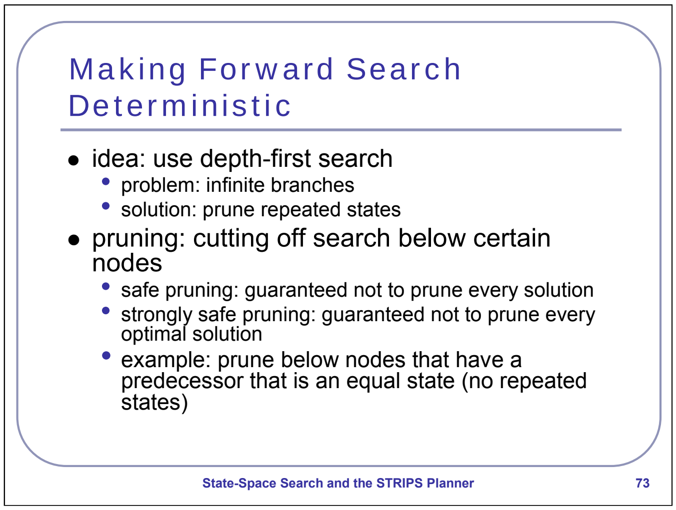


- **function addApplicables(*A*, *op*, *precs*, σ, *s*)**
  - Parameters: set of actions, operator, set of remaining preconditions, partial substitution, state
- **if *precs*$^+$.isEmpty() then**
  - Note: σ should now be complete
- **for every *np* in *precs*$^-$ do**
- **if *s*.falsifies(σ(*np*)) then return**
- ***A*.add(σ(*op*))**
  - test for inconsistent effects before adding!
- **else**
- ***pp* $\leftarrow$ *precs*$^+$.chooseOne()**
  - Heuristics: nr of atoms in state; nr of unbound variables
- **for every *sp* in *s* do**
- **σ' $\leftarrow$ σ.extend(*sp*, *pp*)**
- **if σ'.isValid() then**
- **addApplicables(*A*, *op*, (*precs* - *pp*), σ', *s*)**



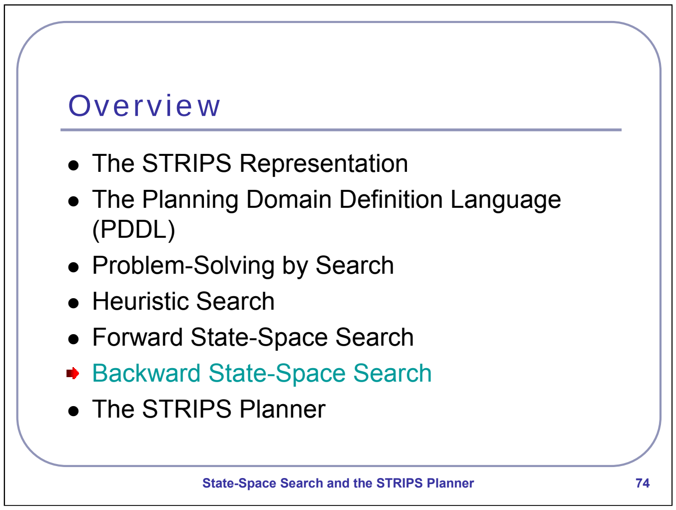
Properties of Forward Search

- **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**
  - variable *state* always contains STRIPS state that is result of applying *plan* (variable) in initial state
  - hence: when *state* contains goal state *plan* contains solution plan
- **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**
  - given a solution plan, the variable *plan* contains a prefix of that plan starting with the initial empty plan
  - chooseOne(…) can always choose the next step in the solution plan we are looking for



Making Forward Search Deterministic

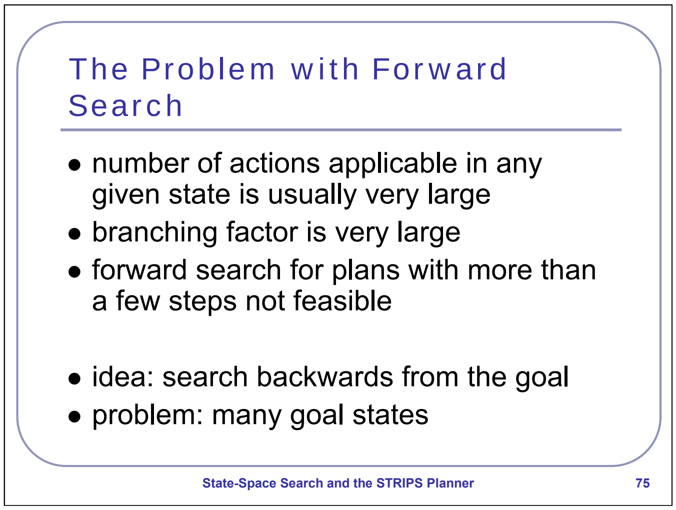
- **idea: use depth-first search**
  - **problem: infinite branches**
    - example: alternating between two states that are not solutions
  - **solution: prune repeated states**
    - search is finite: pruning repeated states means we will eventually enumerate the whole search space
- **pruning: cutting off search below certain nodes**
  - **safe pruning: guaranteed not to prune every solution**
    - but may prune some solutions
  - **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)**
    - pruning repeated states is strongly safe



Overview

The STRIPS Representation

- **The Planning Domain Definition Language (PDDL)**
- **Problem-Solving by Search**
- **Heuristic Search**
- **Forward State-Space Search**
    - just done: using standard search algorithms to perform a forward search for a goal state
- **Backward State-Space Search**
    - now: search backwards from the goal reduces search space size
- **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

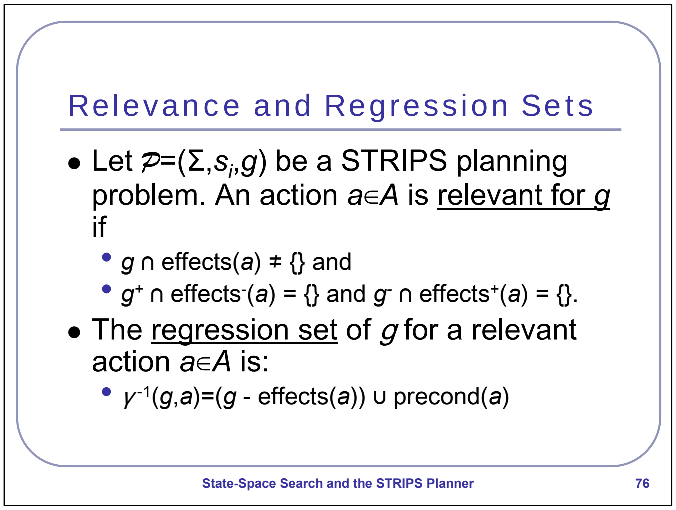
•forward search for plans with more than a few steps not feasible

- •forward search unnecessarily generates a large part of the search space which makes it highly inefficient

•idea: search backwards from the goal

•problem: many goal states

- •applying reverse operators only works for single goal state



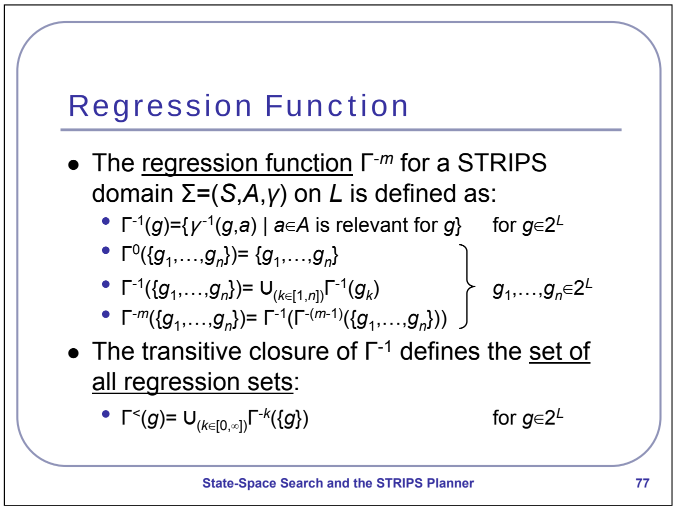
Relevance and Regression Sets

•Let $\mathcal{P}=(\Sigma,s_i,g)$ be a STRIPS planning problem. An action $a \in A$ is <u>relevant for g</u> if

- $g \cap \text{effects}(a) \neq \{\}$ and
  - a's effects contribute to g
- $g^+ \cap \text{effects}^-(a) = \{\}$ and $g^- \cap \text{effects}^+(a) = \{\}$.
  - a's effects do not conflict with g

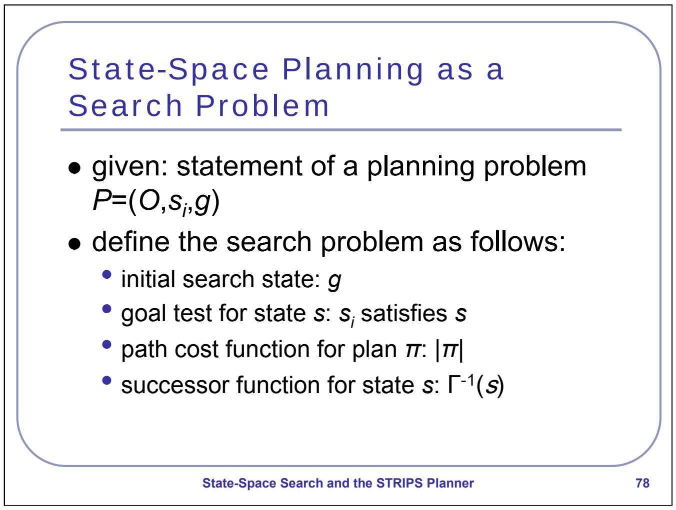
•The <u>regression set</u> of g for a relevant action $a \in A$ is:

- $\gamma^{-1}(g,a)=(g - \text{effects}(a)) \cup \text{precond}(a)$
  - subtract all effects, not just positive ones
  - note: goal and regression set ($\gamma^{-1}(g,a)$) are sets of ground literals
  - regression set can be seen as sub-goal



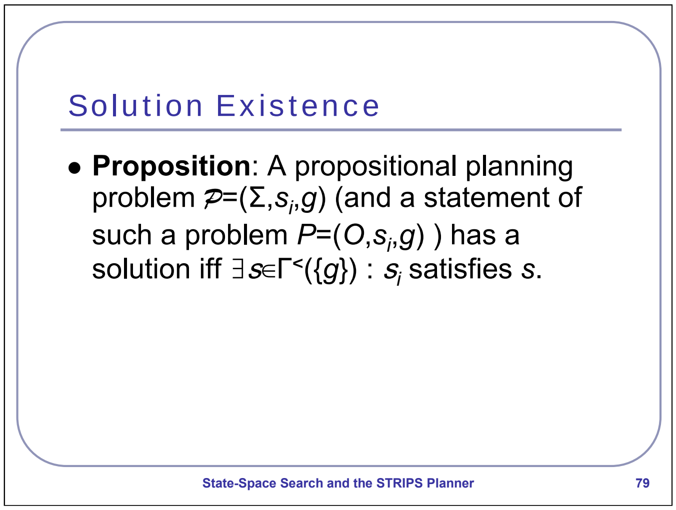
Regression Function

- **The <u>regression function</u>** Γ^{-m} for a STRIPS domain $\mathbf{\Sigma=(S,A,\gamma)}$ on L is defined as:
  - $\Gamma^{-1}(g)=\{\gamma^{-1}(g,a) \mid a\in A$ is relevant for $g\}$ for $g\in 2^L$
    - regression set for a single set of (goal) propositions
  - $\Gamma^{0}(\{g_1,\ldots,g_n\})= \{g_1,\ldots,g_n\}$
    - as for successors
  - $\Gamma^{-1}(\{g_1,\ldots,g_n\})= \cup_{(k\in[1,n])}\Gamma^{-1}(g_k)$
    - union of individual regression sets
  - $\Gamma^{-m}(\{g_1,\ldots,g_n\})= \Gamma^{-1}(\Gamma^{-(m-1)}(\{g_1,\ldots,g_n\}))$
    - minimal sets of propositions that must hold in a state s from which m actions lead to a state in which one of $g_1,\ldots,g_n$ is satisfied
- The transitive closure of Γ^{-1} defines the <u>set of all regression sets</u>:
  - $\Gamma^{<}(g)= \cup_{(k\in[0,\infty])}\Gamma^{-k}(\{g\})$ for $g\in 2^L$
    - pronounce: gamma backward



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**
    - search backwards from the goal
  - **goal test for state s: s satisfies s_i**
    - initial state satisfies regression set (sub-goal)
  - **path cost function for plan π: $|\pi|$**
  - **successor function for state s: $\Gamma^{-1}(s)$**
    - as defined in previous slide

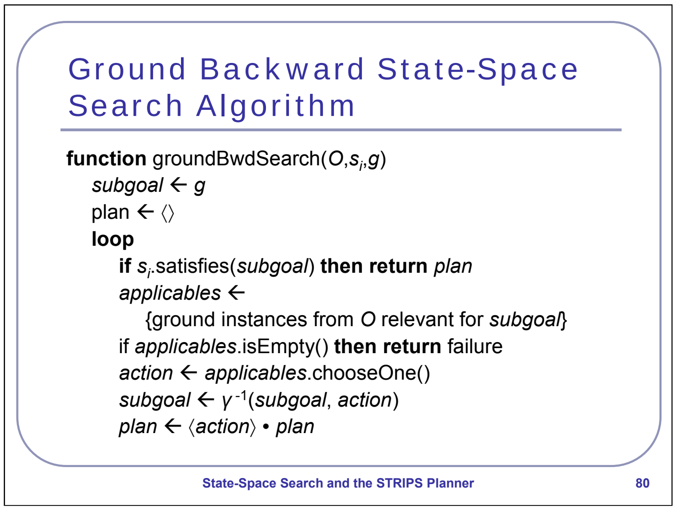


Solution Existence

•Proposition: A propositional planning problem $\mathcal{P}=(\Sigma,s_i,g)$ (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.

- … iff there is a minimal set of propositions amongst all regression sets that is a subset of the initial state

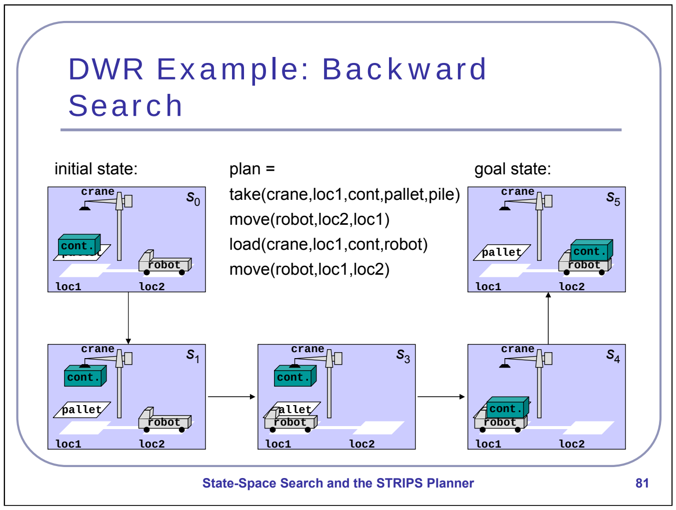
•enumerate all regression sets from the goal (in some good order) and we will generate a subset of the initial state eventually = backward search



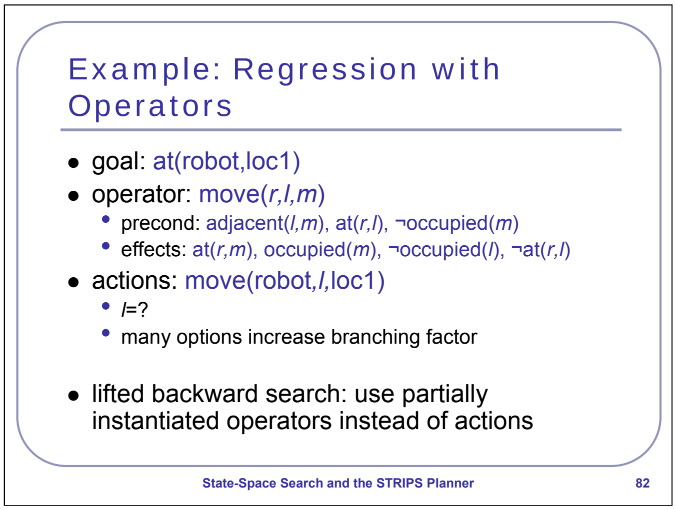
Ground Backward State-Space Search Algorithm

- **function groundBwdSearch(*O*,*s*$_i$,*g*)**
  - given: statement of a STRIPS planning problem; return a solution plan (or failure)
  - non-deterministic version
- ***subgoal* ← *g***
  - start with the overall goal
- **plan ← ⟨⟩**
  - initialize solution with empty plan (partial plan: suffix of the solution)
- **loop**
- **if *s*$_i$.satisfies(*subgoal*) then return *plan***
- ***applicables* ← {ground instances from *O* relevant for *subgoal*}**
- **if *applicables*.isEmpty() then return failure**
- ***action* ← *applicables*.chooseOne()**
  - non-deterministically choose an applicable action
- ***subgoal* ← γ^{-1}(*subgoal*, *action*)**
- ***plan* ← ⟨*action*⟩ • *plan***

- sound and complete
- test for repeated sub-goals can be applied to prune all infinite branches

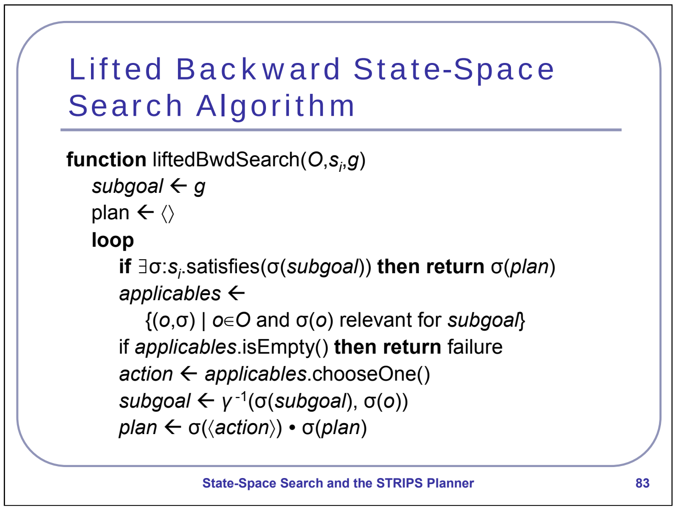


- **DWR Example: Backward Search**
  - note: sub-goal represented as state here, but goal description is not complete state description! shown state satisfies sub-goal
  - choose action
  - compute sub-goal using regression
  - chose action; (non-deterministic; alternative would be “move” returning to s_5)
  - compute sub-goal
  - chose action
  - compute sub-goal
  - chose action
  - compute sub-goal



Example: Regression with Operators

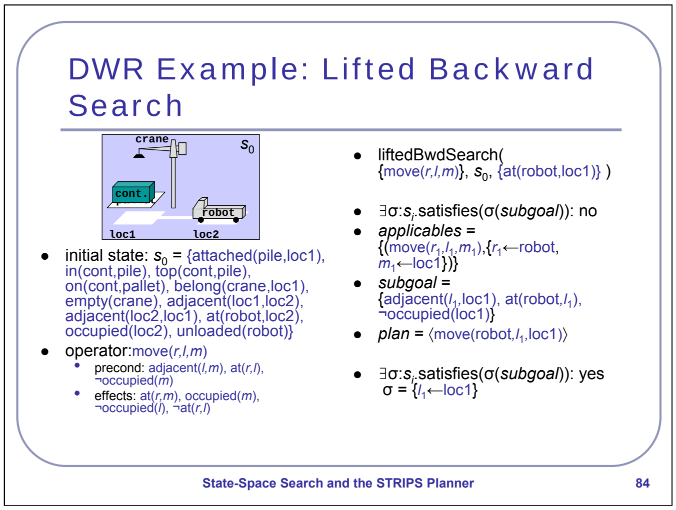
- **goal: at(robot,loc1)**
- **operator: move(*r,l,m*)**
  - **precond: adjacent(*l,m*), at(*r,l*), ¬occupied(*m*)**
  - **effects: at(*r,m*), occupied(*m*), ¬occupied(*l*), ¬at(*r,l*)**
    - operator may achieve or undo goal depending on variable bindings
- **actions: move(robot,*l*,loc1)**
  - ***l*=?**
    - to contribute to goal, *r* must bound to robot and *m* to loc1; *l* can remain unbound
  - **many options increase branching factor**
    - keeping variables unbound can significantly reduce the branching factor (as opposed to using actions)
- **lifted backward search: use partially instantiated operators instead of actions**
  - essentially same as ground version, but need to maintain appropriate variable substitutions



Lifted Backward State-Space Search Algorithm

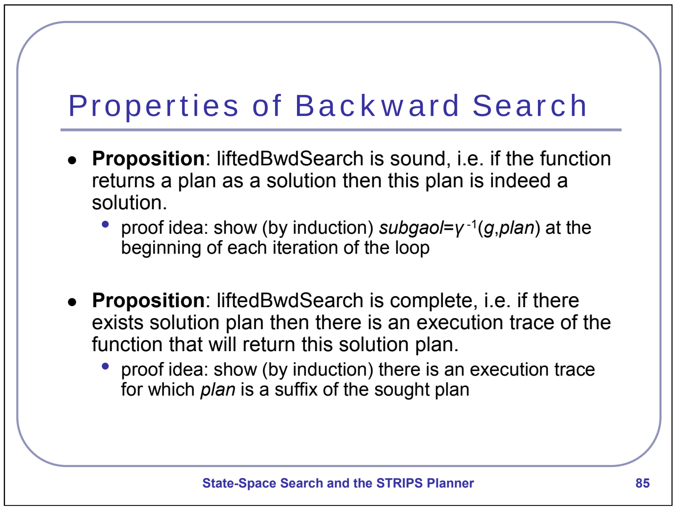
- **function liftedBwdSearch(O,s_i,g)**
- ***subgoal* ← *g***
- **plan ← ⟨⟩**
- **loop**
- **if $\exists\sigma$:s_i.satisfies(σ(*subgoal*)) then return σ(*plan*)**
  - need existence of substitution to test for goal satisfaction (variables in sub-goals are implicitly existentially quantified)
- ***applicables* ← {(*o*,σ) | $o \in O$ and σ(*o*) relevant for *subgoal*}**
  - need partial instantiation to test for relevance of operator (note: extension of definition of relevance straight forward)
- **if *applicables*.isEmpty() then return failure**
- ***action* ← *applicables*.chooseOne()**
- ***subgoal* ← γ^{-1}(σ(*subgoal*), σ(*o*))**
  - new sub-goal may contain variables (note: extension of definition of γ^{-1} straight forward)
- ***plan* ← σ(⟨*action*⟩) • σ(*plan*)**
  - add partially instantiated operator to plan and apply substitution to existing plan

- sound and complete



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)})**
- **∃σ:s_i.satisfies(σ(*subgoal*)): no**
  - at(robot,loc1) $\notin s_0$
- ***applicables* ={(move(r_1,l_1,m_1),{r_1←robot, m_1←loc1})}**
  - variable l_1 remains unbound
- ***subgoal* = {adjacent(l_1,loc1), at(robot,l_1), ¬occupied(loc1)}**
  - instantiated preconditions of the move-operator
- ***plan* = ⟨move(robot,l_1,loc1)⟩**
- **∃σ:s_i.satisfies(σ(*subgoal*)): yes**
  σ = {l_1←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.

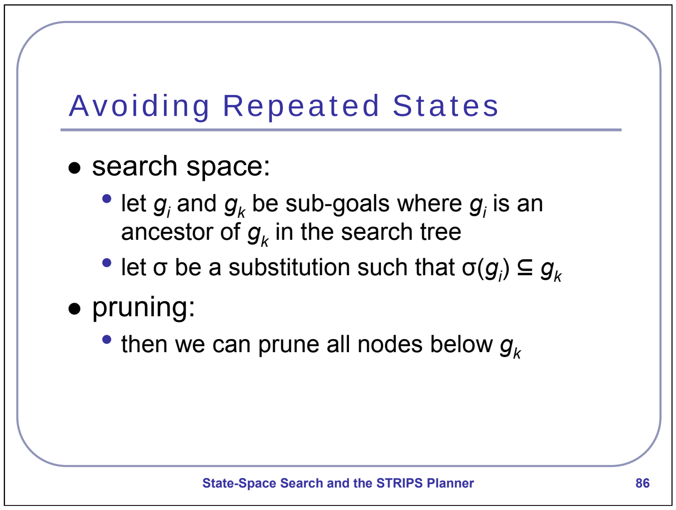
- **•proof idea: show (by induction) *subgaol*=γ^{-1}(*g*,*plan*) at the beginning of each iteration of the loop**

•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**

•proof ideas similar to forward case, but need to show that there are no variables in the final plan

- •final sub-goal must be satisfied by initial state which is ground



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 $\sigma(g_i) \subseteq g_k$**
    - g_k is more specific sub-goal than g_i :
      - subset relation: g_k may contain additional conjuncts
      - substitution: variables in g_i are specific values in g_k
    - note similarity to subsumption relation in theorem proving
- **pruning:**
  - **then we can prune all nodes below g_k**
    - any plan achieving g_k from the initial state would also achieve g_i
    - thus: solution via g_k and g_i is redundant

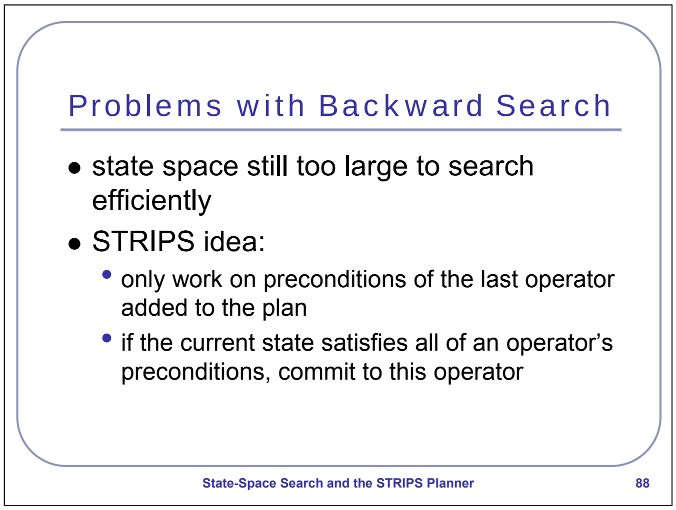
Overview

- 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

Overview

The STRIPS Representation

- **The Planning Domain Definition Language (PDDL)**
- **Problem-Solving by Search**
- **Heuristic Search**
- **Forward State-Space Search**
- **Backward State-Space Search**
  - just done: search backwards from the goal reduces search space size
- **The STRIPS Planner**
  - now: further reduction of the search space size in the STRIPS algorithm (not complete)



Problems with Backward Search

•state space still too large to search efficiently

•especially when STRIPS was developed (early 70s), but still true today

•STRIPS idea:

•only work on preconditions of the last operator added to the plan

•reduces branching factor significantly

•if the current state satisfies all of an operator's preconditions, commit to this operator

•reduces need for backtracking (in deterministic implementation)

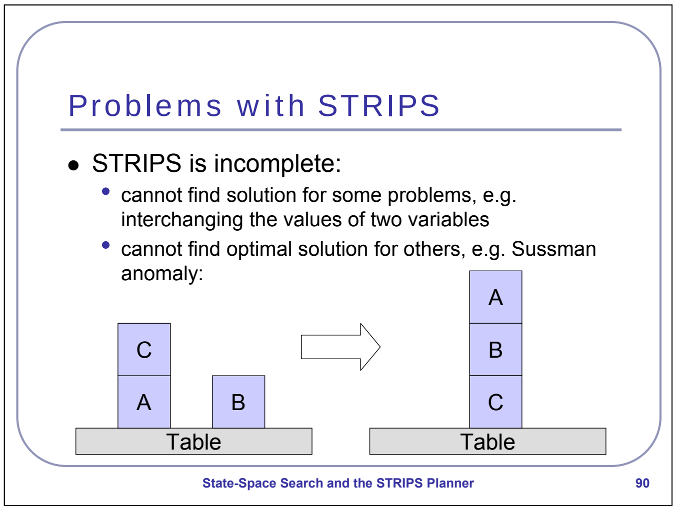
Ground-STRIPS Algorithm

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 ← γ(*s*, *subplan* • ⟨*action*⟩)
    plan ← *plan* • *subplan* • ⟨*action*⟩

State-Space Search and the STRIPS Planner 89

Ground-STRIPS Algorithm

- **function groundStrips(*O*,*s*,*g*)**
  - recursive function will be called with intermediate state and new sub-goals
- **plan ← ⟨⟩**
- **loop**
- **if *s*.satisfies(*g*) then return *plan***
- ***applicables* ← {ground instances from *O* relevant for *g*-*s*}**
  - focus on unachieved parts of the sub-goal
- **if *applicables*.isEmpty() then return failure**
- ***action* ← *applicables*.chooseOne()**
  - non-deterministic choice point
- ***subplan* ← groundStrips(*O*,*s*,*action*.preconditions())**
  - recursive call: generate sub-plan that achieves the preconditions of the regression operator
- **if *subplan* = failure then return failure**
- ***s* ← γ(*s*, *subplan* • ⟨*action*⟩)**
  - commit to the successful plan and action and use resulting state as new “initial” state
- ***plan* ← *plan* • *subplan* • ⟨*action*⟩**
  - update the plan accordingly



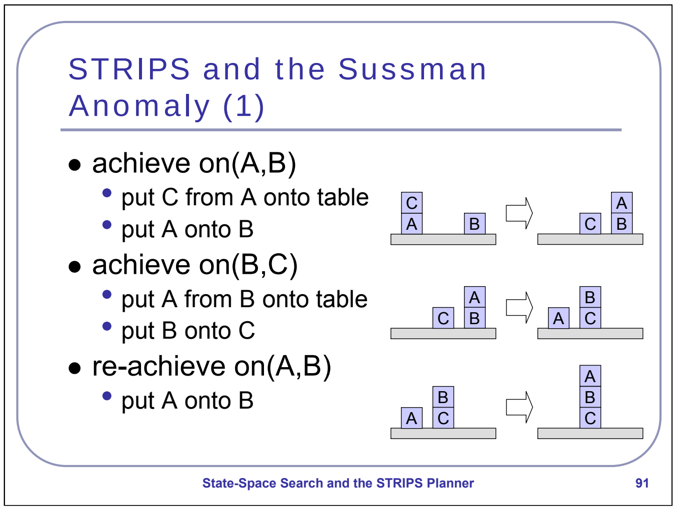
Problems with STRIPS

•STRIPS is incomplete:

- **•cannot find solution for some problems, e.g. interchanging the values of two variables**
  - •why?
- **•cannot find optimal solution for others, e.g. Sussman anomaly:**
  - •after achieving sub-goal, plan for next sub-goal will un-achieve previous sub-goal

•[figure]

- •Sussman anomaly: find plan for transforming left configuration into right configuration
- •goal given as {on(A,B), on(B,C)}



STRIPS and the Sussman Anomaly (1)

•two relevant operators at top level: “put A onto B” and “put B onto C”

•first case: choose “put A onto B”

•achieve on(A,B)

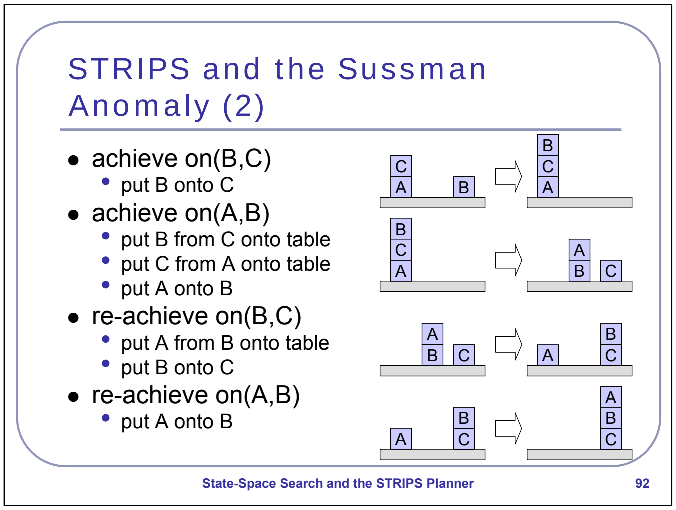
- **•put C from A onto table**
- **•put A onto B**
- •sub-plan complete from initial state; commit to it

•achieve on(B,C)

- **•put A from B onto table**
- **•put B onto C**
- •sub-plan complete from new state (un-achieves first sub-goal); commit to it

•re-achieve on(A,B)

- **•put A onto B**
- •plan complete



STRIPS and the Sussman Anomaly (2)

•second case: choose "put B onto C"

•**achieve on(B,C)**

- •**put B onto C**
- •sub-plan complete from initial state; commit to it

•**achieve on(A,B)**

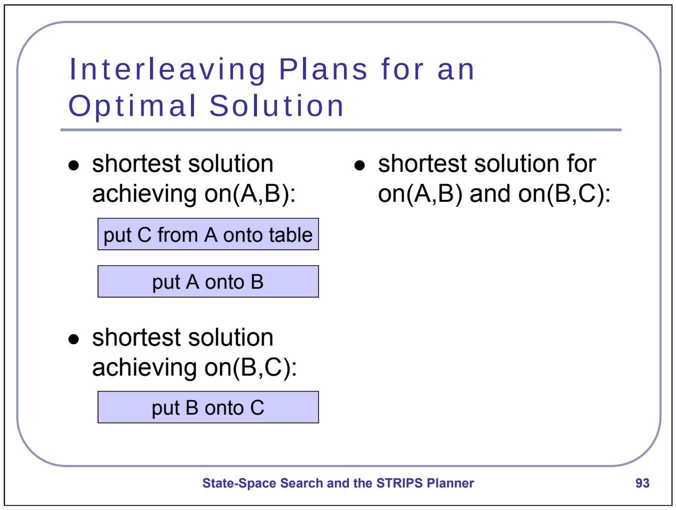
- •**put B from C onto table**
- •**put C from A onto table**
- •**put A onto B**
- •sub-plan complete from new state (un-achieves first sub-goal); commit to it

•**re-achieve on(B,C)**

- •**put A from B onto table**
- •**put B onto C**
- •sub-plan complete from new state (un-achieves second sub-goal); commit to it

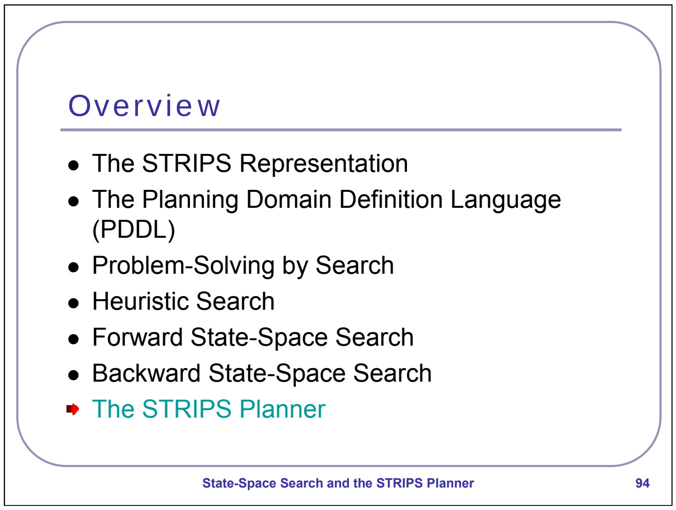
•**re-achieve on(A,B)**

- •**put A onto B**
- •plan complete



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):**
  - **put C from A onto table**
  - **put B onto C**
  - **put A onto B**
- note: optimal solution cannot be found by STRIPS algorithm because:
  - it cannot switch the sub-goal to work on during the search and
  - commits as soon as it found a path to the initial state



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**
    - just done: further reduction of the search space size in the STRIPS algorithm (not complete)