Automated Planning

Gerhard Wickler, U. Edinburgh

Introduction and Overview

Overview

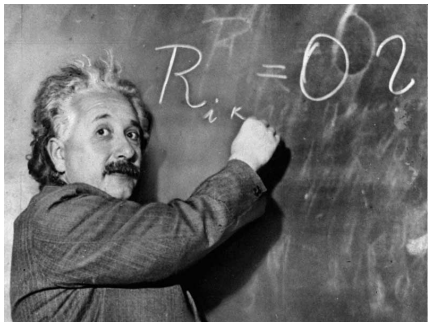
- What is AI Planning?
- A Conceptual Model for Planning
- Restricting Assumptions
- A Running Example: Dock-Worker Robots

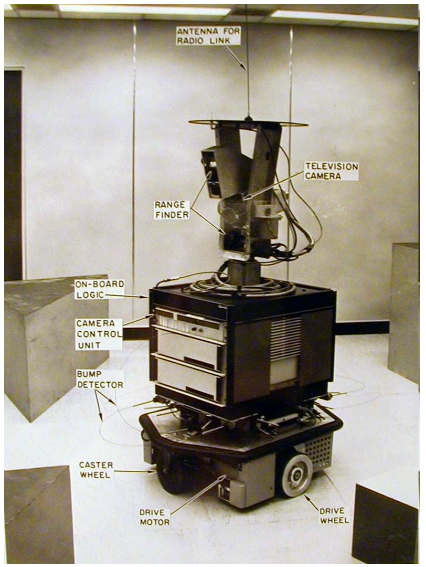
Defining AI Planning

- planning:
  - explicit deliberation process that chooses and organizes actions by anticipating their outcomes
  - aims at achieving some pre-stated objectives
- AI planning:
  - computational study of this deliberation process

Why Study Planning in AI?

- scientific goal of AI: understand intelligence
  - planning is an important component of rational (intelligent) behaviour
- engineering goal of AI: build intelligent entities
  - build planning software for choosing and organizing actions for autonomous intelligent machines





Domain-Specific vs. Domain-Independent Planning

- domain-specific planning: use specific representations and techniques adapted to each problem
  - important domains: path and motion planning, perception planning, manipulation planning, communication planning
- domain-independent planning: use generic representations and techniques
  - exploit commonalities to all forms of planning
  - leads to general understanding of planning

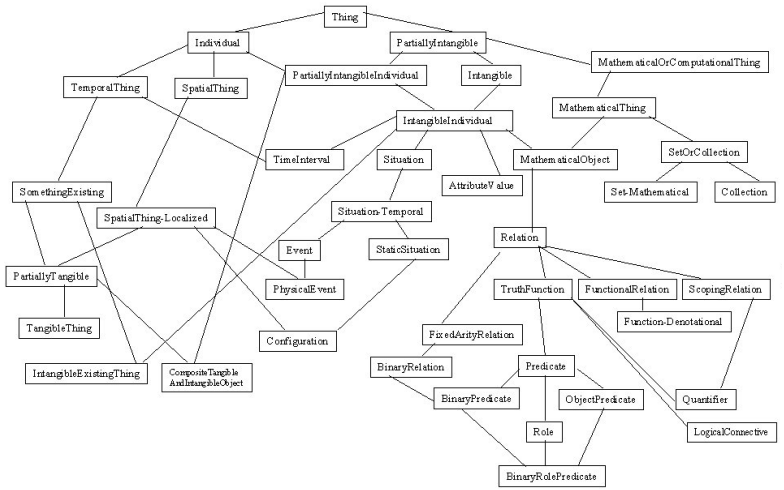
- domain-independent planning complements domain-specific planning

Overview

- What is AI Planning?
- A Conceptual Model for Planning
- Restricting Assumptions
- A Running Example: Dock-Worker Robots

Why a Conceptual Model?

- conceptual model: theoretical device for describing the elements of a problem
- good for:
  - explaining basic concepts
  - clarifying assumptions
  - analyzing requirements
  - proving semantic properties
- not good for:
  - efficient algorithms and computational concerns



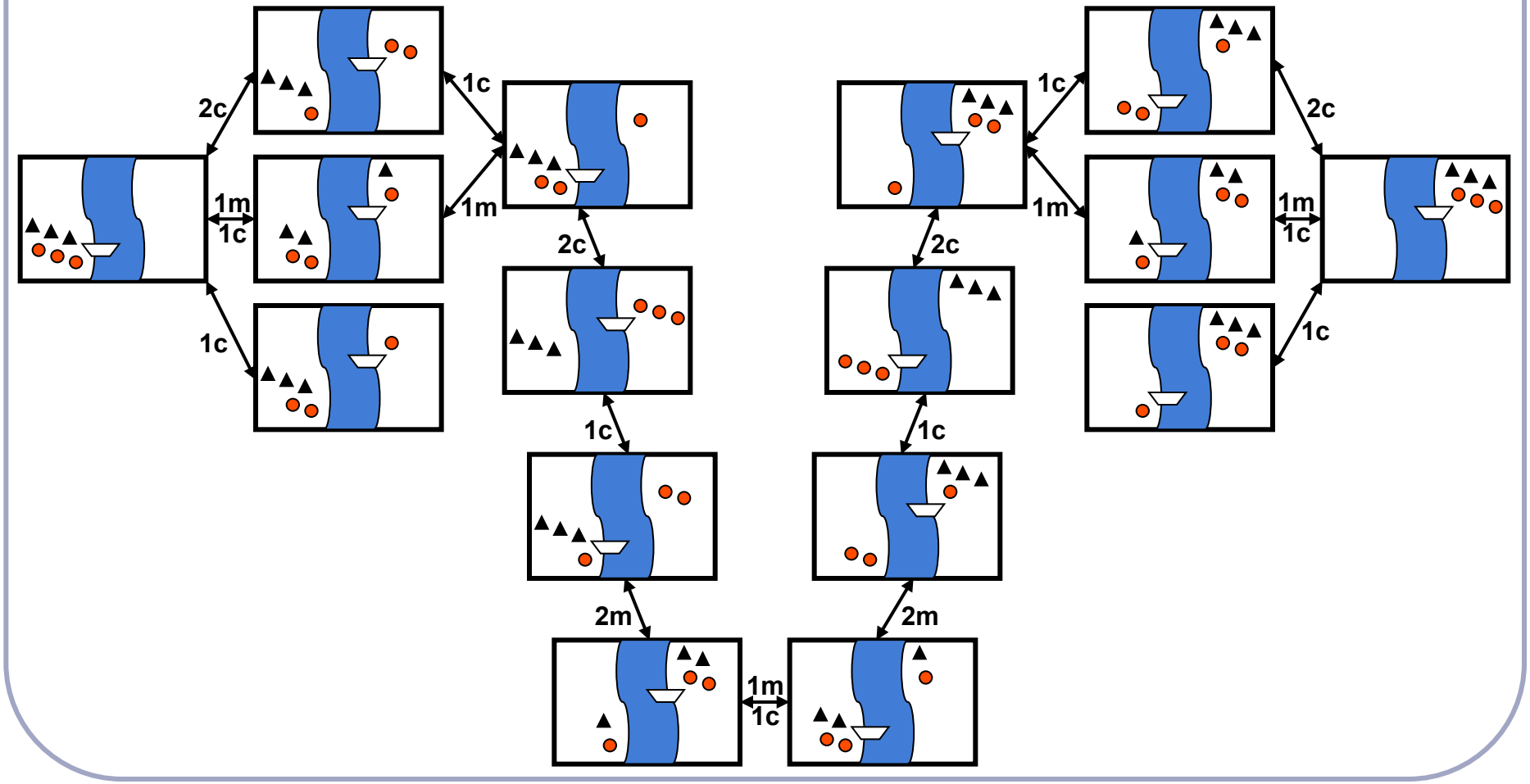
Conceptual Model for Planning: State-Transition Systems

- A state-transition system is a 4-tuple $\Sigma = (S,A,E,\gamma)$, where:
  - $S = \{s_1,s_2,\ldots\}$ is a finite or recursively enumerable set of states;
  - $A = \{a_1,a_2,\ldots\}$ is a finite or recursively enumerable set of actions;
  - $E = \{e_1,e_2,\ldots\}$ is a finite or recursively enumerable set of events; and
  - $\gamma: S\times(A\cup E)\rightarrow 2^S$ is a state transition function.
- if $a\in A$ and $\gamma(s,a) \neq \varnothing$ then a is applicable in s
- applying a in s will take the system to $s'\in\gamma(s,a)$

State-Transition Systems as Graphs

- A state-transition system $\Sigma = (S,A,E,\gamma)$ can be represented by a directed labelled graph $G = (N_G,E_G)$ where:
  - the nodes correspond to the states in S, i.e. $N_G=S$; and
  - there is an arc from $s \in N_G$ to $s' \in N_G$, i.e. $s \rightarrow s' \in E_G$, with label $u \in (A \cup E)$ if and only if $s' \in \gamma(s,a)$.

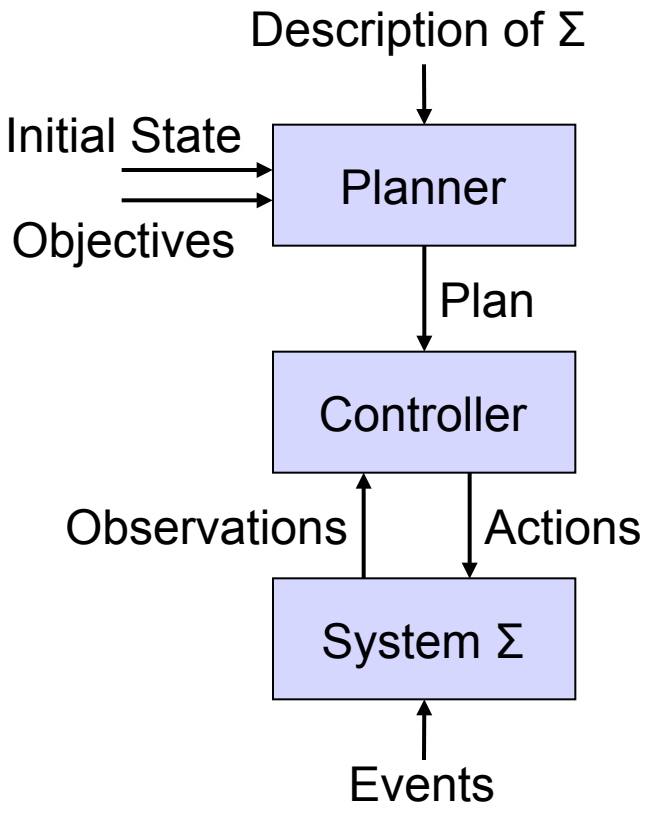
State-Transition Graph Example: Missionaries and Cannibals



Objectives and Plans

- state-transition system:
  - describes all ways in which a system may evolve
- plan:
  - a structure that gives appropriate actions to apply in order to achieve some objective when starting from a given state
- types of objective:
  - goal state s_g or set of goal states S_g
  - satisfy some conditions over the sequence of states
  - optimize utility function attached to states
  - task to be performed

Planning and Plan Execution



- planner:
  - given: description of Σ, initial state, objective
  - generate: plan that achieves objective
- controller:
  - given: plan, current state (observation function: $\eta : S \rightarrow O$)
  - generate: action
- state-transition system:
  - evolves as actions are executed and events occur

Overview

- What is AI Planning?
- A Conceptual Model for Planning
- Restricting Assumptions
- A Running Example: Dock-Worker Robots

A0: Finite Σ

- Assumption A0
  - system Σ has a finite set of states
- Relaxing A0
  - why?
    - to describe actions that construct or bring new objects into the world
    - to handle numerical state variables
  - issues:
    - decidability and termination of planners

A1: Fully Observable Σ

- Assumption A1
  - system Σ is fully observable, i.e. η is the identity function
- Relaxing A1
  - why?
    - to handle states in which not every aspect is or can be known
  - issues:
    - if $\eta(s)=o$, $\eta^{-1}(o)$ usually more than one state (ambiguity)
    - determining the successor state

A2: Deterministic Σ

- Assumption A2
  - system Σ is deterministic, i.e. for all $s \in S$, $u \in A \cup E$: $|\gamma(s,u)| \leq 1$
  - short form: $\gamma(s,u)=s'$ for $\gamma(s,u)=\{s'\}$
- Relaxing A2
  - why?
    - to plan with actions that may have multiple alternative outcomes
  - issues:
    - controller has to observe actual outcomes of actions
    - solution plan may include conditional and iterative constructs

A3: Static Σ

- Assumption A3
  - system Σ is static, i.e. $E=\emptyset$
  - short form: $\Sigma = (S,A,\gamma)$ for $\Sigma = (S,A,\emptyset,\gamma)$
- Relaxing A3
  - why?
    - to model a world in which events can occur
  - issues:
    - world becomes nondeterministic from the point of view of the planner (same issues)

A4: Restricted Goals

- Assumption A4
  - the planner handles only restricted goals that are given as an explicit goal state s_g or set of goal states S_g
- Relaxing A4
  - why?
    - to handle constraints on states and plans, utility functions, or tasks
  - issues:
    - representation and reasoning over constraints, utility, and tasks

A5: Sequential Plans

- Assumption A5
  - a solution plan is a linearly ordered finite sequence of actions
- Relaxing A5
  - why?
    - to handle dynamic systems (see A3: static Σ)
    - to create different types of plans
  - issues:
    - must not shift problem to the controller
    - reasoning about (more complex) data structures

A6: Implicit Time

- Assumption A6
  - actions and events have no duration in state transition systems
- Relaxing A6
  - why?
    - to handle action duration, concurrency, and deadlines
  - issues:
    - representation of and reasoning about time
    - controller must wait for effects of actions to occur

A7: Offline Planning

- Assumption A7
  - planner is not concerned with changes of Σ while it is planning
- Relaxing A7
  - why?
    - to drive a system towards some objectives
  - issues:
    - check whether the current plan remains valid
    - if needed, revise current plan or re-plan

The Restricted Model

- restricted model: make assumptions A0-A7

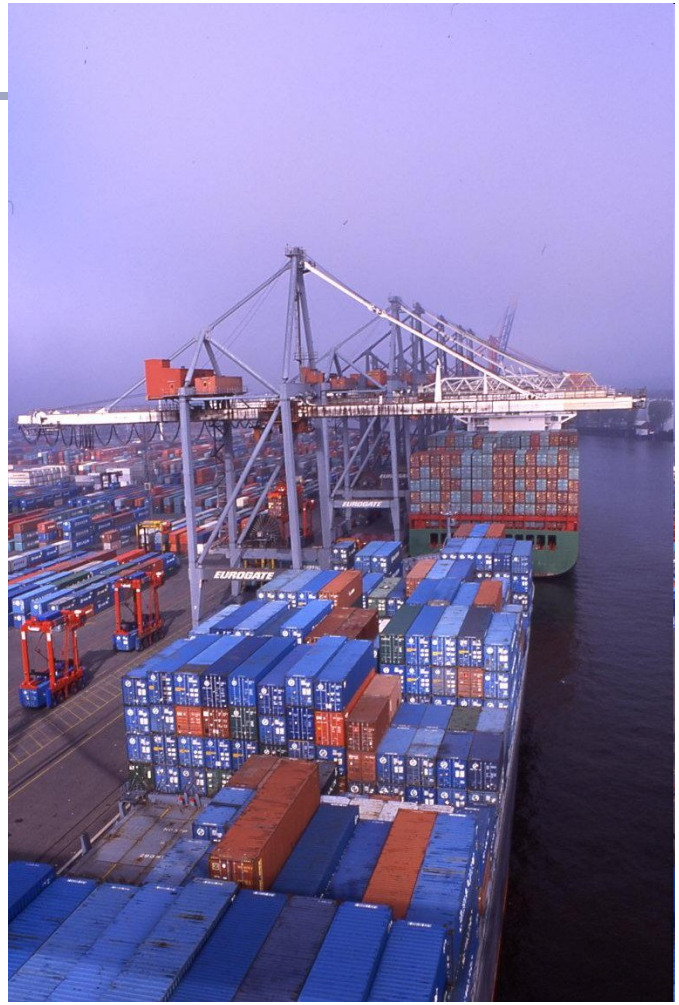
- Given a planning problem $\mathcal{P}=(\Sigma,s_i,S_g)$ where
  - $\Sigma=(S,A,\gamma)$ is a state transition system,
  - $s_i \in S$ is the initial state, and
  - $S_g \subset S$ is a set of goal states,
- find a sequence of actions $\langle a_1,a_2,\ldots,a_k\rangle$
  - corresponding to a sequence of state transitions $\langle s_i,s_1,\ldots,s_k\rangle$ such that
  - $s_1=\gamma(s_i,a_1)$, $s_2=\gamma(s_1,a_2),\ldots,$ $s_k=\gamma(s_{k-1},a_k)$, and $s_k \in S_g$.

Overview

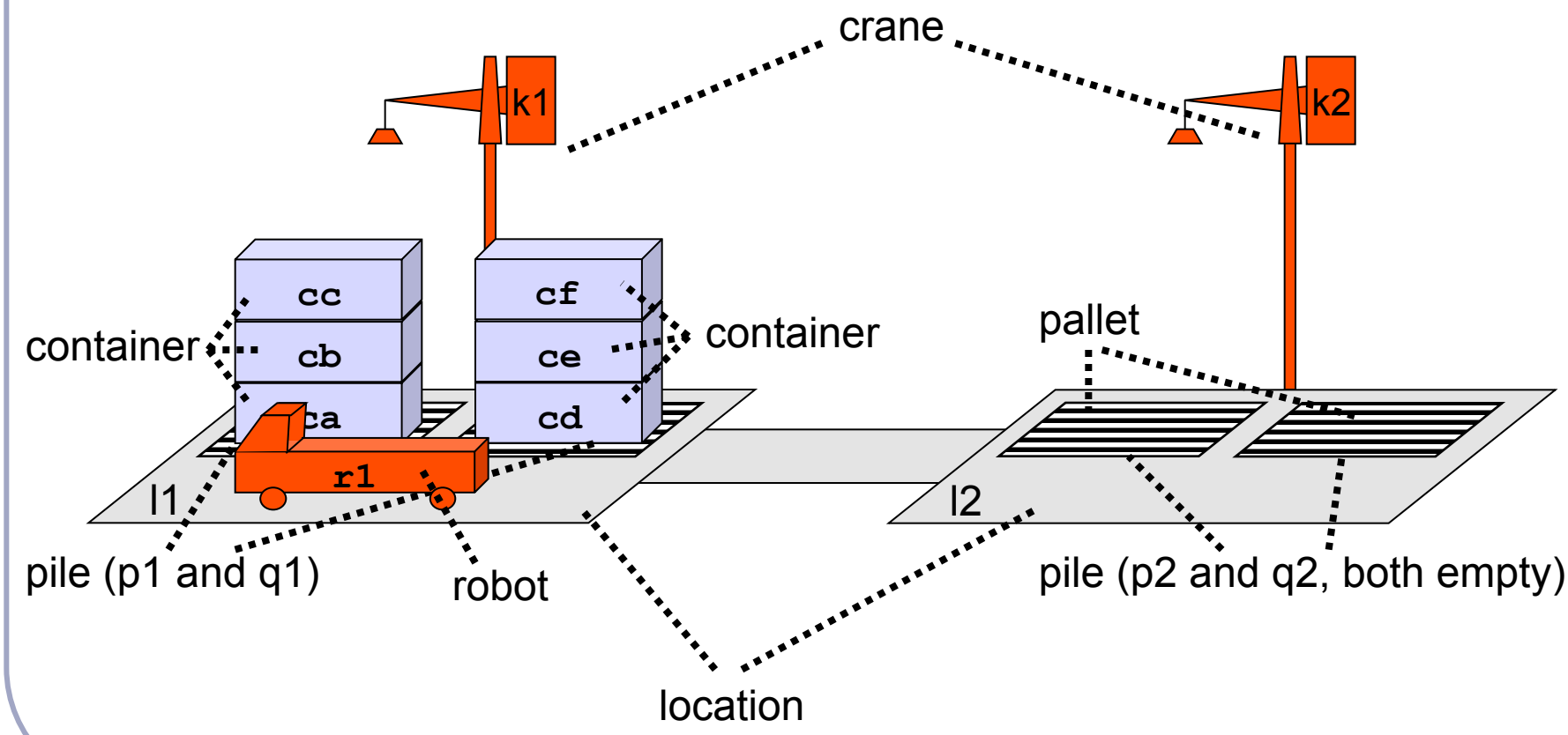
- What is AI Planning?
- A Conceptual Model for Planning
- Restricting Assumptions
- A Running Example: Dock-Worker Robots

The Dock-Worker Robots (DWR) Domain

- aim: have one example to illustrate planning procedures and techniques
- informal description:
  - harbour with several locations (docks), docked ships, storage areas for containers, and parking areas for trucks and trains
  - cranes to load and unload ships etc., and robot carts to move containers around



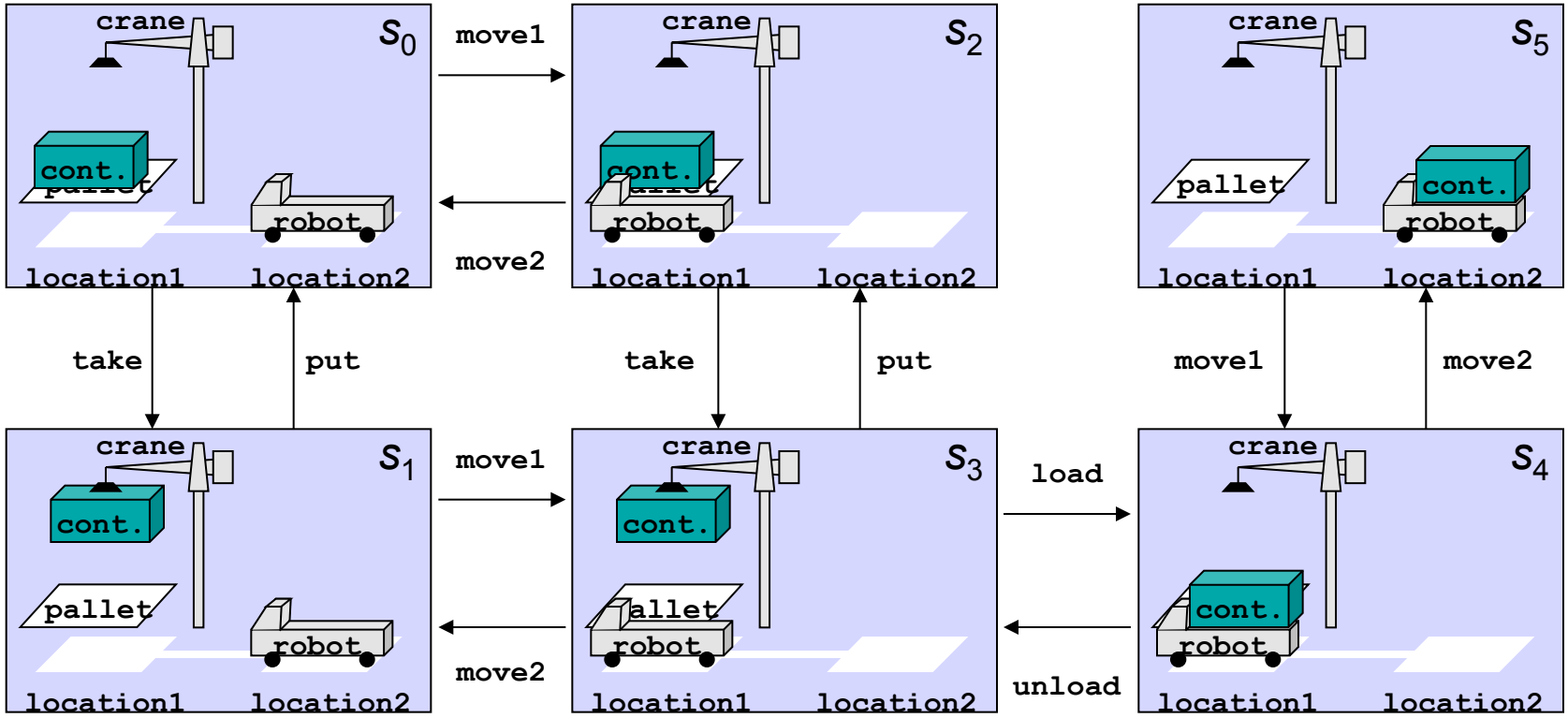
DWR Example State



Actions in the DWR Domain

- `move` robot *r* from location *l* to some adjacent and unoccupied location *l'*
- `take` container *c* with empty crane *k* from the top of pile *p*, all located at the same location *l*
- `put` down container *c* held by crane *k* on top of pile *p*, all located at location *l*
- `load` container *c* held by crane *k* onto unloaded robot *r*, all located at location *l*
- `unload` container *c* with empty crane *k* from loaded robot *r*, all located at location *l*

State-Transition Systems: Graph Example



State-Space Search and the STRIPS Planner

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

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

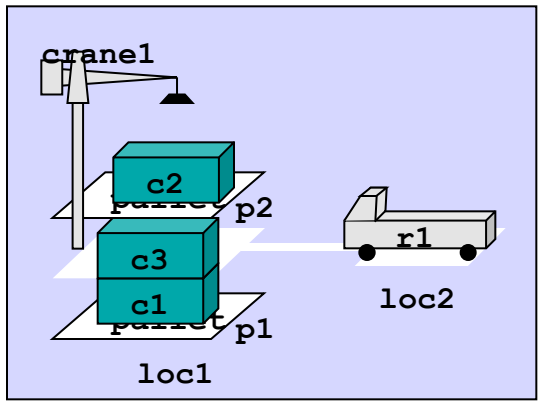
- A restricted state-transition system is a triple $\Sigma=(S,A,\gamma)$, where:
  - $S=\{s_1,s_2,\ldots\}$ is a set of states;
  - $A=\{a_1,a_2,\ldots\}$ is a set of actions;
  - $\gamma: S\times A\rightarrow S$ is a state transition function.
- defining STRIPS planning domains:
  - define STRIPS states
  - 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 fluent or flexible relations.
  - example: at
- A state-invariant predicate is called a rigid relation.
  - 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.
  - L^+ is the set of atoms that are positive literals in L and
  - L^- is the set of all atoms whose negations are in L.
- Let a be an action and s a state. Then a is applicable 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:
  - $\gamma(s,a) = (s - \text{effects}^-(a)) \cup \text{effects}^+(a)$

STRIPS Planning Domains

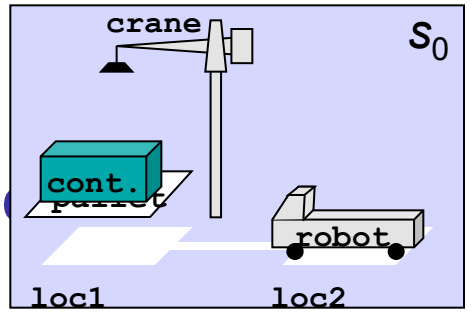
- Let $\mathcal{L}$ be a function-free first-order language. A STRIPS planning domain on $\mathcal{L}$ is a restricted state-transition system $\Sigma=(S,A,\gamma)$ such that:
  - S is a set of STRIPS states, i.e. sets of ground atoms
  - A is a set of ground instances of some STRIPS planning operators O
  - $\gamma: S \times A \rightarrow S$ 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

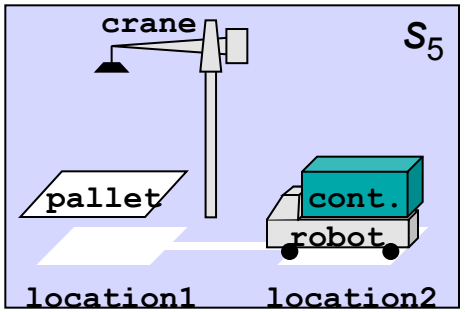
STRIPS Planning Problems

- 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), a (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 $\Sigma=(S,A,\gamma)$ on $\mathcal{L}$
  - 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 $\mathcal{P}$

Classical Plans

- A plan is any sequence of actions $\pi = \langle a_1, \ldots, a_k \rangle$, where $k \geq 0$.
  - 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 $\mathcal{P}=(\Sigma,s_i,g)$ be a planning problem. A plan π is a solution for $\mathcal{P}$ if $\gamma(s_i,\pi)$ satisfies g.
  - A solution π is redundant if there is a proper subsequence of π is also a solution for $\mathcal{P}$.
  - π is minimal 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

Overview

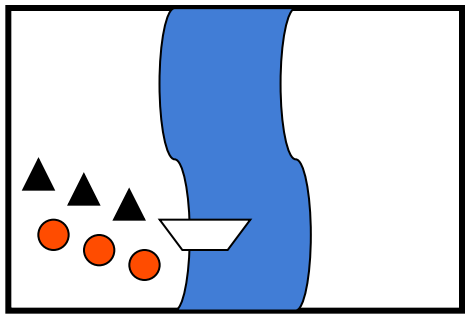
- The STRIPS Representation
- The Planning Domain Definition Language (PDDL)
- Problem-Solving by Search
- Heuristic Search
- Forward State-Space Search
- Backward State-Space Search
- The STRIPS Planner

Search Problems

- 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

Missionaries and Cannibals: Initial State and Actions

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

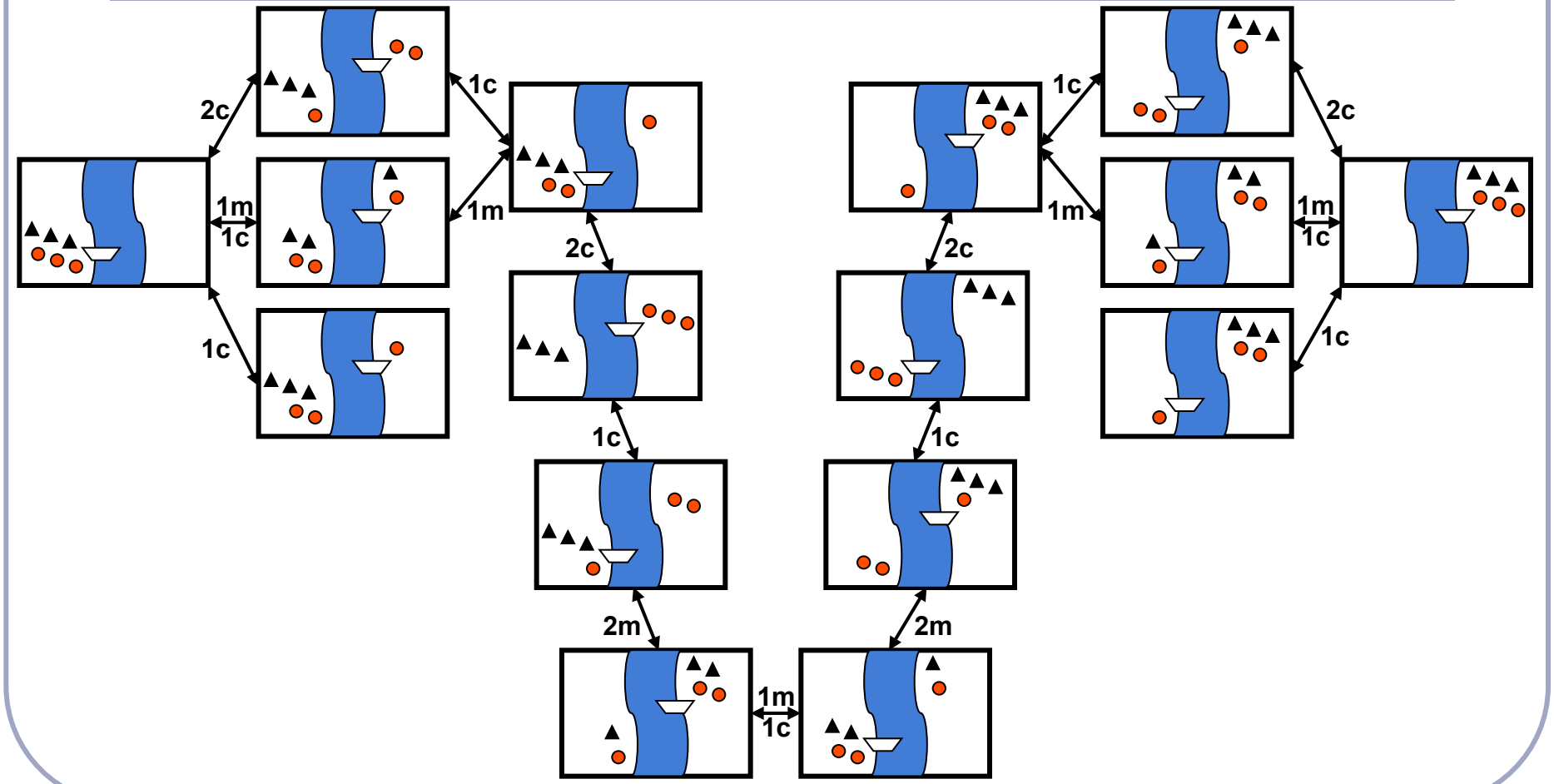


- 5 possible actions:
  - one missionary crossing
  - one cannibal crossing
  - two missionaries crossing
  - two cannibals crossing
  - one missionary and one cannibal crossing

Missionaries and Cannibals: Successor Function

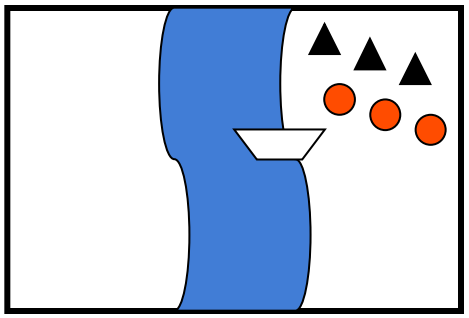
state	set of <*action*, *state*>
(L:3m,3c,b-R:0m,0c) →	{<2c, (L:3m,1c-R:0m,2c,b)>, <1m1c, (L:2m,2c-R:1m,1c,b)>, <1c, (L:3m,2c-R:0m,1c,b)>}
(L:3m,1c-R:0m,2c,b) →	{<2c, (L:3m,3c,b-R:0m,0c)>, <1c, (L:3m,2c,b-R:0m,1c)>}
(L:2m,2c-R:1m,1c,b) →	{<1m1c, (L:3m,3c,b-R:0m,0c)>, <1m, (L:3m,2c,b-R:0m,1c)>}
⋮	⋮

Missionaries and Cannibals: State Space



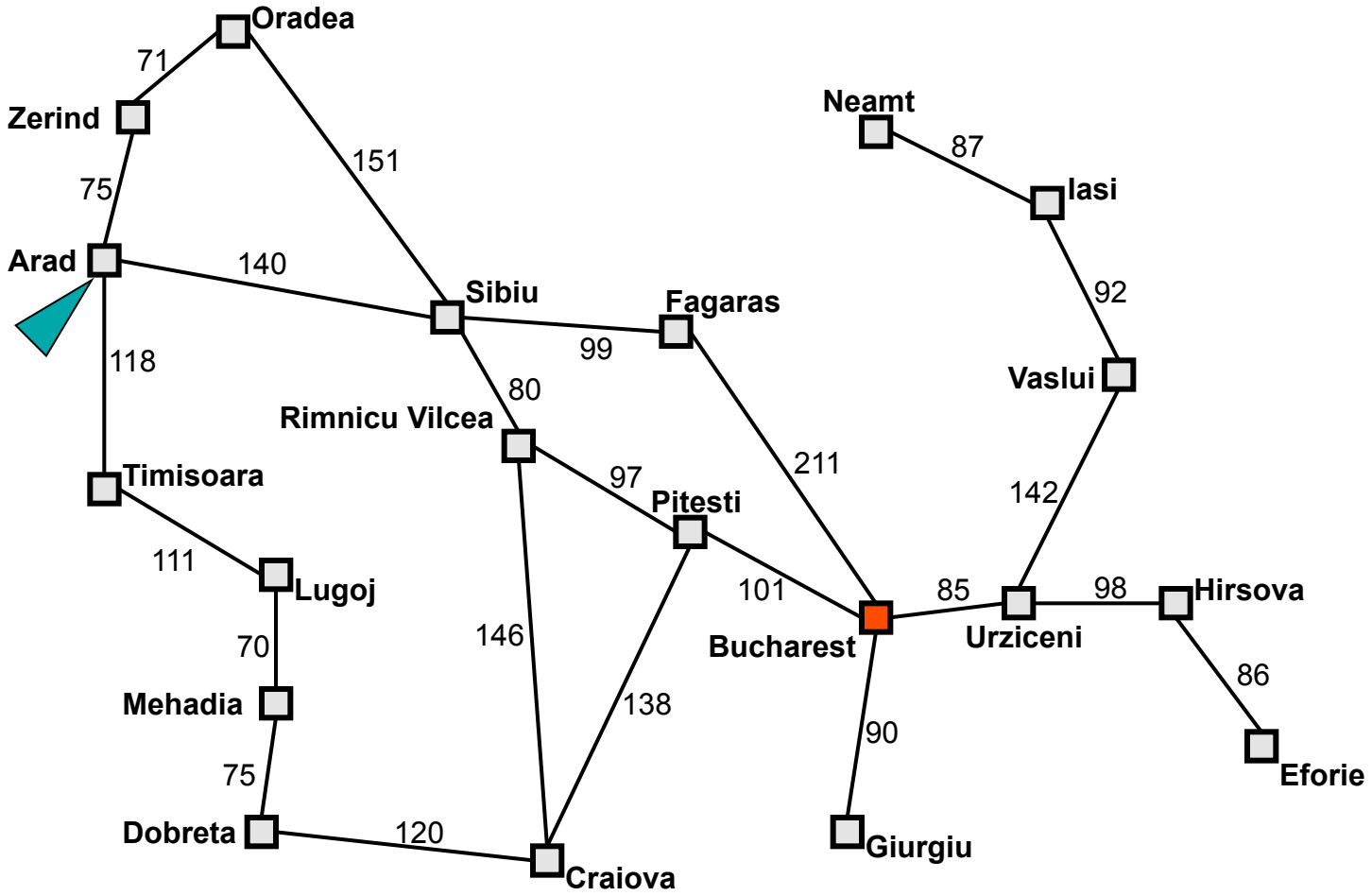
Missionaries and Cannibals: Goal State and Path Cost

- 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

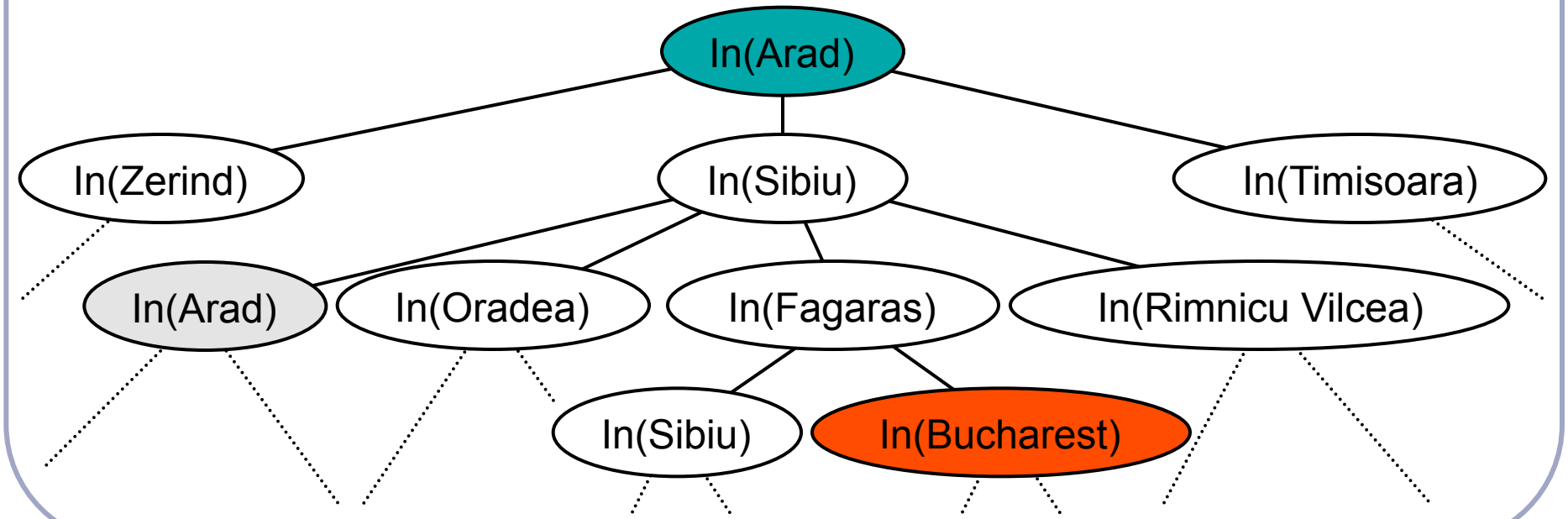


Touring Romania: Search Problem Definition

- 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):

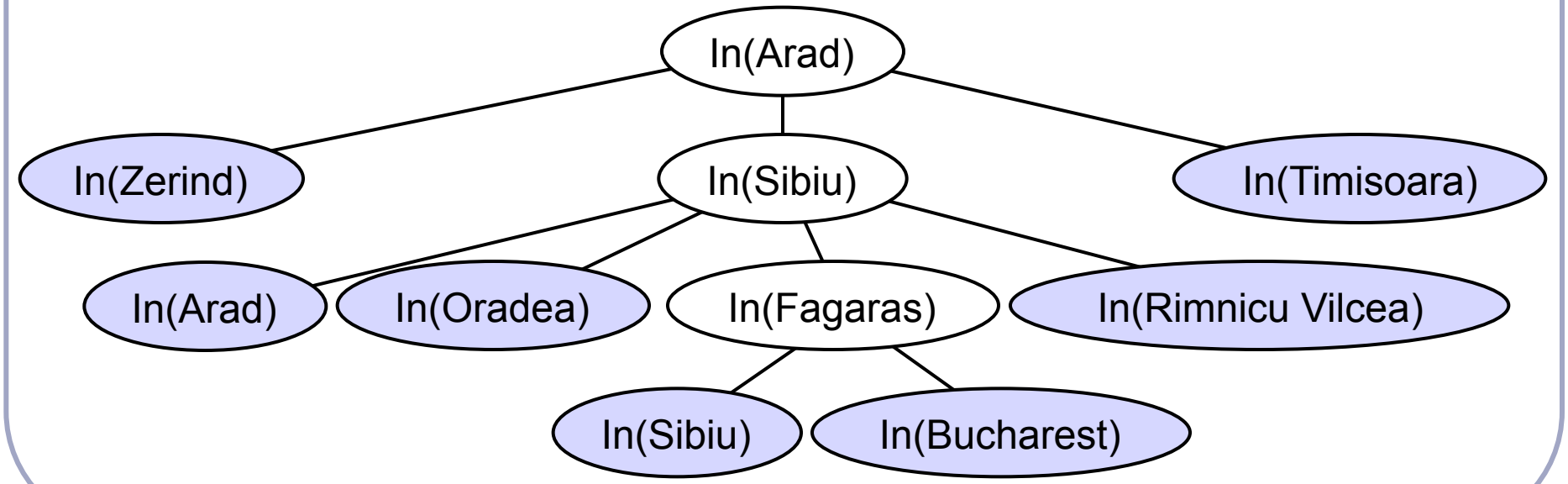


Search Nodes

- 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



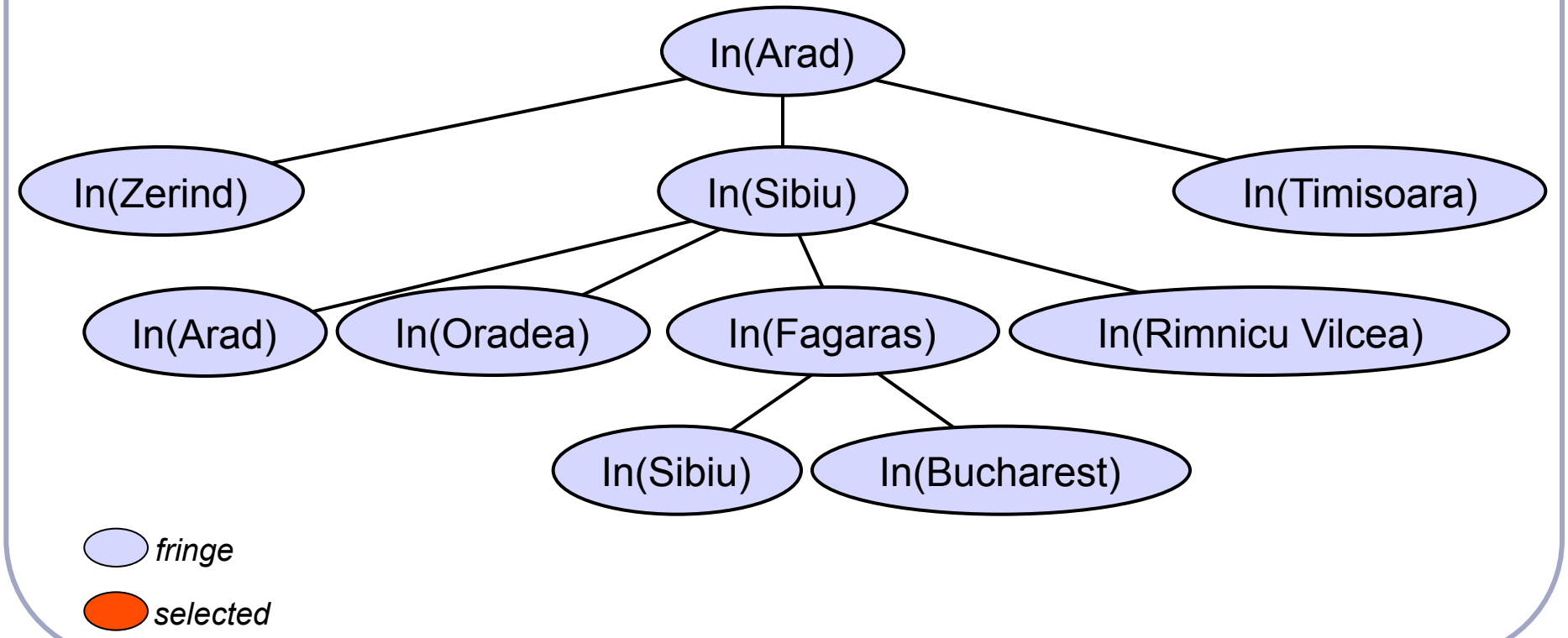
Search (Control) Strategy

- 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)
```

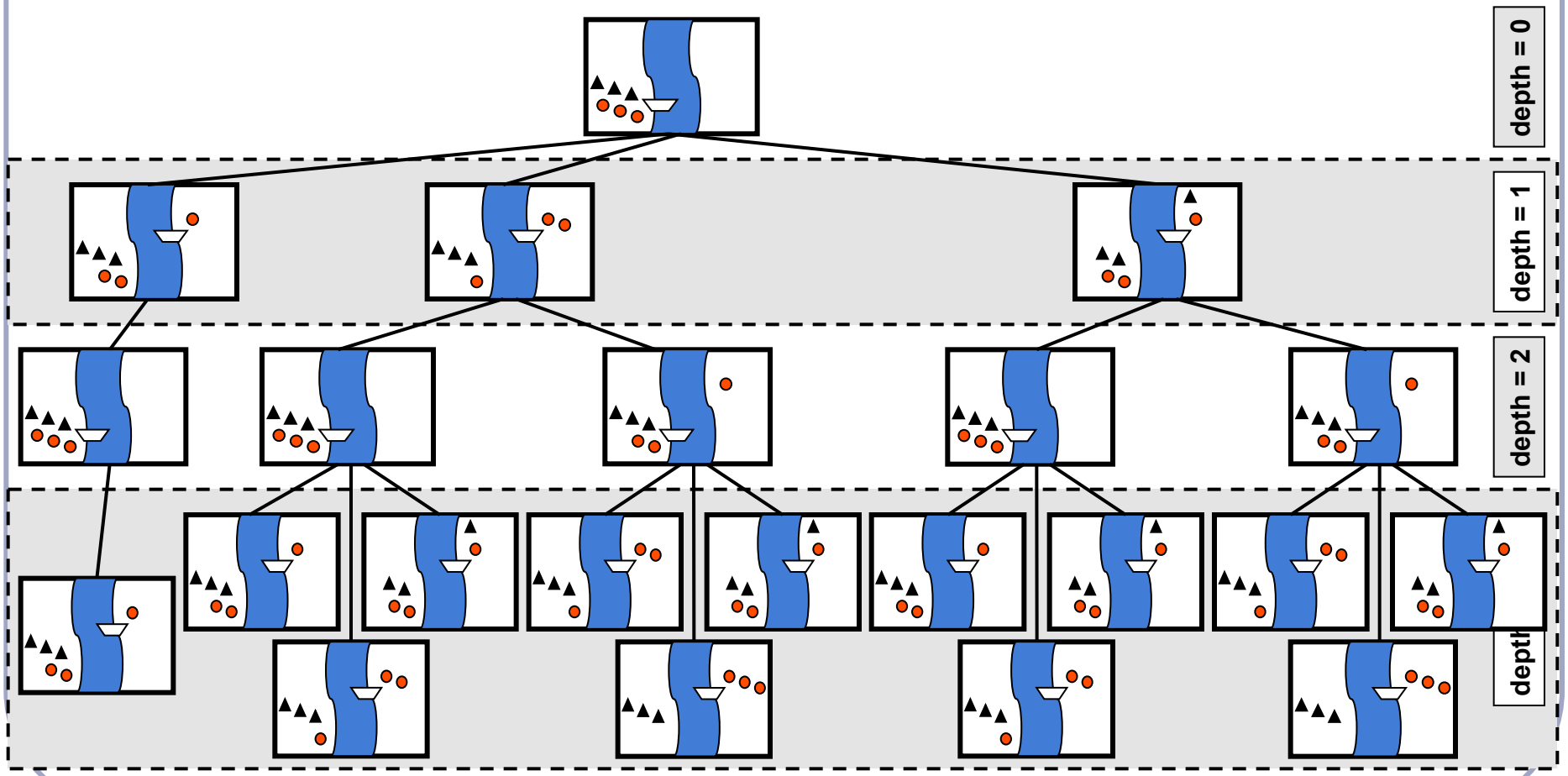
General Search Algorithm: Touring Romania Example



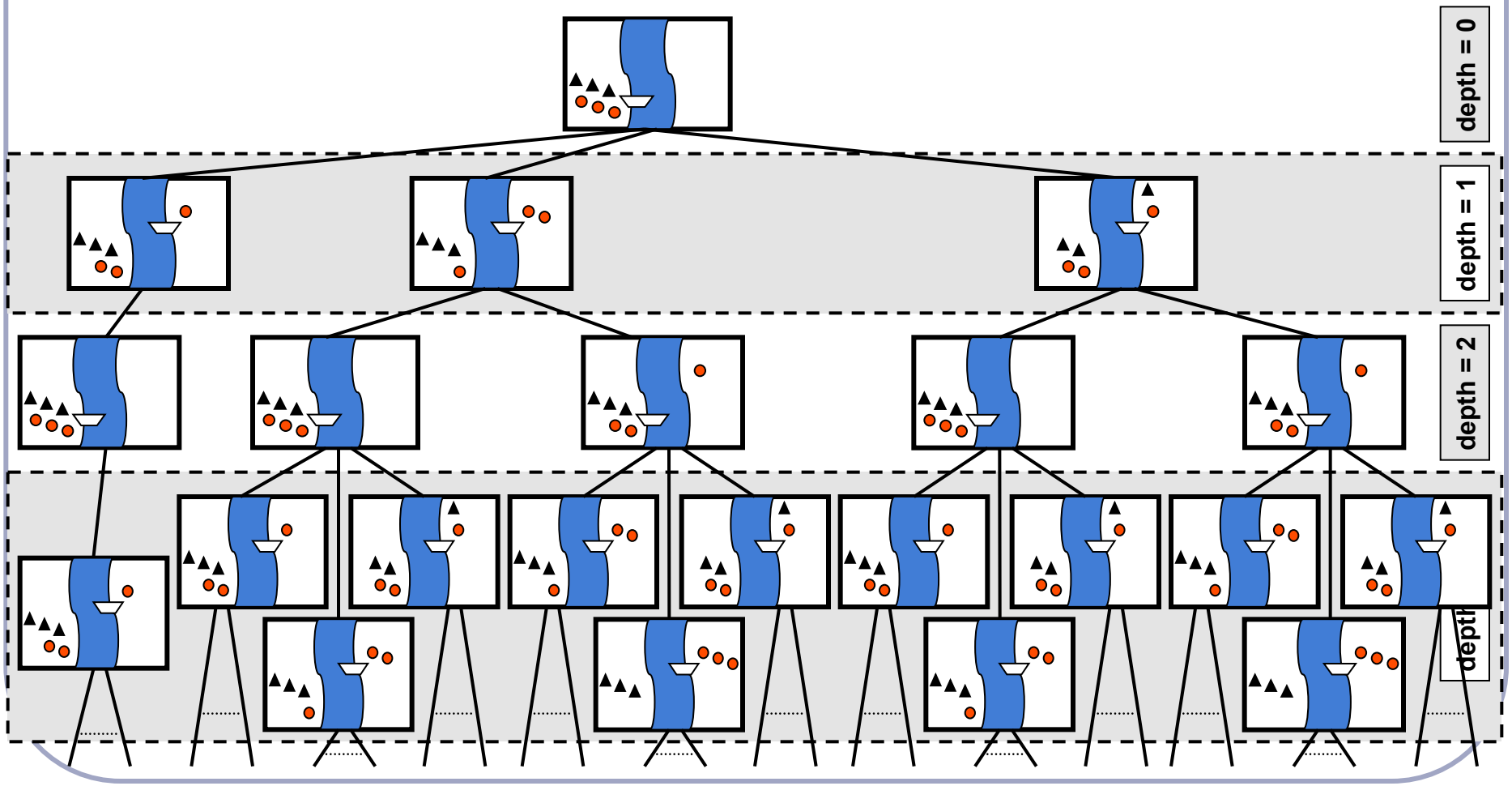
Uninformed vs. Informed Search

- 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*:

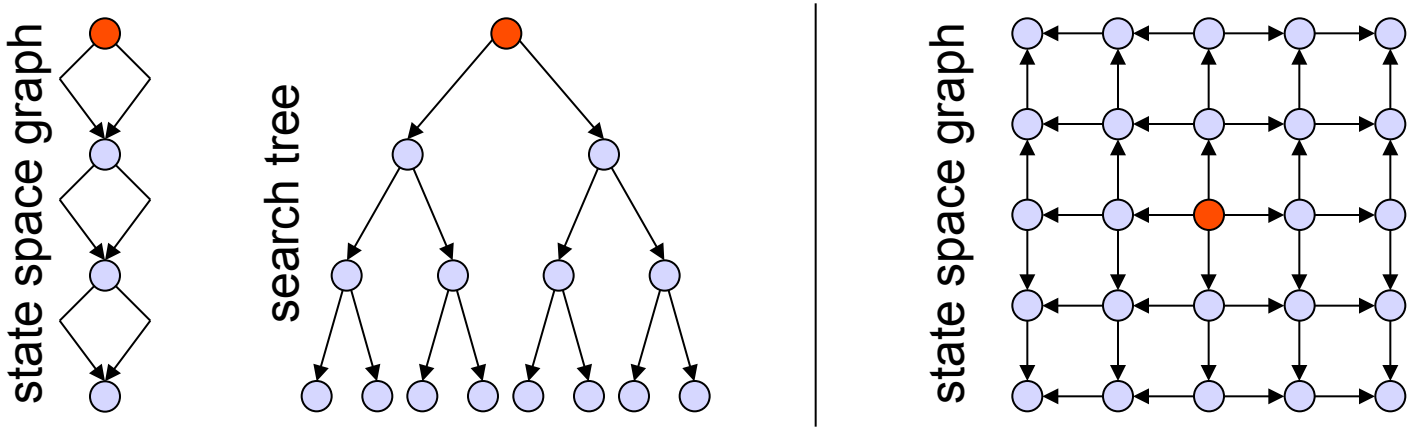
  for *depth* ← 0 **to** ∞ **do**

    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

Uniform-Cost 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”

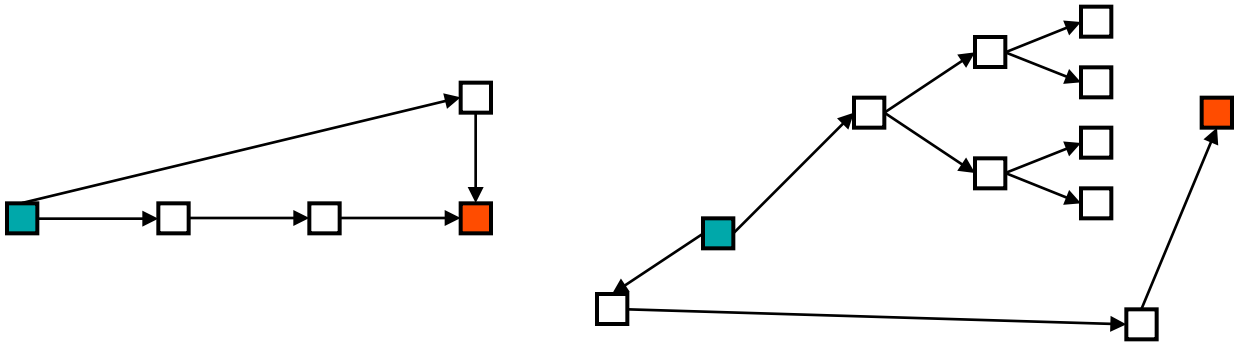
Touring in Romania: Heuristic

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

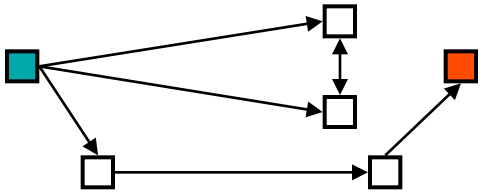
Arad	366	Hirsova	151	Rimnicu Vilcea	193
Bucharest	0	Iasi	226		
Craiova	160	Lugoj	244	Sibiu	253
Dobreta	242	Mehadia	241	Timisoara	329
Eforie	161	Neamt	234	Urziceni	80
Fagaras	176	Oradea	380	Vaslui	199
Giurgiu	77	Pitesti	100	Zerind	374

Greediness

- greediness is susceptible to false starts



- repeated states may lead to infinite oscillation



A* Search

- Uniform-cost 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

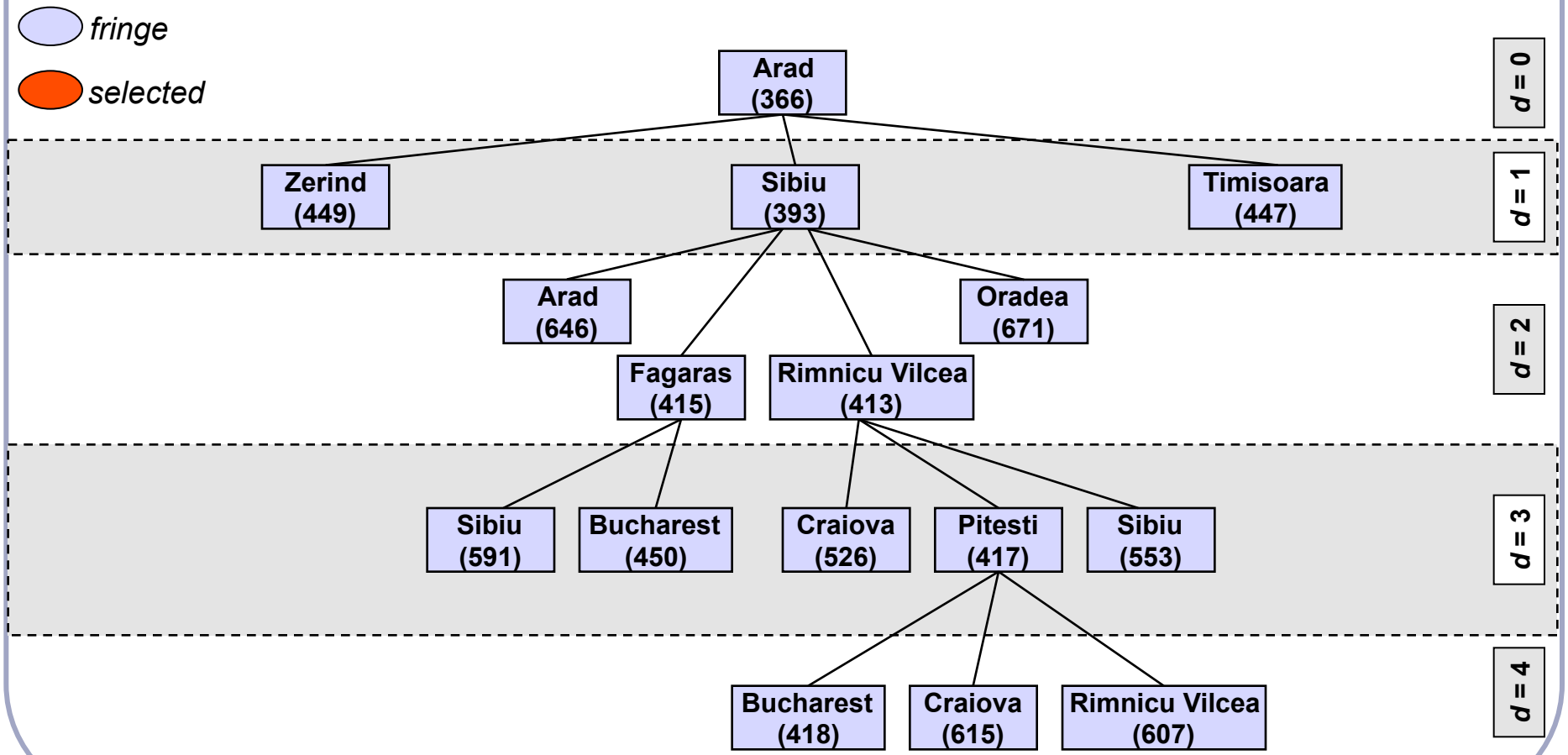
- A* search is optimal if $h(n)$ is admissible

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* 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

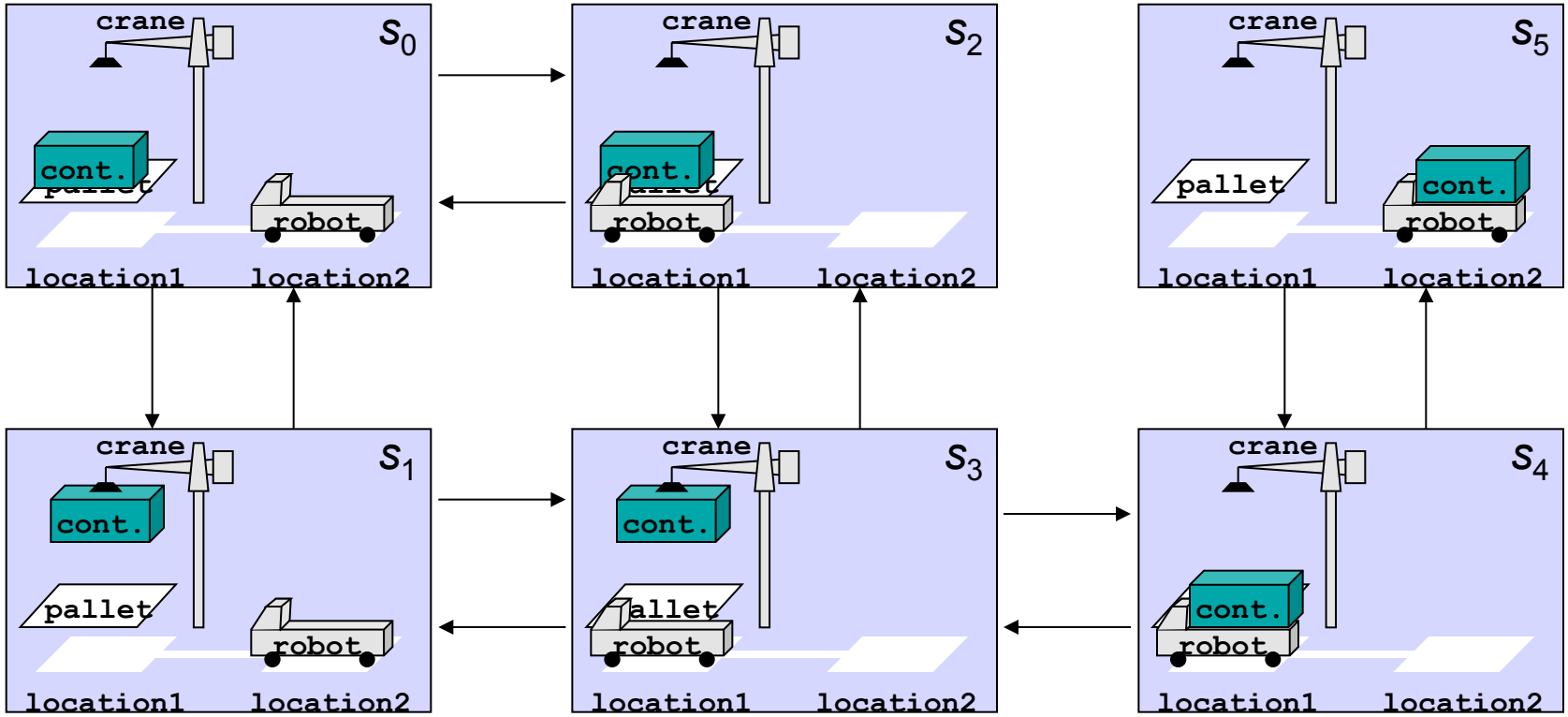
Overview

- 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 $s \in S$
  - $\Gamma(\{s_1,\ldots,s_n\}) = \bigcup_{(k\in[1,n])} \Gamma(s_k)$
  - $\Gamma^0(\{s_1,\ldots,s_n\}) = \{s_1,\ldots,s_n\}$
  - $\Gamma^m(\{s_1,\ldots,s_n\}) = \Gamma(\Gamma^{m-1}(\{s_1,\ldots,s_n\}))$

  } $s_1,\ldots,s_n \in S$
- The transitive closure of Γ defines the set of all reachable states:
  - $\Gamma^>(s) = \bigcup_{(k\in[0,\infty])} \Gamma^k(\{s\})$ for $s \in S$

Solution Existence

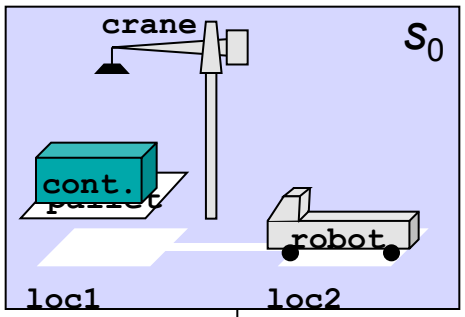
- **Proposition**: A STRIPS planning problem $\mathcal{P}=(\Sigma,s_i,g)$ (and a statement of such a problem $P=(O,s_i,g)$) has a solution iff $S_g \cap \Gamma^{>}(\{s_i\}) \neq \{\}$.

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 ← γ(*state*,*action*)
        plan ← plan • 〈*action*〉

DWR Example: Forward Search

initial state:



plan =

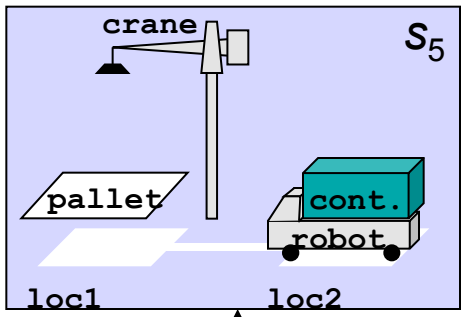
take(crane,loc1,cont,pallet,pile)

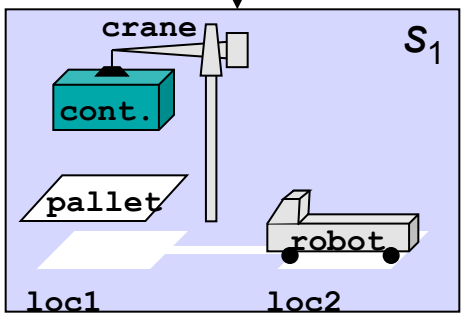
move(robot,loc2,loc1)

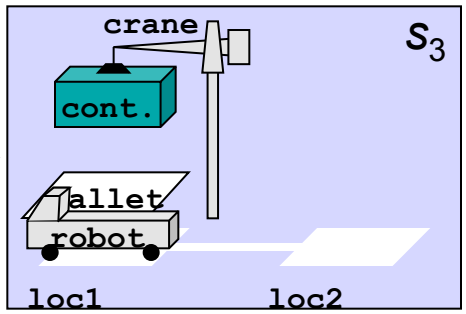
load(crane,loc1,cont,robot)

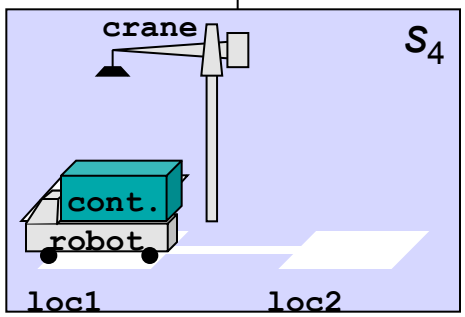
move(robot,loc1,loc2)

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*=$\gamma(s_i$,*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:
  - $\Gamma^{-1}(g)=\{\gamma^{-1}(g,a) \mid a\in A \text{ is relevant for } g\}$ for $g\in 2^L$
  - $\Gamma^{0}(\{g_1,\ldots,g_n\})= \{g_1,\ldots,g_n\}$
  - $\Gamma^{-1}(\{g_1,\ldots,g_n\})= \bigcup_{(k\in[1,n])}\Gamma^{-1}(g_k)$
  - $\Gamma^{-m}(\{g_1,\ldots,g_n\})= \Gamma^{-1}(\Gamma^{-(m-1)}(\{g_1,\ldots,g_n\}))$

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

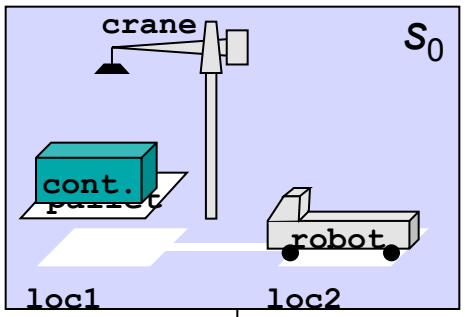
- **Proposition**: A propositional planning problem $\mathcal{P}=(\Sigma,s_i,g)$ (and a statement of such a problem $P=(O,s_i,g)$) has a solution iff $\exists s \in \Gamma^{<}(\{g\})$: s_i satisfies s.

Ground Backward State-Space Search Algorithm

function groundBwdSearch(O,s_i,g)
    subgoal ← g
    plan ← ⟨⟩
    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()
        subgoal ← γ^{-1}(*subgoal*, *action*)
        plan ← ⟨*action*⟩ • *plan*

DWR Example: Backward Search

initial state:



plan =

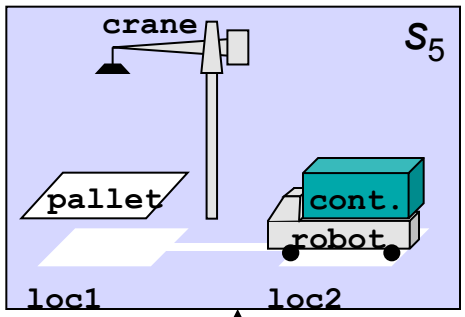
take(crane,loc1,cont,pallet,pile)

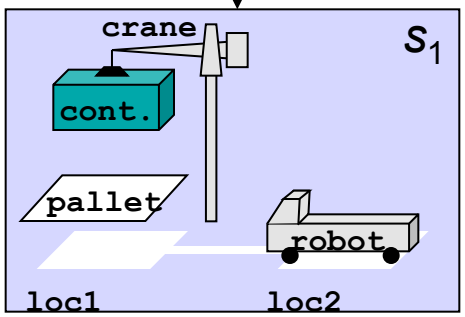
move(robot,loc2,loc1)

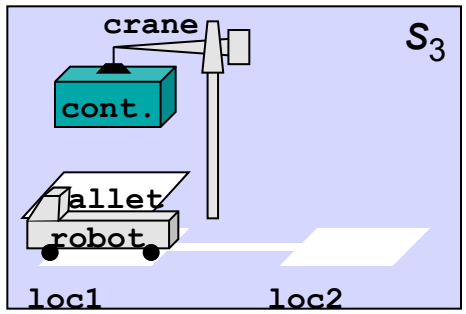
load(crane,loc1,cont,robot)

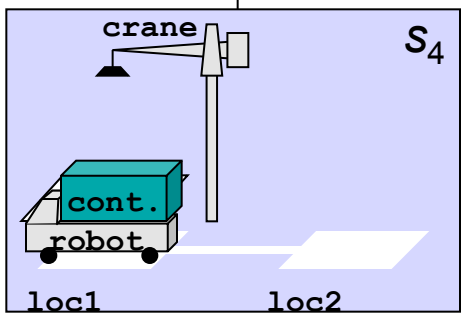
move(robot,loc1,loc2)

goal state:









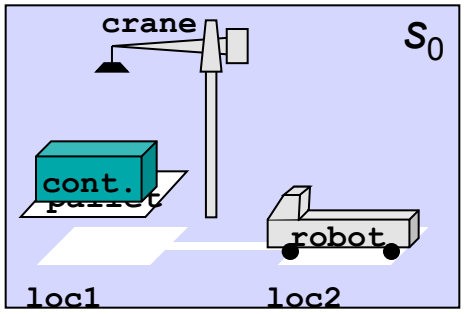
Example: Regression with Operators

- 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 ← g
    plan ← 〈〉
    loop
        if ∃σ:s_i.satisfies(σ(*subgoal*)) **then return** σ(*plan*)
        applicables ←
            {(o,σ) | o∈O and σ(o) relevant for *subgoal*}
        if *applicables*.isEmpty() **then return** failure
        action ← *applicables*.chooseOne()
        subgoal ← γ^{-1}(σ(*subgoal*), σ(o))
        plan ← σ(〈*action*〉) • σ(*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)})

- ∃σ:s_i.satisfies(σ(*subgoal*)): no
- *applicables* = {(move(r_1,l_1,m_1),{r_1←robot, m_1←loc1})}
- *subgoal* = {adjacent(l_1,loc1), at(robot,l_1), ¬occupied(loc1)}
- *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) $subgaol=\gamma^{-1}(g,plan)$ 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

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

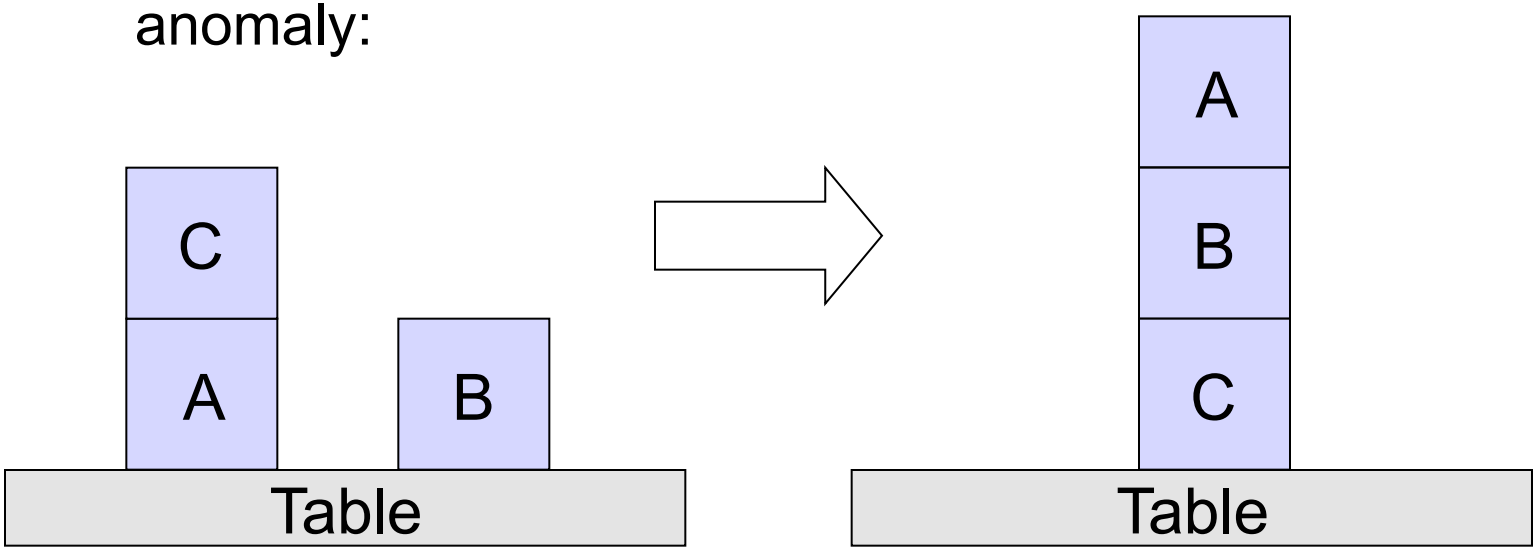
- state space still too large to search efficiently
- STRIPS idea:
  - only work on preconditions of the last operator added to the plan
  - if the current state satisfies all of an operator's preconditions, commit to this operator

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 ← γ(s, subplan • 〈action〉)
        plan ← plan • subplan • 〈action〉
```

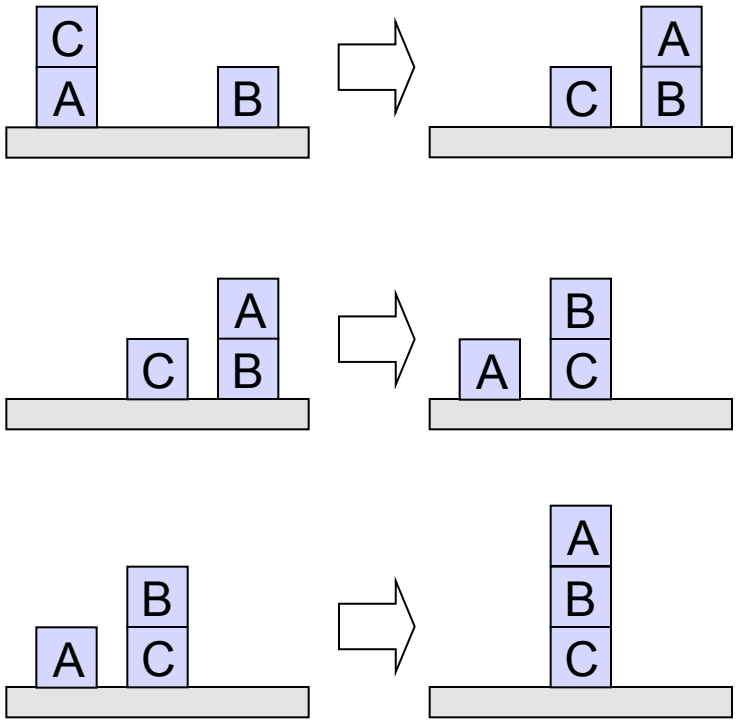
Problems with STRIPS

- 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:



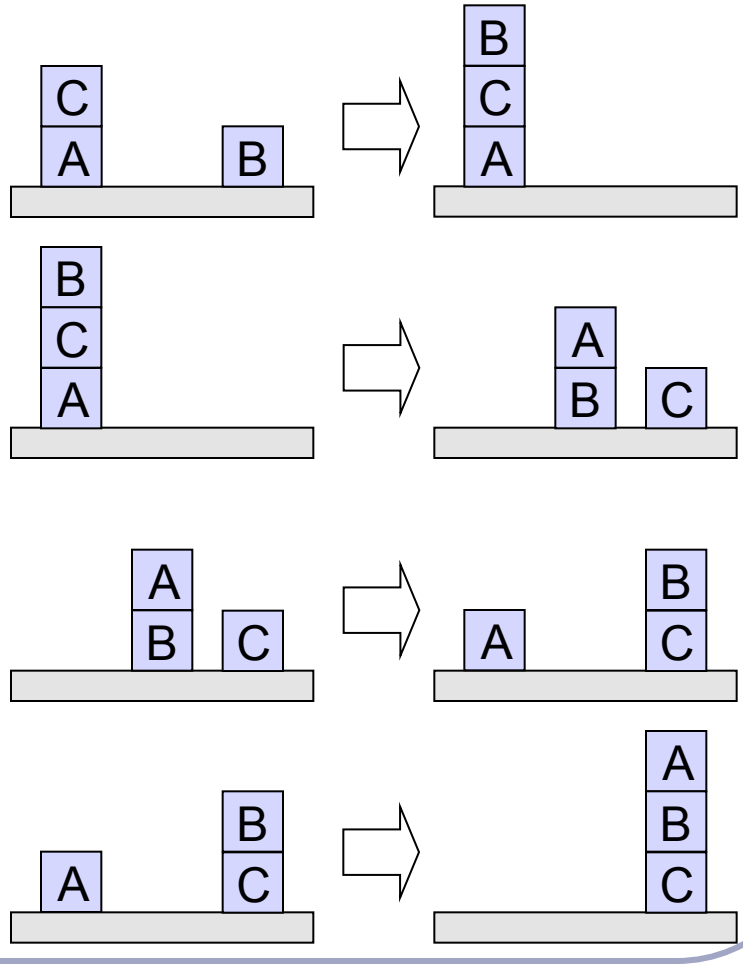
STRIPS and the Sussman Anomaly (1)

- 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