Chapter 2
Deliberation with Deterministic Models

2.1: State-Variable Representation
2.2: Forward Search
2.6: Planning and Acting

Dana S. Nau
University of Maryland
Motivation and Outline

- How to model a complex environment?
  - Generally need simplifying assumptions

- **Classical planning**
  - Finite, static world, just one actor
  - No concurrent actions, no explicit time
  - Determinism, no uncertainty
  - Sequence of states and actions \( \langle s_0, a_1, s_1, a_2, s_2, \ldots \rangle \)

- Avoids many complications

- Most real-world environments don’t satisfy the assumptions
  \( \Rightarrow \) Errors in prediction

- OK if they’re infrequent and don’t have severe consequences

Outline

2.1 State-variable representation
  - state variables, states, actions, plans

2.2 Forward state-space search

2.3 Heuristic functions

2.4 Backward search

2.5 Plan-space search

2.6 Incorporating planning into an actor
Domain Model

State-transition system
or classical planning domain:

- \( \Sigma = (S,A,\gamma,\text{cost}) \) or \( (S,A,\gamma) \)
  - \( S \) - finite set of states
  - \( A \) - finite set of actions
  - \( \gamma: S \times A \rightarrow S \)
    - prediction (or state-transition) function
  - partial function
    - defined only when \( a \) is applicable in \( s \)
  - Domain(\( a \)) = \{s \in S \mid a \text{ is applicable in } s\}
    = \{s \in S \mid \gamma(s,a) \text{ is defined}\}
  - Range(\( a \)) = \{\gamma(s,a) \mid s \in \text{Domain}(a)\}
  - cost: \( S \times A \rightarrow \mathbb{R}^+ \) or cost: \( A \rightarrow \mathbb{R}^+ \)
    - optional; default is cost(\( a \)) = 1
    - money, time, something else

- plan:
  - a sequence of actions \( \pi = \langle a_1, \ldots, a_n \rangle \)
  - \( \pi \) is applicable in \( s_0 \) if the actions are applicable in the order given
    - \( \gamma(s_0, a_1) = s_1 \)
    - \( \gamma(s_1, a_2) = s_2 \)
    - \( \ldots \)
    - \( \gamma(s_{n-1}, a_n) = s_n \)
  - In this case define \( \gamma(s_0, \pi) = s_n \)

- Classical planning problem:
  - \( P = (\Sigma, s_0, S_g) \)
    - planning domain, initial state, set of goal states

- Solution for \( P \):
  - a plan \( \pi \) such that that \( \gamma(s_0, \pi) \in S_g \)
Representing $\Sigma$

- If $S$ and $A$ are small enough
  - Give each state and action a name
  - For each $s$ and $a$, store $\gamma(s,a)$ in a lookup table

- In larger domains, don’t represent all states explicitly
  - Language for describing properties of states
  - Language for describing how each action changes those properties
  - Start with initial state
  - Use actions to produce other states

$$\begin{align*}
    s_0 &\xrightarrow{a_1} s_1 = \gamma(s_0,a_1) \\
    s_1 &\xrightarrow{a_1'} s_1' = \gamma(s_0,a_1')
\end{align*}$$
Domain-Specific Representation

- Made to order for a specific environment

- State: arbitrary data structure

- Action: (head, preconditions, effects, cost)
  - head: name and parameter list
    - Get actions by instantiating the parameters
  - preconditions:
    - Computational tests to predict whether an action can be performed
    - Should be necessary/sufficient for the action to run without error
  - effects:
    - Procedures that modify the current state
  - cost: procedure that returns a number
    - Can be omitted, default is cost ≡ 1
Example

- Drilling holes in a metal workpiece
  - A state
    - annotated geometric model of the workpiece
    - capabilities and status of drilling machine and drill bit
  - Several actions
    - put workpiece onto the drilling machine
    - clamp it
    - load a drill bit
    - drill

- Name and parameters:
  - drill-hole(machine, drill-bit, workpiece, geometry, machining-tolerances)

- Preconditions
  - Capabilities: can the machine and drill bit produce a hole with the desired geometry and machining tolerances?
  - Current state: Is the drill bit installed? Is the workpiece clamped onto the table? Etc.

- Effects
  - annotated geometric model of modified workpiece

- Cost
  - estimate of time or monetary cost
Discussion

● Advantage of domain-specific representation:
  ▶ use whatever works best for that particular domain

● Disadvantage:
  ▶ for each new domain, need new representation and deliberation algorithms

● Alternative: *domain-independent* representation
  ▶ Try to create a “standard format” that can be used for many different planning domains
  ▶ Deliberation algorithms that work for anything in this format

● *State-variable* representation
  ▶ Simple formats for describing states and actions
  ▶ Limited representational capability
    • But easy to compute, easy to reason about
  ▶ Domain-independent search algorithms and heuristic functions that can be used in all state-variable planning problems
State-Variable Representation

- $E$: environment that we want to represent
- $B$: set of symbols called objects
  - names for objects in $E$, mathematical constants, …

Example
  - $B = \text{Robots} \cup \text{Containers} \cup \text{Locs} \cup \{\text{nil}\}$
    - $\text{Robots} = \{r_1\}$
    - $\text{Containers} = \{c_1, c_2\}$
    - $\text{Locs} = \{d_1, d_2, d_3\}$

- $B$ only needs to include objects that matter at the current level of abstraction
- Can omit lots of details
  - physical characteristics of robots, containers, loading docks, roads, …
Properties of Objects

- Define ways to represent properties of objects
  - Two kinds of properties: *rigid* and *varying*

- *Rigid* property: stays the same in every state
  - Two equivalent notations:
    - A mathematical relation
      \[ \text{adjacent} = \{ (d1,d2), (d2,d1), (d1,d3), (d3,d1) \} \]
    - A set of *ground atoms*
      \[ \text{adjacent}(d1,d2), \text{adjacent}(d2,d1), \text{adjacent}(d1,d3), \text{adjacent}(d3,d1) \]

- Terminology from first-order logic:
  - *ground*: fully instantiated, no variable symbols
  - *atom* ≡ *atomic formula* ≡ *positive literal* ≡ predicate symbol with list of arguments
  - *negative literal* ≡ *negated atom* ≡ atom with a negation sign in front of it
Varying Properties

- *Varying* property (or *fluent*): may differ in different states
  - Represent it using a *state variable* that we can assign a value to

- Set of state variables
  \[ X = \{ \text{loc}(r1), \text{loc}(c1), \text{loc}(c2), \text{cargo}(r1) \} \]

- Each state variable \( x \in X \) has a *range* = \{all values that can be assigned to \( x \)\}
  - \( \text{Range} (\text{loc}(r1)) = \text{Locs} \)
  - \( \text{Range} (\text{loc}(c1)) = \text{Range} (\text{loc}(c2)) = \text{Robots} \cup \text{Locs} \)
  - \( \text{Range} (\text{cargo}(r1)) = \text{Containers} \cup \{\text{nil}\} \)

Instead of "domain", to avoid confusion with planning domains.
States as Functions

- Represent each state as a *variable-assignment function*
  - Function that maps each \( x \in X \) to a value in \( \text{Range}(x) \)

\[
\begin{align*}
  s_1(\text{loc}(r_1)) &= d_1, & s_1(\text{cargo}(r_1)) &= \text{nil}, \\
  s_1(\text{loc}(c_1)) &= d_1, & s_1(\text{loc}(c_2)) &= d_2
\end{align*}
\]

- Mathematically, a function is a set of ordered pairs
  \[
  s_1 = \{ \text{loc}(r_1), d_1, \text{cargo}(r_1), \text{nil}, \text{loc}(c_1), d_1, \text{loc}(c_2), d_2 \} 
  \]

- Write it as a set of *ground positive literals* (or *ground atoms*):
  \[
  s_1 = \{ \text{loc}(r_1) = d_1, \text{cargo}(r_1) = \text{nil}, \text{loc}(c_1) = d_1, \text{loc}(c_2) = d_2 \} 
  \]
Action Templates

- Action template: a parameterized set of actions
  \[ \alpha = (\text{head}(\alpha), \text{pre}(\alpha), \text{eff}(\alpha), \text{cost}(\alpha)) \]

- head(\alpha): name, parameters
  Each parameter has a range \( \subseteq B \)

- pre(\alpha): precondition literals
  \[ \text{rel}(t_1, \ldots, t_k), \quad \text{var}(t_1, \ldots, t_k) = t_0, \]
  \[ -\text{rel}(t_1, \ldots, t_k), -\text{var}(t_1, \ldots, t_k) = t_0 \]
  - Each \( t_i \) is a parameter or an element of \( B \)

- eff(\alpha): effect literals
  \[ \text{var}(t_1, \ldots, t_k) \leftarrow t_0 \]

- cost(\alpha): a number
  - Optional, default is 1

\[ \text{move}(r,l,m) \]
  pre: \( \text{loc}(r)=l, \text{adjacent}(l,m) \)
  eff: \( \text{loc}(r) \leftarrow m \)

\[ \text{take}(r,l,c) \]
  pre: \( \text{cargo}(r)=\text{nil}, \text{loc}(r)=l, \text{loc}(c)=l \)
  eff: \( \text{cargo}(r) \leftarrow c, \text{loc}(c) \leftarrow r \)

\[ \text{put}(r,l,c) \]
  pre: \( \text{loc}(r)=l, \text{loc}(c)=r \)
  eff: \( \text{cargo}(r) \leftarrow \text{nil}, \text{loc}(c) \leftarrow l \)

Range(\( r \)) = \textit{Robots} = \{r1\}
Range(\( l \)) = Range(\( m \)) = \textit{Locs} = \{d1,d2,d3\}
Range(\( c \)) = \textit{Containers} = \{c1,c2\}
**Actions**

- $\mathcal{A}$ = set of action templates
  
  move($r,l,m$)
  
  \[\text{pre: } \text{loc}(r) = l, \text{adjacent}(l, m)\]
  
  \[\text{eff: } \text{loc}(r) \leftarrow m\]

  take($r,l,c$)
  
  \[\text{pre: } \text{cargo}(r) = \text{nil}, \text{loc}(r) = l, \text{loc}(c) = l\]
  
  \[\text{eff: } \text{cargo}(r) \leftarrow c, \text{loc}(c) \leftarrow r\]

  put($r,l,c$)
  
  \[\text{pre: } \text{loc}(r) = l, \text{loc}(c) = r\]
  
  \[\text{eff: } \text{cargo}(r) \leftarrow \text{nil}, \text{loc}(c) \leftarrow l\]

  Range($r$) = Robots = \{r1\}
  
  Range($l$) = Range($m$) = Locs = \{d1,d2,d3\}
  
  Range($c$) = Containers = \{c1,c2\}

- Action: *ground instance* of an $\alpha \in \mathcal{A}$
  
  - replace each parameter with something in its range

  - $\mathcal{A}$ = \{all actions we can get from $\mathcal{A}$\}
    
    = \{all ground instances of members of $\mathcal{A}$\}

  move($r1,d1,d2$)
  
  \[\text{pre: } \text{loc}(r1) = d1, \text{adjacent}(d1,d2)\]
  
  \[\text{eff: } \text{loc}(r1) \leftarrow d2\]
### Actions

- \( A = \text{set of action templates} \)

\[
\text{move}(r,l,m) \\
\begin{align*}
p\text{re}: & \quad \text{loc}(r)=l, \text{adjacent}(l,m) \\
\text{ef}: & \quad \text{loc}(r) \leftarrow m
\end{align*}
\]

\[
\text{take}(r,l,c) \\
\begin{align*}
p\text{re}: & \quad \text{cargo}(r)=\text{nil}, \text{loc}(r)=l, \text{loc}(c)=l \\
\text{ef}: & \quad \text{cargo}(r) \leftarrow c, \text{loc}(c) \leftarrow r
\end{align*}
\]

\[
\text{put}(r,l,c) \\
\begin{align*}
p\text{re}: & \quad \text{loc}(r)=l, \text{loc}(c)=r \\
\text{ef}: & \quad \text{cargo}(r) \leftarrow \text{nil}, \text{loc}(c) \leftarrow l
\end{align*}
\]

- \( \text{Range}(r) = \text{Robots} = \{r1\} \)
- \( \text{Range}(l) = \text{Range}(m) = \text{Locs} = \{d1,d2,d3\} \)
- \( \text{Range}(c) = \text{Containers} = \{c1,c2\} \)

- \( \text{Action: ground instance of an } a \in A \)
  - replace each parameter with something in its range

- \( A = \{\text{all actions we can get from } A\} \)
  = \{\text{all ground instances of members of } A\}

**Poll:** Let \( A = \{\text{the action templates on this page}\}. \) How many move actions in \( A? \)

1: 1
2: 2
3: 3
4: 4
5: 6
6: 9
7: something else
Applicability

- $a$ is applicable in $s$ if
  - for every positive literal $l \in \text{pre}(a)$, $l \in s$ or $l$ is in one of the rigid relations
  - for every negative literal $\neg l \in \text{pre}(a)$, $l \notin s$ and $l$ isn’t in any of the rigid relations

- Rigid relation
  \[ \text{adjacent} = \{(d1,d2), (d2,d1), (d1,d3), (d3,d1)\} \]

- State
  \[ s_1 = \{\text{loc}(r1)=d1, \text{cargo}(r1)=\text{nil, loc}(c1)=d1\} \]

- Action template
  \[ \text{move}(r,l,m) \]
  \[ \text{pre: loc}(r) = l, \text{adjacent}(l, m) \]
  \[ \text{eff: } \text{loc}(r) \leftarrow m \]
  \[ \text{Range}(r) = \text{Robots} \]
  \[ \text{Range}(l) = \text{Range}(m) = \text{Locs} \]

- Applicable:
  \[ \text{move}(r1,d1,d2) \]
  \[ \text{pre: loc}(r1)=d1, \text{adjacent}(d1,d2) \]
  \[ \text{eff: loc}(r1) \leftarrow d2 \]

- Not applicable:
  \[ \text{move}(r1,d2,d1) \]
  \[ \text{pre: loc}(r1)=d2, \text{adjacent}(d2,d1) \]
  \[ \text{eff: loc}(r1) \leftarrow d1 \]

Poll: In $s_1$, how many applicable move actions?

1. 1 5. 5
2. 2 6. 6
3. 3 7. 7
4. 4 8. other
Computing $\gamma$

• If $a$ is applicable in $s$:
  
  \[
  \gamma(s,a) = \{(x,w) \mid \text{"x ← w" is in eff(a)}\}
  \cup \{(x,w) \in s \mid x \text{ isn’t the target of anything in eff(a)}\}
  \]

• $s_2 = \{\text{loc(r1)=d2, cargo(r1)=nil, loc(c1)=d1, loc(c2)=d2}\}$

• $\text{take(r1,d2,c2)}$
  
  \[
  \text{pre: cargo(r1)=nil, loc(r1)=d2, loc(c2)=d2}
  \]
  
  \[
  \text{eff: cargo(r1) ← c2, loc(c2) ← r1}
  \]

• $\gamma(s_2, \text{take(r1,d2,c2)}) = \{\text{loc(r1)=d2, cargo(r1)=c2, loc(c1)=d1, loc(c2)=r1}\}$
State-Variable Planning Domain

- Let
  
  \( B = \) finite set of objects
  
  \( R = \) finite set of rigid relations over \( B \)
  
  \( X = \) finite set of state variables
    
    - for every state variable \( x \), \( \text{Range}(x) \subseteq B \)
  
  \( S = \) state space over \( X \)
    
    - \( S = \) \{all variable-assignment functions that have sensible interpretations\}
  
  \( A = \) finite set of action templates
    
    - for every parameter \( y \), \( \text{Range}(y) \subseteq B \)
    
    \( A = \) \{all ground instances of action templates in \( A \)\}
  
  \( \gamma(s,a) = \{(x,w) \mid \text{eff}(a) \text{ contains the effect } x \leftarrow w\} \)
    
    \( \cup \{(x,w) \in S \mid x \text{ isn’t the target of any effect in } \text{eff}(a)\} \)
      
- Then \( \Sigma = (S,A,\gamma) \) is a state-variable planning domain
Interpretations

- Let $s$ be a variable-assignment function
  - $s$ is a state only if the values make sense in the environment we’re trying to represent
    - relation to model theory
- Can $\text{loc}(c1) = r1$ if $\text{cargo}(r1) = \text{nil}$?
  - Not in our intended interpretation
    - Mapping of symbols to what they represent
- Can both $\text{loc}(c1) = r1$ and $\text{loc}(c2) = r1$?
  - In our intended interpretation, can a robot carry more than one object at a time?

- How to enforce the intended interpretation?
  - Explicitly
    - Mathematical axioms
    - Integrity constraints
  - Implicitly
    - Write an initial state $s_0$ that satisfies the interpretation
    - Write the actions in such a way that whenever $s$ satisfies the interpretation, $\gamma(s,a)$ will too

\[ s_0 = \{ \text{loc}(r1) = d2, \text{cargo}(r1) = \text{nil}, \text{loc}(c1) = d1, \text{loc}(c2) = d2 \} \]
Plans

- **Plan**: sequence of actions $\pi = \langle a_1, a_2, \ldots, a_n \rangle$
  - $\text{cost}(\pi) = \sum_i \text{cost}(a_i)$
- $\pi$ is *applicable* in $s_0$ if the actions can be applied in the order given, i.e., there are states $s_1, s_2, \ldots, s_n$ such that $\gamma(s_0, a_1) = s_1$, $\gamma(s_1, a_2) = s_2$, $\ldots$, $\gamma(s_{n-1}, a_n) = s_n$
  - If so, then $\gamma(s_0, \pi) = s_n$

- $\pi = \langle \text{move}(r_1, d_3, d_1), \text{take}(r_1, d_1, c_1), \text{move}(r_1, d_1, d_3) \rangle$
- $\text{cost}(\pi) = 3$

\[ s_3 = \{ \text{loc}(r_1) = d_3, \text{cargo}(r_1) = \text{nil}, \text{loc}(c_1) = d_1, \text{loc}(c_2) = d_2 \} \]

\[ \gamma(s_3, \pi) = \{ \text{loc}(r_1) = d_3, \text{cargo}(r_1) = c_1, \text{loc}(c_1) = r_1, \text{loc}(c_2) = d_2 \} \]
State Space

- Directed graph
  - Nodes = states of the world
  - Directed edges: $\gamma$

- If $\pi = \langle a_1, a_2, \ldots, a_n \rangle$ is applicable in $s_0$, it produces a path $\langle s_0, s_1, s_2, \ldots, s_n \rangle$

\[
\begin{align*}
\gamma(s_0, a_1) &= s_1, \\
\gamma(s_1, a_2) &= s_2, \\
\vdots \\
\gamma(s_{n-1}, a_n) &= s_n
\end{align*}
\]
Planning Problems

- **State-variable planning problem** $P = (\Sigma, s_0, g)$
  - state-variable representation of a classical planning problem
    - $\Sigma = (S, A, \gamma)$ is a state-variable planning domain
    - $s_0 \in S$ is the initial state
    - $g$ is a set of ground literals called the goal

- $S_g = \{\text{all states in } S \text{ that satisfy } g\}$
  - $= \{s \in S \mid s \cup R \text{ contains every positive literal in } g, \text{ and none of the negative literals in } g\}$

- If $\gamma(s_0, \pi) \in S_g$ then $\pi$ is a solution for $P$

$$
adjacent = \{(d1,d2), (d2,d1), (d1,d3), (d3,d1)\}
$$

$$
s_0 = \{\text{loc}(r1)=d2, \text{cargo}(r1)=\text{nil}, \text{loc}(c1)=d1\}
$$

$$
g = \{\text{cargo}(r1)=c1\}
$$

$$
\pi = \langle \text{move}(r1,d2,d1), \text{take}(r1,d1,c1) \rangle
$$

**Poll:** How many solutions of length 3?
1. 1   5. 5
2. 2   6. 6
3. 3   7. 7
4. 4   8. other
Classical Representation

- **Motivation**
  - The field of AI planning started out as automated theorem proving
  - It still uses a lot of that notation
- **Classical representation is equivalent to state-variable representation**
  - Represents both rigid and varying properties using logical predicates
    - `adjacent(l,m)` - location `l` is adjacent to `m`
    - `loc(r) = l → loc(r,l)` - robot `r` is at location `l`
    - `loc(c) = r → loc(c,r)` - container `c` is on robot `r`
    - `cargo(r) = c → loaded(r)` - there’s a container on `r`

- **State `s` = a set of ground atoms**
  - Atom `a` is true in `s` iff `a ∈ s`

---

Poll: Should `s_0` also contain `¬loaded(r1)`?
1: yes 2: no
Classical planning operators

- Action templates

  move($r,l,m$)
  - pre: $\text{loc}(r)=l$, $\text{adjacent}(l, m)$
  - eff: $\text{loc}(r) \leftarrow m$

  take($r,l,c$)
  - pre: $\text{cargo}(r)=\text{nil}$, $\text{loc}(r)=l$, $\text{loc}(c)=l$
  - eff: $\text{cargo}(r) \leftarrow c$, $\text{loc}(c) \leftarrow r$

  put($r,l,c$)
  - pre: $\text{loc}(r)=l$, $\text{loc}(c)=r$
  - eff: $\text{cargo}(r) \leftarrow \text{nil}$, $\text{loc}(c) \leftarrow l$

Range($r$) = Robots = \{r1\}
Range($l$) = Range($m$) = Locs = \{d1,d2,d3\}
Range($c$) = Containers = \{c1,c2\}

- Classical planning operators

  move($r,l,m$)
  - pre: $\text{loc}(r,l)$, $\text{adjacent}(l, m)$
  - eff: $\neg \text{loc}(r,l)$, $\text{loc}(r,m)$

  take($r,l,c$)
  - pre: $\neg \text{loaded}(r)$, $\text{loc}(r,l)$, $\text{loc}(c,l)$
  - eff: $\text{loaded}(r)$, $\neg \text{loc}(c,l)$, $\text{loc}(c,r)$

  put($r,l,c$)
  - pre: $\text{loc}(r,l)$, $\text{loc}(c,r)$
  - eff: $\neg \text{loaded}(r)$, $\text{loc}(c,l)$
Actions

- Planning operator:
  
  \( o: \ move(r,l,m) \)
  
  pre: \( \text{loc}(r,l), \text{adjacent}(l,m) \)
  
  eff: \( \neg\text{loc}(r,l), \text{loc}(r,m) \)

- Action:
  
  \( a_1: \ move(r_1,d_2,d_1) \)
  
  pre: \( \text{loc}(r_1,d_2), \text{adjacent}(d_2,d_1) \)
  
  eff: \( \neg\text{loc}(r_1,d_2), \text{loc}(r_1,d_1) \)

- Let
  
  \( \gamma(s,a) = (s \setminus \text{eff}^{-}(a)) \cup \text{eff}^{+}(a) \)

\( s_0 = \{\text{adjacent}(d_1,d_2), \text{adjacent}(d_2,d_1), \text{adjacent}(d_1,d_3), \text{adjacent}(d_3,d_1), \text{loc}(c_1,d_1), \text{loc}(r_1,d_2)\} \)

\( \gamma(s_0, a_1) = \{\text{adjacent}(d_1,d_2), \text{adjacent}(d_2,d_1), \text{adjacent}(d_1,d_3), \text{adjacent}(d_3,d_1), \text{loc}(c_1,d_1), \text{loc}(r_1,d_1)\} \)

\( s \cap \text{pre}^{-}(a) = \emptyset \) and \( \text{pre}^{+}(a) \subseteq s \)

\( a \) is applicable in state \( s \) iff

- meaning?
Discussion

- Equivalent expressive power
  - Each can be converted to the other in linear time and space

- Classical representation
  - More natural for logicians
  - Don’t require single-valued functions

- State variables
  - More natural for engineers and computer programmers
  - When changing a value, don’t have to explicitly delete the old one

- Historically, classical representation has been more widely used
  - That’s starting to change

\[ x(b_1, \ldots, b_{n-1}) = b_n \] becomes \[ P_x(b_1, \ldots, b_{n-1}, b_n) \]

\[ P(b_1, \ldots, b_k) \] becomes \[ x_P(b_1, \ldots, b_k) = 1 \]

Poll: Could we instead use \[ x_P(b_1, \ldots, b_{k-1}) = b_k \]?

1: yes  2: no
PDDL

- Language for defining planning domains and problems
- Original version ≈ 1996
  - Just classical planning
- Multiple revisions and extensions
  - Different subsets accommodate different kinds of planning

- We’ll discuss the classical-planning subset
  - Chapter 2 of the PDDL book
Example domain

Classical actions:

move\( (r,l,m) \)
Precond: \( \text{loc}(r,l), \text{adjacent}(l,m) \)
Effects: \( \neg \text{loc}(r,l), \text{loc}(r,m) \)

take\( (r,l,c) \)
Precond: \( \text{loc}(r,l), \text{loc}(c,l), \neg \text{loaded}(r) \)
Effects: \( \text{loc}(c,r), \neg \text{loc}(c,l), \text{loaded}(r) \)

put\( (r,l,c) \)
Precond: \( \text{loc}(r,l), \text{loc}(c,r) \)
Effects: \( \text{loc}(c,l), \neg \text{loc}(c,r), \neg \text{loaded}(r) \)

(define (domain example-domain-1)
  (requirements :negative-preconditions)
  (:action move
    :parameters (?r ?l ?m)
    :precondition (and (loc ?r ?l)
                       (adjacent ?l ?m))
    :effect (and (not (loc ?r ?l))
                (loc ?r ?m)))
  (:action take
    :parameters (?r ?l ?c)
    :precondition (and (loc ?r ?l)
                       (loc ?c ?l)
                       (not (loaded ?r)))
    :effect (and (not (loc ?r ?l))
                (loc ?r ?m)))
  (:action put
    :parameters (?r ?l ?c)
    :precondition (and (loc ?c ?l)
                       (loc ?c ?r))
    :effect (and (loc ?c ?l)
                 (not (loc ?c ?r))
                 (not (loaded ?r))))
)
Example problem

\[ s_0 = \{ \text{adjacent}(d1,d2), \text{adjacent}(d2,d1), \text{adjacent}(d1,d3), \text{adjacent}(d3,d1), \text{loc}(c1,d1), \text{loc}(r1,d2) \} \]

\[ g = \{ \text{loc}(c1,r1) \} \]

\[
\text{(define (problem example-problem-1)}
\hspace{1cm} (:domain example-domain-1))
\]

\[
(:\text{init}
\hspace{1cm} (\text{adjacent} \ d1 \ d2)
\hspace{1cm} (\text{adjacent} \ d2 \ d1)
\hspace{1cm} (\text{adjacent} \ d1 \ d3)
\hspace{1cm} (\text{adjacent} \ d3 \ d1)
\hspace{1cm} (\text{loc} \ c1 \ d1)
\hspace{1cm} (\text{loc} \ r1 \ d2)
\]

\[
(:\text{goal} \ (\text{loc} \ c1 \ r1))
\]
Example typed domain

(define (domain example-domain-2)
  (:requirements
   :negative-preconditions
   :typing)
  (:types
   location movable-obj object
   robot container movable-obj)
  (:predicates
   (loc ?r - movable-obj
        ?l - location)
   (load ?r - robot)
   (adjacent ?l ?m - location))

(:action move
 :parameters (?r - robot
              ?l ?m - location)
 :precondition (and (loc ?r ?l)
                   (adjacent ?l ?m))
 :effect (and (not (loc ?r ?l))
            (loc ?r ?m)))

(:action take
 :parameters (?r - robot
              ?l - location
              ?c - container)
 :precondition (and (loc ?r ?l)
                   (loc ?c ?l)
                   (not (loaded ?r)))
 :effect (and (not (loc ?r ?l))
            (loc ?r ?m)))

(:action put
 :parameters (?r - robot
              ?l - location
              ?c - container)
 :precondition (and (loc ?r ?l)
                   (loc ?c ?r))
 :effect (and (loc ?c ?l)
              (not (loc ?c ?r))
              (not (loaded ?r))))
Example typed problem

\[ s_0 = \{\text{adjacent}(d1,d2), \text{adjacent}(d2,d1), \text{adjacent}(d1,d3), \text{adjacent}(d3,d1), \text{loc}(c1,d1), \text{loc}(r1,d2)\} \]

\[ g = \{\text{loc}(c1,r1)\} \]

(define (problem example-problem-2)
  (:domain example-domain-2)
  (:objects
    r1 - robot
    c1 - container
    d1 d2 d3 - location)
  (:init
    (adjacent d1 d2)
    (adjacent d2 d1)
    (adjacent d1 d3)
    (adjacent d3 d1)
    (loc c1 d1)
    (loc r1 d2)
  )
  (:goal (loc c1 r1)))
Summary

2.1 State-Variable Representation
- State-transition systems, classical planning assumptions
- Classical planning problems, plans, solutions
- Objects, rigid properties
- Varying properties, state variables, states as functions
- Action templates, actions, applicability, $\gamma$
- State-variable planning domains, plans, problems, solutions
- Comparison with classical representation

Classical fragment of PDDL
- Planning domains, planning problems
- untyped, typed
Outline

2.1 State-variable representation

2.2 Forward state-space search
  ▶ Start at initial state, search toward goal

2.6 Incorporating planning into an actor
2.3 Heuristic functions
2.4 Backward search
2.5 Plan-space search
Planning as Search

- Nearly all planning procedures are search procedures
  - *Search tree*: the data structure the procedure uses to keep track of which paths it has explored

**Search-Tree Terminology**

- **Node**: a pair \( v = (\pi, s) \), where \( s = \gamma(s_0, \pi) \)
  - In practice, \( v \) may contain other things
    - pointer to parent, cost(\( \pi \)), …
    - \( \pi \) not always stored explicitly, can be computed from the parent pointers
- **children** of \( v = \{ (\pi.a, \gamma(s,a)) \mid a \text{ is applicable in } s \} \)
- **successors** or **descendants** of \( v \):
  - children, children of children, etc.
- **ancestors** of \( v \)
  - = \{nodes that have \( v \) as a successor\}
- **initial** or **starting** node: \( v_0 \)
  - = (\{\}, \( s_0 \)) root of the search tree
- **path** in the search space: sequence of nodes \( \langle v_0, v_1, \ldots, v_n \rangle \) such that each \( v_i \) is a child of \( v_{i-1} \)
- **height** of search space
  - = length of longest acyclic path from \( v_0 \)
- **depth** of \( v \)
  - = length(\( \pi \)) = length of path from \( v_0 \) to \( v \)
- **branching factor** of \( v \)
  - = number of children of \( v \)
- **branching factor** of search tree
  - = max branching factor of the nodes
- **expand** \( v \): generate all children
Forward Search

Forward-search \((\Sigma, s_0, g)\)

\[
\begin{align*}
s & \leftarrow s_0; \\
\pi & \leftarrow \langle \rangle \\
\text{loop} & \\
\text{if } s \text{ satisfies } g & \text{ then return } \pi \\
A' & \leftarrow \{ a \in A \mid a \text{ is applicable in } s \} \\
\text{if } A' = \emptyset & \text{ then return failure} \\
\text{nondeterministically choose } a & \in A' \\
s & \leftarrow \gamma(s,a); \\
\pi & \leftarrow \pi.a
\end{align*}
\]

- Nondeterministic algorithm
  - *Sound*: if an execution trace returns a plan \(\pi\), it’s a solution
  - *Complete*: if the planning problem is solvable, at least one of the possible execution traces will return a solution

- Represents a class of deterministic search algorithms
  - Depends on how you implement the nondeterministic choice
    - Which leaf node to expand next, which nodes to prune
  - Won’t necessarily be complete
Deterministic Version

Deterministic-Search($\Sigma$, $s_0$, $g$)

$\text{Frontier} \leftarrow \{(\langle \rangle, s_0)\}$

$\text{Expanded} \leftarrow \emptyset$

while $\text{Frontier} \neq \emptyset$ do

select a node $\nu = (\pi, s) \in \text{Frontier}$ \hspace{1em} (i)

remove $\nu$ from $\text{Frontier}$

add $\nu$ to $\text{Expanded}$

if $s$ satisfies $g$ then return $\pi$ \hspace{1em} (ii)

$\text{Children} \leftarrow \{(\pi.a, \gamma(s,a)) \mid s \text{ satisfies } \text{pre}(a)\}$

prune 0 or more nodes from $\text{Children, Frontier, Expanded}$ \hspace{1em} (iii)

$\text{Frontier} \leftarrow \text{Frontier} \cup \text{Children}$

return failure

- Special cases:
  - depth-first, breath-first, A*, many others

- Classify by
  - how they select nodes (i)
  - how they prune nodes (iii)

- Pruning often includes $\text{cycle-checking}$:
  - Remove from $\text{Children}$ every node $(\pi,s)$ that has an ancestor $(\pi',s')$ such that $s' = s$

- In classical planning problems, $S$ is finite
  - Cycle-checking will guarantee termination
Breadth-First Search (BFS)

\[
\text{Deterministic-Search}(\Sigma, s_0, g)
\]

\[
\text{Frontier} \leftarrow \{ (\langle \rangle, s_0) \}
\]

\[
\text{Expanded} \leftarrow \emptyset
\]

\[
\text{while } \text{Frontier} \neq \emptyset \text{ do}
\]

\[
\text{select a node } \nu = (\pi, s) \in \text{Frontier} \quad (i)
\]

\[
\text{remove } \nu \text{ from } \text{Frontier}
\]

\[
\text{add } \nu \text{ to } \text{Expanded}
\]

\[
\text{if } s \text{ satisfies } g \text{ then return } \pi \quad (ii)
\]

\[
\text{Children} \leftarrow \{(\pi.a, \gamma(s,a)) \mid s \text{ satisfies pre}(a)\}
\]

\[
\text{prune 0 or more nodes from}
\]

\[
\text{Children, Frontier, Expanded} \quad (iii)
\]

\[
\text{Frontier} \leftarrow \text{Frontier} \cup \text{Children}
\]

\[
\text{return failure}
\]

(i): select \((\pi, s) \in \text{Frontier}\)

- with smallest length(\(\pi\))
  - tie-breaking rule: select oldest

(iii): remove every \((\pi, s) \in \text{Children} \cup \text{Frontier}\)

- such that \(s\) is in \(\text{Expanded}\)
  - Thus expand states at most once

- Properties
  - Terminates
  - Returns solution if one exists
    - shortest, but not least-cost
  - Worst-case complexity:
    - memory \(O(|S|)\)
    - running time \(O(b|S|)\)
  - Where
    - \(b = \text{max branching factor}\)
    - \(|S| = \text{number of states in } S\)
Depth-First Search (DFS)

Deterministic-Search(\(\Sigma, s_0, g\))

\[
\text{Frontier} \leftarrow \{() \} \\
\text{Expanded} \leftarrow \emptyset \\
\text{while} \ \text{Frontier} \neq \emptyset \ \text{do} \\
\quad \text{select a node } v = (\pi, s) \in \text{Frontier} \quad (i) \\
\quad \text{remove } v \text{ from } \text{Frontier} \\
\quad \text{add } v \text{ to } \text{Expanded} \\
\quad \text{if } s \text{ satisfies } g \text{ then return } \pi \quad (ii) \\
\quad \text{Children} \leftarrow \\
\quad \quad \{ (\pi.a, \gamma(s,a)) \mid s \text{ satisfies } \text{pre}(a) \} \\
\quad \text{prune 0 or more nodes from} \\
\quad \text{Children, Frontier, Expanded} \quad (iii) \\
\quad \text{Frontier} \leftarrow \text{Frontier} \cup \text{Children} \\
\text{return failure}
\]

\( (i) \): Select \((\pi, s) \in \text{Children}\) that has largest length(\(\pi\))

- Possible tie-breaking rules:
  - left-to-right, smallest \(h(s)\)

\( (iii) \): do cycle-checking, then prune all nodes that recursive depth-first search would discard

- Repeatedly remove from \(\text{Expanded}\) any node that has no children in \(\text{Children} \cup \text{Frontier} \cup \text{Expanded}\)

- Properties
  - Terminates
  - Returns solution if there is one
    - No guarantees on quality
  - Worst-case running time \(O(b^l)\)
  - Worst-case memory \(O(bl)\)
    - \(b = \text{max branching factor}\)
    - \(l = \text{max depth of any node}\)
Uniform-Cost Search

Deterministic-Search($\Sigma, s_0, g$)

1. $\text{Frontier} \leftarrow \{(), s_0\}$
2. $\text{Expanded} \leftarrow \emptyset$
3. while $\text{Frontier} \neq \emptyset$ do
   1. select a node $\nu = (\pi, s) \in \text{Frontier}$ (i)
   2. remove $\nu$ from $\text{Frontier}$
   3. add $\nu$ to $\text{Expanded}$
   4. if $s$ satisfies $g$ then return $\pi$ (ii)
4. $\text{Children} \leftarrow \{(\pi.a, g(s,a)) \mid s \text{ satisfies } \text{pre}(a)\}$
5. prune 0 or more nodes from $\text{Children}$, $\text{Frontier}$, $\text{Expanded}$ (iii)
6. $\text{Frontier} \leftarrow \text{Frontier} \cup \text{Children}$
7. return failure

(i): Select $(\pi, s) \in \text{Children}$ that has smallest cost($\pi$)

(iii): Prune every $(\pi, s) \in \text{Children} \cup \text{Frontier}$ such that $\text{Expanded}$ already contains a node $(\pi', s)$

- Properties
  - Terminates
  - Finds optimal solution if one exists
  - Worst-case time $O(b|S|)$
  - Worst-case memory $O(|S|)$

Poll: If node $\nu$ is expanded before node $\nu'$, then how are cost($\nu$) and cost($\nu'$) related?
1. $\text{cost}(\nu) < \text{cost}(\nu')$
2. $\text{cost}(\nu) \leq \text{cost}(\nu')$
3. $\text{cost}(\nu) > \text{cost}(\nu')$
4. $\text{cost}(\nu) \geq \text{cost}(\nu')$
5. None of the above
Heuristic Functions

- Idea: estimate the cost of getting from a state \( s \) to a goal

- Let \( h^*(s) = \min \{ \text{cost}(\pi) \mid \gamma(s,\pi) \in S_g \} \)
  - Note that \( h^*(s) \geq 0 \) for all \( s \)

- heuristic function \( h(s) \):
  - Returns estimate of \( h^*(s) \)
  - Require \( h(s) \geq 0 \) for all \( s \)

- Example:
  - \( s \) = the city you’re in
  - Action: follow road from \( s \) to a neighboring city
  - \( h^*(s) = \) smallest distance by road from \( s \) to Bucharest
  - \( h(s) = \) straight-line distance from \( s \) to Bucharest

---

from Russell & Norvig, Artificial Intelligence: A Modern Approach
Greedy Best-First Search (GBFS)

Deterministic-Search($\Sigma, s_0, g$)

1. $Frontier \leftarrow \{(\emptyset, s_0)\}$
2. $Expanded \leftarrow \emptyset$

while $Frontier \neq \emptyset$ do

1. select a node $\nu = (\pi, s) \in Frontier$
2. remove $\nu$ from $Frontier$
3. add $\nu$ to $Expanded$
4. if $s$ satisfies $g$ then return $\pi$
5. $Children \leftarrow \{(\pi.a, \gamma(s,a)) \mid s \text{ satisfies } \text{pre}(a)\}$
6. prune 0 or more nodes from $Children, Frontier, Expanded$
7. $Frontier \leftarrow Frontier \cup Children$

return failure

- Idea: choose a node that’s likely to be close to a goal
- Node selection
  - Select a node $\nu = (\pi, s) \in Frontier$ for which $h(s)$ is smallest
- Pruning: for every node $\nu = (\pi, s)$ in $Children$:
  - If $Children \cup Frontier \cup Expanded$ contains another node with the same state $s$, then we’ve found multiple paths from $s_0$ to $s$
  - Keep only the one with the lowest cost
  - If more than one such node, keep the oldest
- Properties
  - Terminates; returns a solution if one exists
    - Often near-optimal
    - will usually find it quickly

Poll: Have you seen GBFS before?

1. yes
2. no
3. yes, but I don’t remember it very well
- generates 10 nodes
- solution cost 450
Deterministic-Search($\Sigma, s_0, g$)

$\text{Frontier} \leftarrow \{(\emptyset, s_0)\}$
$\text{Expanded} \leftarrow \emptyset$

while $\text{Frontier} \neq \emptyset$ do

select a node $\nu = (\pi, s) \in \text{Frontier}$ (i)
remove $\nu$ from $\text{Frontier}$
add $\nu$ to $\text{Expanded}$

if $s$ satisfies $g$ then return $\pi$ (ii)

$\text{Children} \leftarrow \{(\pi.a, \gamma(s,a)) \mid s \text{ satisfies } \text{pre}(a)\}$
prune 0 or more nodes from $\text{Children, Frontier, Expanded}$ (iii)

$\text{Frontier} \leftarrow \text{Frontier} \cup \text{Children}$
return failure

---

### A*

- **Idea:** try to choose a node on an optimal path from $s_0$ to goal
- **Node selection**
  - Select a node $v = (\pi, s)$ in $\text{Frontier}$ that has smallest value of $f(v) = \text{cost}(\pi) + h(s)$
  - Tie-breaking rule: choose oldest
- **Pruning:** same as in GBFS
  - for every node $v = (\pi,s)$ in $\text{Children}$:
    - If $\text{Children} \cup \text{Frontier} \cup \text{Expanded}$ contains another node with the same state $s$, then we’ve found multiple paths to $s$
    - Keep only the one with the lowest cost
    - If more than one such node, keep the oldest
- **Properties**
  - Terminates; returns a solution if one exists
  - Under certain conditions (I’ll discuss later), can guarantee optimality

---

**Poll:** Have you seen A* before?
1. yes
2. no
3. yes, but I don’t remember it very well
\[ f(s) = g(s) + h(s) \]

- generates 16 nodes
- solution cost 418
Admissibility

- Notation:
  - \( v = (\pi, s) \), where \( \pi \) is the plan for going from \( s_0 \) to \( s \)
  - \( h^*(s) = \min \{ \text{cost}(\pi') | \gamma(s, \pi') \text{ satisfies } g \} \)
  - \( f^*(v) = \text{cost}(\pi) + h^*(s) \)
  - \( f(v) = \text{cost}(\pi) + h(s) \)

- Definition: \( h \) is **admissible** if for every \( s \), \( h(s) \leq h^*(s) \)

**Poll:** If \( h(s) = \) straight-line distance from \( s \) to Bucharest, is \( h \) admissible?
1. Yes
2. No
3. I’m not sure
Admissibility

- **Notation:**
  - $v = (\pi, s)$, where $\pi$ is the plan for going from $s_0$ to $s$
  - $h^*(s) = \min \{ \text{cost}(\pi') \mid \gamma(s, \pi') \text{ satisfies } g \}$
  - $f^*(v) = \text{cost}(\pi) + h^*(s)$
  - $f(v) = \text{cost}(\pi) + h(s)$

- **Definition:** $h$ is admissible if for every $s$, $h(s) \leq h^*(s)$

**Poll:** If $h$ is admissible, does it follow that $f(v) \leq f^*(v)$ for every node $v$?
1. Yes
2. No
3. I’m not sure
**Dominance**

- Definition:
  - Let $h_1$, $h_2$ be heuristic functions
  - $h_2$ dominates $h_1$ if $\forall s$, $h_1(s) \leq h_2(s) \leq h^*(s)$

**Poll:** Let $h_1(s) = 0$ and $h_2(s)$ = straight-line distance from $s$ to Bucharest. Does $h_2$ dominate $h_1$?
1. Yes
2. No
3. Not sure
Properties of A*

- In classical planning problems,
  - *Termination:* A* will always terminate
  - *Completeness:* if the problem is solvable, A* will return a solution
  - *Optimality:* if $h$ is admissible then the solution will be optimal (least cost)

- If $h_2$ dominates $h_1$ then (assuming A* always resolves ties in favor of the same node)
  - A* with $h_2$ will never expand more nodes than A* with $h_1$
  - In most cases, A* with $h_2$ will expand fewer nodes than A* with $h_1$

- A* needs to store every node it visits
  - Running time and memory both $O(b|S|)$ in worst case
  - With good heuristic function, usually much smaller

- The book discusses additional properties
Comparison

- If $h$ is admissible, A* will return optimal solutions
  - But running time and memory requirement grow exponentially in $b$ and $d$

- GBFS returns the first solution it finds
  - There are cases where GBFS takes more time and memory than A*
    - But with a good heuristic function, such cases are rare
  - On classical planning problems with a good heuristic function
    - GBFS usually near-optimal solutions
    - GBFS does very little backtracking
    - Running time and memory requirement usually much less than A*
  - GBFS is used by most classical planners nowadays
### Depth-First Branch and Bound (DFBB)

**Deterministic-Search**($\Sigma, s_0, g$)

- $\textit{Frontier} \leftarrow \{(\langle \rangle, s_0)\}$
- $\textit{Expanded} \leftarrow \emptyset$
- $c^* \leftarrow \infty$; $\pi^* \leftarrow \text{failure}$

while $\textit{Frontier} \neq \emptyset$ do

- select a node $\nu = (\pi, s) \in \textit{Frontier}$ (i)
- remove $\nu$ from $\textit{Frontier}$ and add it to $\textit{Expanded}$
- if $s$ satisfies $g$ then return $\pi$ (ii)
- if $s$ satisfies $g$ and $\text{cost}(\pi) < c^*$ then
  - $c^* \leftarrow \text{cost}(\pi)$; $\pi^* \leftarrow \pi$
- else if $f(\nu) < c^*$ then
  - $\textit{Children} \leftarrow \{(\pi.a, \gamma(s,a)) \mid s \text{ satisfies } \text{pre}(a)\}$
  - prune 0 or more nodes from $\textit{Children}, \textit{Frontier}, \textit{Expanded}$ (iii)

Frontier $\leftarrow \textit{Frontier} \cup \textit{Children}$

return failure $\pi^*$

- **Node (step i) selection like DFS:**
  - Select $\nu = (\pi,s) \in \textit{Children}$ that has largest $\text{length}(\pi)$
  - Tie-breaking: smallest $h(s)$

- **Pruning (step iii)**
  - Like DFS, do cycle-checking and prune what recursive depth-first search would discard

- **Additional pruning during node expansion:**
  - If $f(\nu) \geq c^*$ then discard $\nu$

- **Properties**
  - Termination, completeness, optimality same as A*
  - Comparison to A*:
    - Usually less memory, more time
  - Worst-case is like DFS:
    $O(bl)$ memory, $O(b^l)$ time

---

**Poll:** Have you seen DFBB before?

1. yes
2. no
3. yes, but I don’t remember it very well

---

Basic ideas:

- depth-first search, guided by $h$
- $\pi^* =$ best solution so far
- $c^* =$ cost($\pi^*$)
- prune $\nu$ if $\text{cost}(\nu) \geq c^*$
- when frontier is empty, return $\pi^*$

- Node (step i) selection like DFS:
  - Select $\nu = (\pi,s) \in \textit{Children}$ that has largest $\text{length}(\pi)$
  - Tie-breaking: smallest $h(s)$

- Pruning (step iii)
  - Like DFS, do cycle-checking and prune what recursive depth-first search would discard

- Additional pruning during node expansion:
  - If $f(\nu) \geq c^*$ then discard $\nu$

- Properties
  - Termination, completeness, optimality same as A*
  - Comparison to A*:
    - Usually less memory, more time
  - Worst-case is like DFS:
    - $O(bl)$ memory, $O(b^l)$ time
Basic ideas:
- depth-first search, guided by $h$
- $\pi^* = \text{best solution so far}$
- $c^* = \text{cost}(\pi^*)$
- prune $v$ if $\text{cost}(v) \geq c^*$
- when frontier is empty, return $\pi^*$

straight-line dist. from $s$ to Bucharest

<table>
<thead>
<tr>
<th>Location</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arad</td>
<td>366</td>
</tr>
<tr>
<td>Bucharest</td>
<td>0</td>
</tr>
<tr>
<td>Craiova</td>
<td>160</td>
</tr>
<tr>
<td>Dobreta</td>
<td>242</td>
</tr>
<tr>
<td>Fagaras</td>
<td>176</td>
</tr>
<tr>
<td>Iasi</td>
<td>226</td>
</tr>
<tr>
<td>Lugoj</td>
<td>244</td>
</tr>
<tr>
<td>Mehadia</td>
<td>241</td>
</tr>
<tr>
<td>Neamt</td>
<td>234</td>
</tr>
<tr>
<td>Oradea</td>
<td>380</td>
</tr>
<tr>
<td>Pitesti</td>
<td>100</td>
</tr>
<tr>
<td>Rimnicu Vilcea</td>
<td>193</td>
</tr>
<tr>
<td>Sibiu</td>
<td>253</td>
</tr>
<tr>
<td>Timisoara</td>
<td>329</td>
</tr>
<tr>
<td>Urziceni</td>
<td>80</td>
</tr>
<tr>
<td>Vaslui</td>
<td>199</td>
</tr>
<tr>
<td>Zerind</td>
<td>374</td>
</tr>
</tbody>
</table>

$\pi^* = \text{failure}$
$c^* = \infty$
Basic ideas:
- depth-first search, guided by $h$
- $\pi^*$ = best solution so far
- $c^*$ = cost($\pi^*$)
- prune $v$ if $\text{cost}(v) \geq c^*$
- when frontier is empty, return $\pi^*$

$\pi^*$ = $< AS, SR, RP, PB >$
$c^*$ = 418

straight-line dist. from $s$ to Bucharest

<table>
<thead>
<tr>
<th>City</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arad</td>
<td>366</td>
</tr>
<tr>
<td>Bucharest</td>
<td>0</td>
</tr>
<tr>
<td>Craiova</td>
<td>160</td>
</tr>
<tr>
<td>Dobreta</td>
<td>242</td>
</tr>
<tr>
<td>Fagaras</td>
<td>176</td>
</tr>
<tr>
<td>Iasi</td>
<td>226</td>
</tr>
<tr>
<td>Lugoj</td>
<td>244</td>
</tr>
<tr>
<td>Mehadia</td>
<td>241</td>
</tr>
<tr>
<td>Neamț</td>
<td>234</td>
</tr>
<tr>
<td>Oradea</td>
<td>380</td>
</tr>
<tr>
<td>Pitesti</td>
<td>100</td>
</tr>
<tr>
<td>Rimnicu Vilcea</td>
<td>193</td>
</tr>
<tr>
<td>Sibiu</td>
<td>253</td>
</tr>
<tr>
<td>Timisoara</td>
<td>329</td>
</tr>
<tr>
<td>Urziceni</td>
<td>80</td>
</tr>
<tr>
<td>Vaslui</td>
<td>199</td>
</tr>
<tr>
<td>Zerind</td>
<td>374</td>
</tr>
</tbody>
</table>
Basic ideas:

- depth-first search, guided by $h$
- $\pi^* =$ best solution so far
- $c^* =$ cost($\pi^*$)
- prune $\nu$ if cost($\nu$) $\geq c^*$
- when frontier is empty, return $\pi^*$

$\pi^* = \langle \text{AS, SR, RP, PB} \rangle$

$c^* = 418$

- generates 16 nodes
- solution cost 418
Comparisons

- If $h$ is admissible, both A* and DFBB will return optimal solutions
  - Usually DFBB generates more nodes, but A* takes more memory
  - DFBB does badly in highly connected graphs (many paths to each state)
    - Can have exponentially worse running time than A* (generates nodes exponentially many times)
  - DFBB best in problems where $S$ is a tree of uniform height, all solutions at the bottom (e.g., constraint satisfaction)
    - DFBB and A* have similar running time
    - A* can take exponentially more memory than DFBB

- DFS returns the first solution it finds
  - can take much less time than DFBB
  - but solution can be very far from optimal
Iterative Deepening (IDS)

\[ \text{IDS}(\Sigma, s_0, g) \]

for \( k = 1 \) to \( \infty \) do

- do a depth-first search, backtracking at every node of depth \( k \)
- if the search found a solution then return it
- if the search generated no nodes of depth \( k \) then return failure

- Nodes generated:
  - \( a \)
  - \( a, b, c \)
  - \( a, b, c, d, e, f, g \)
  - \( a, b, c, d, e, f, g, h, i, j, k, l, m, n, o \)

- Solution path \( \langle a, c, g, o \rangle \)

- Total number of nodes generated:
  \[ 1 + 3 + 7 + 15 = 26 \]

- If goal is at depth \( d \) and branching factor is 2:
  \[ \sum_{i=1}^{d} (2^i - 1) = \sum_{i=1}^{d} 2^i - \sum_{i=1}^{d} 1 = (2^{d+1} - 2) - d = O(2^d) \]

**Poll:** Have you seen Iterative Deepening before?

1. yes
2. no
3. yes, but I don’t remember it very well

**Poll:** How many nodes generated if branching factor is \( b \) instead of 2?

1. \( O(b2^d) \)
2. \( O(b^d) \)
3. \( O(b^{d+1}) \)
4. something else
Iterative Deepening (IDS)

IDS($\Sigma$, $s_0$, $g$)
for $k = 1$ to $\infty$
do a depth-first search, backtracking at every node of depth $k$
if the search found a solution then return it
if the search generated no nodes of depth $k$ then return failure

- Nodes generated:
  
  $a$
  $a,b,c$
  $a,b,c,d,e,f,g$
  $a,b,c,d,e,f,g,h,i,j,k,l,m,n,o$

- Solution path $\langle a,c,g,o \rangle$
- Total number of nodes generated:
  
  $1 + 3 + 7 + 15 = 26$

- If goal is at depth $d$ and branching factor is 2:
  
  $\sum_1^d (2^i-1) = \sum_1^d 2^i - \sum_1^d 1 = (2^{d+1} - 2) - d = O(2^d)$

Properties:

- Termination, completeness, optimality
  - same as BFS
- Memory (worst case): $O(bd)$
  - vs. $O(b^d)$ for BFS
- If the number of nodes grows exponentially with $d$:
  - worst-case running time $O(b^d)$, vs. $O(b^l)$ for DFS
  - $b = \text{max branching factor}$
  - $l = \text{max depth of any node}$
  - $d = \text{min solution depth if there is one, otherwise } l$
Summary

- 2.2 Forward State-Space Search
  - Forward-search, Deterministic-Search
  - cycle-checking
  - Breadth-first, depth-first, uniform-cost search
  - A*, GBFS, DFBB, IDS

Outline

2.1 State-variable representation
2.2 Forward state-space search

2.6 Incorporating planning into an actor
   Online lookahead, unexpected events

2.3 Heuristic functions
2.4 Backward search
2.5 Plan-space search
2.6 Incorporating Planning into an Actor

The best laid plans of mice and men oft go astray

–Robert Burns
(translated from Scots dialect)
Service Robot

\[ s_0 = \{ \text{loc}(r1) = \text{room3}, \text{loc}(o7) = \text{room1}, \text{cargo}(r1) = \text{nil} \} \]
\[ g = \{ \text{loc}(o7) = \text{room2} \} \]
\[ \pi = \langle a_1, a_2, a_3, a_4, a_5 \rangle \]

\[ a_1 = \text{go}(r1, \text{room3}, \text{hall}) \]
\[ a_2 = \text{navigate}(r1, \text{hall}, \text{room1}) \]
\[ a_3 = \text{take}(r1, \text{room1}, o7) \]
\[ a_4 = \text{navigate}(r1, \text{room1}, \text{room2}) \]
\[ a_5 = \text{put}(r1, \text{room2}, o7) \]
- **Execution failures**
  - locked door
  - robot battery goes dead

- **Unexpected events**
  - class ends, hallway gets crowded
  - someone puts an object onto r1

- **Incorrect info**
  - navigation error, go to wrong place

- **Missing information**
  - where is loc(o7)?

\[ s_0 = \{\text{loc}(r1)=\text{room3}, \text{loc}(o7)=\text{room1}, \text{cargo}(r1)=\text{nil}\} \]

\[ g = \{\text{loc}(o7)=\text{room2}\} \]

\[ \pi = \{a_1, a_2, a_3, a_4, a_5\} \]

\[ a_1 = \text{go}(r1,\text{room3},\text{hall}) \]

\[ a_2 = \text{navigate}(r1,\text{hall},\text{room1}) \]

\[ a_3 = \text{take}(r1,\text{room1},o7) \]

\[ a_4 = \text{navigate}(r1,\text{room1},\text{room2}) \]

\[ a_5 = \text{put}(r1,\text{room2},o7) \]
Using Planning in Acting

Run-Lookahead($\Sigma, g$)
while ($s \leftarrow$ abstraction of observed state $\xi) \not= g$ do
  $\pi \leftarrow$ Lookahead($\Sigma, s, g$)
  if $\pi =$ failure then return failure
  $a \leftarrow$ pop-first-action($\pi$); perform($a$)

- Lookahead is the planner
- Receding horizon:
  - Call Lookahead, obtain $\pi$, perform 1st action, call Lookahead again …
  - Like game-tree search (chess, checkers, etc.)
- Useful when unpredictable things are likely to happen
  - Replans immediately
- Potential problem:
  - May pause repeatedly while waiting for Lookahead to return
  - What if $\xi$ changes during the wait?
Using Planning in Acting

Run-Lazy-Lookahead($\Sigma, g$)
\[
s \leftarrow \text{abstraction of observed state } \xi \\
\text{while } s \not\equiv g \text{ do} \\
\quad \pi \leftarrow \text{Lookahead}(\Sigma, s, g) \\
\quad \text{if } \pi = \text{failure} \text{ then return failure} \\
\quad \text{while } \pi \not\equiv \langle \rangle \text{ and } s \not\equiv g \text{ and } \text{Simulate}(\Sigma, s, g, \pi) \neq \text{failure} \text{ do} \\
\quad \quad a \leftarrow \text{pop-first-action}(\pi); \ \text{perform}(a) \\
\quad s \leftarrow \text{abstraction of observed state } \xi
\]

- Call Lookahead, execute the plan as far as possible, don’t call Lookahead again unless necessary
- Simulate tests whether the plan will execute correctly
  - Could just compute $\gamma(s, \pi)$, or could do something more detailed
    - lower-level refinement, physics-based simulation
- Potential problems
  - may miss opportunities to replace $\pi$ with a better plan
  - without Simulate, may not detect problems until it’s too late
Using Planning in Acting

Run-Concurrent-Lookahead(\(\Sigma, g\))

\[
\pi \leftarrow \langle \rangle; \quad s \leftarrow \text{abstraction of observed state } \xi
\]
thread 1: // threads 1 and 2 run concurrently

loop

\[
\pi \leftarrow \text{Lookahead}(\Sigma, s, g)
\]

thread 2:

loop

if \(s \models g\) then return success
else if \(\pi = \text{failure}\) then return failure;
else if \(\pi \neq \langle \rangle\) and Simulate(\(\Sigma, s, g, \pi\)) \neq \text{failure}\) then

\[
a \leftarrow \text{pop-first-action}(\pi); \quad \text{perform}(a)
\]

\[
s \leftarrow \text{abstraction of observed state } \xi
\]

- May detect opportunities earlier than Run-Lazy-Lookahead
  - But may miss some that Run-Lookahead would find
- Without Simulate, may fail to detect problems until it’s too late
  - Not as bad at this as Run-Lazy-Lookahead
  - Possible work-around: restart Lookahead each time \(s\) changes
How to do Lookahead

- **Subgoaling**
  - Instead of planning for $g$, plan for a subgoal $g'$
  - Once $g'$ is achieved, plan for next subgoal

- **Receding horizon**
  - Return a plan that goes just part-way to $g'$
  - *E.g.*, cut off search at
    - every plan whose cost exceeds some value $c_{\text{max}}$
    - or whose length exceeds some value $l_{\text{max}}$
    - or when no time is left
Receding-Horizon Search

Deterministic-Search($\Sigma$, $s_0$, $g$)

$\begin{align*}
\text{Frontier} &\leftarrow \{(), s_0\} \\
\text{Expanded} &\leftarrow \emptyset \\
\text{while} \ \text{Frontier} \neq \emptyset \ \text{do} \\
\quad &\text{select a node } v = (\pi, s) \in \text{Frontier} \\
\quad &\text{remove } v \text{ from } \text{Frontier} \\
\quad &\text{add } v \text{ to } \text{Expanded} \\
\quad &\text{if } s \text{ satisfies } g \text{ then return } \pi \\
\quad &\text{Children} \leftarrow \\
\quad &\quad \{ (\pi.a, \gamma(s,a)) \mid s \text{ satisfies } \text{pre}(a) \} \\
\quad &\text{prune 0 or more nodes from } \\
\quad &\text{Children, Frontier, Expanded} \\
\quad &\text{Frontier} \leftarrow \text{Frontier} \cup \text{Children} \\
\quad &\text{return failure}
\end{align*}$

- Before line (i), put something like one of these:
  - cost-based cutoff:
    - if $\text{cost}(\pi) + h(s) > c_{\text{max}}$ then return $\pi$
  - length-based cutoff:
    - if $|\pi| > l_{\text{max}}$ then return $\pi$
  - time-based cutoff:
    - if $\text{time-left}() = 0$ then return $\pi$
Partial or Non-Optimal Plans

- Sampling
  - Planner is a modified version of greedy algorithm
    - Make randomized choice in line 4
    - Run several times, get several solutions
    - Return best one
  - Actor calls the planner repeatedly as it acts
    - An analogous technique is used in the game of go

**Greedy**($\Sigma, s, g, Visited$)
1. if $s$ satisfies $g$ then return $\pi$
2. $Act \leftarrow \{a \in A \mid s$ satisfies $pre(a)$ and $\gamma(s,a) \not\in Visited\}$
3. if $Act = \emptyset$ then return failure
4. $a \leftarrow \arg\min_{a \in Act} h(\gamma(s,a))$
5. $\pi \leftarrow \text{Greedy}(\Sigma, \gamma(s,a), g, Visited \cup \{s\})$
6. if $\pi \neq \text{failure}$ then return $a.\pi$
7. return failure
Example

- **Killzone 2**
  - “First-person shooter” game
  - ≈ 2009

- Special-purpose AI planner
  - Plans enemy actions at the squad level
    - Subproblems; solution plans are maybe 4–6 actions long
  - Different planning algorithm than what we’ve discussed so far
    - Hierarchical refinement as in Chapter 3
  - Quickly generates a plan for a subgoal
  - Replans several times per second as the world changes

- Why it worked:
  - Don’t want to get the best possible plan
  - Need actions that appear believable and consistent to human users
  - Need them very quickly
Summary

- 2.6 Incorporating Planning into an actor
  - Things that can go wrong while acting
  - Algorithms
    - Run-Lookahead,
    - Run-Lazy-Lookahead,
    - Run-Concurrent-Lookahead
  - Lookahead
    - subgoaling
    - receding-horizon search
    - sampling