True or False, and explain why: It is okay to use a non-admissible heuristic that over-estimates the distance to the goal by up to $C$ units if you would be happy with a solution path that is not more than $C$ units longer than the optimal path.

Solution: True.

Suppose that $h(n) \leq h^*(n) + c$, where $h^*$ is the minimal cost to a goal node (that is, it's the optimal heuristic function, which is obviously admissible). Let $C^*$ be the path cost of an optimal goal, that is, $C^* = g(n^*)$ for an optimal goal node $n^*$.

Let $G$ be a goal node that is suboptimal by more than $c$, that is, $g(G) > C^* + c$. Now consider any node $n$ on a path to an optimal goal. We have:

\[
\begin{align*}
    f(n) &= g(n) + h(n) \\
    \leq g(n) + h^*(n) + c & \quad \text{because } h \text{ does not overestimate by more than } c \\
    \leq C^* + c & \quad \text{because } n \text{ is on an optimal path to a goal} \\
    \leq g(G) & \quad \text{because } g(G) > C^* + c
\end{align*}
\]

Thus $G$ will never be expanded before $n$ is expanded. Since this holds for every $n$ on an optimal path to a goal, an entire optimal path to a goal is expanded before $G$ is expanded.
\( f(n) < g(G) \)
Planning
Coming Up

• Planning 2: Planning as Satisfiability
• Results of Phase I Othello Tournament (???)
• Homework 3 solutions given out in class
• Have a (warm?) March Break
• Tuesday March 18 - Exam 2: Logic
The goal of planning is to choose actions and ordering relations among these actions to achieve specified goals.

Search-based problem solving (e.g. 8-puzzle) was one example of planning, but our description of this problem used specific data structures and functions.

Here, we will develop a non-specific, logic-based language to represent knowledge about actions, states, and goals, and we will study how search algorithms can exploit this representation.
Knowledge Representation Tradeoff

- Expressiveness vs. computational efficiency
- STRIPS: a simple, still reasonably expressive planning language based on propositional logic
  1) Examples of planning problems in STRIPS
  2) Planning methods
  3) Extensions of STRIPS
- Like programming, knowledge representation is still an art
STRIPS Language through Examples
Vacuum-Robot Example

- Two rooms: $R_1$ and $R_2$
- A vacuum robot
- Dust
State Representation

Propositions that “hold” (i.e. are true) in the state

Logical “and” connective

\[ \text{In(Robot, } R_1) \land \text{Clean}(R_1) \]
State Representation

\[
\text{In}(\text{Robot, } R_1) \land \text{Clean}(R_1)
\]

- Conjunction of propositions
- No negated proposition, such as \( \neg \text{Clean}(R_2) \)
- Closed-world assumption: Every proposition that is not listed in a state is false in that state
- No “or” connective, such as \( \text{In}(\text{Robot}, R_1) \lor \text{In}(\text{Robot}, R_2) \)
- No quantified variables, e.g., \( \exists x \text{ Clean}(x) \)

No uncertainty
Goal Representation

Example: \( \text{Clean}(R_1) \land \text{Clean}(R_2) \)

- Conjunction of propositions
- No negated proposition
- No “or” connective
- No variable

A goal \( G \) is **achieved** in a state \( S \) if all the propositions in \( G \) (called **sub-goals**) are also in \( S \).
Action Representation

Right
- Precondition = In(Robot, R₁)
- Delete-list = In(Robot, R₁)
- Add-list = In(Robot, R₂)

\[
\text{In(Robot, } R_1) \land \text{Clean}(R_1)
\]

\[
\text{In(Robot, } R_2) \land \text{Clean}(R_1)
\]
Action Representation

Right

- Precondition = $\text{In(Robot, } R_1)$
- Delete-list = $\text{In(Robot, } R_1)$
- Add-list = $\text{In(Robot, } R_2)$

Sets of propositions

Same form as a goal: conjunction of propositions
Action Representation

Right

- **Precondition** = In(Robot, $R_1$)
- **Delete-list** = In(Robot, $R_1$)
- **Add-list** = In(Robot, $R_2$)

- An action $A$ is *applicable* to a state $S$ if the propositions in its precondition are all in $S$
- The *application* of $A$ to $S$ is a new state obtained by deleting the propositions in the delete list from $S$ and adding those in the add list
Other Actions

Left
- $P = \text{In}(\text{Robot, } R_2)$
- $D = \text{In}(\text{Robot, } R_2)$
- $A = \text{In}(\text{Robot, } R_1)$

Suck($R_1$)
- $P = \text{In}(\text{Robot, } R_1)$
- $D = \emptyset$ [empty set]
- $A = \text{Clean}(R_1)$

Suck($R_2$)
- $P = \text{In}(\text{Robot, } R_2)$
- $D = \emptyset$ [empty set]
- $A = \text{Clean}(R_2)$
Action Schema

It describes several actions, here: Suck(R₁) and Suck(R₂)

Parameter that will get “instantiated” by matching the precondition against a state

Suck(r)

- **P = In(Robot, r)**
- **D = Ø**
- **A = Clean(r)**
Action Schema

\[
P = \text{In}(\text{Robot, } R_2)
\]

\[
D = \emptyset
\]

\[
A = \text{Clean}(r)
\]

**Suck(r)**

- \( P = \text{In}(\text{Robot, } r) \)
- \( D = \emptyset \)
- \( A = \text{Clean}(r) \)

\[
\text{In}(\text{Robot, } R_2) \land \text{Clean}(R_1)
\]

\[
\text{In}(\text{Robot, } R_2) \land \text{Clean}(R_1) \land \text{Clean}(R_2)
\]
Action Schema

\[ P = \text{In}(\text{Robot}, R_2) \]
\[ D = \text{In}(\text{Robot}, R_2) \]
\[ A = \text{In}(\text{Robot}, R_1) \]

\[ Suck(r) \]

\[ r \leftarrow R_1 \]

\[ P = \text{In}(\text{Robot}, r) \]
\[ D = \emptyset \]
\[ A = \text{Clean}(r) \]

\[ \text{In}(\text{Robot}, R_1) \land \text{Clean}(R_1) \]

\[ \text{In}(\text{Robot}, R_1) \land \text{Clean}(R_1) \]
Blocks-World Example

- A robot hand can move blocks on a table
- The hand cannot hold more than one block at a time
- No two blocks can fit directly on the same block
- The table is arbitrarily large
Block(A) \land Block(B) \land Block(C) \land
On(A,TABLE) \land On(B,TABLE) \land On(C,A) \land
Clear(B) \land Clear(C) \land Handempty
Goal

\[ \text{On}(A, \text{TABLE}) \land \text{On}(B, A) \land \text{On}(C, B) \land \text{Clear}(C) \]
Goal

On(A, TABLE) \land On(B, A) \land On(C, B) \land Clear(C)
Goal

\[ \text{On}(A,\text{TABLE}) \land \text{On}(C,B) \]
Action

Unstack(x,y)

\[ P = Handempty \land Block(x) \land Block(y) \land Clear(x) \land On(x,y) \]

\[ D = Handempty, Clear(x), On(x,y) \]

\[ A = Holding(x), Clear(y) \]
Action

Unstack(x,y)

\[ P = \text{Handempty} \land \text{Block}(x) \land \text{Block}(y) \land \text{Clear}(x) \land \text{On}(x,y) \]

\[ D = \text{Handempty}, \text{Clear}(x), \text{On}(x,y) \]

\[ A = \text{Holding}(x), \text{Clear}(y) \]

Unstack(C,A)

\[ P = \text{Handempty} \land \text{Block}(C) \land \text{Block}(A) \land \text{Clear}(C) \land \text{On}(C,A) \]

\[ D = \text{Handempty}, \text{Clear}(C), \text{On}(C,A) \]

\[ A = \text{Holding}(C), \text{Clear}(A) \]
Action

Unstack(x,y)

P = Handempty \land Block(x) \land Block(y) \land Clear(x) \land On(x,y)
D = Handempty, Clear(x), On(x,y)
A = Holding(x), Clear(y)

Unstack(C,A)

P = Handempty \land Block(C) \land Block(A) \land Clear(C) \land On(C,A)
D = Handempty, Clear(C), On(C,A)
A = Holding(C), Clear(A)
All Actions

Unstack(x,y)
P = Handempty \land Block(x) \land Block(y) \land Clear(x) \land On(x,y)
D = Handempty, Clear(x), On(x,y)
A = Holding(x), Clear(y)

Stack(x,y)
P = Holding(x) \land Block(x) \land Block(y) \land Clear(y)
D = Clear(y), Holding(x)
A = On(x,y), Clear(x), Handempty

Pickup(x)
P = Handempty \land Block(x) \land Clear(x) \land On(x,Table)
D = Handempty, Clear(x), On(x,Table)
A = Holding(x)

Putdown(x)
P = Holding(x), \land Block(x)
D = Holding(x)
A = On(x,Table), Clear(x), Handempty
All Actions

Unstack(x,y)
P = Handempty \land Block(x) \land Block(y) \land Clear(x) \land On(x,y)
D = Handempty, Clear(x), On(x,y)
A = Holding(x), \text{Clear}(y)

Stack(x,y)
P = Holding(x) \land Block(x) \land Block(y) \land Clear(y)
D = \text{Clear}(y), Holding(x),
A = On(x,y), Clear(x), Handempty

Pickup(x)
P = Handempty \land Block(x) \land Clear(x) \land On(x,\text{Table})
D = Handempty, Clear(x), On(x,\text{TABLE})
A = Holding(x)

Putdown(x)
P = Holding(x), \land Block(x)
D = Holding(x)
A = On(x,\text{TABLE}), Clear(x), Handempty

A block can always fit on the table
Key-in-Box Example

- The robot must lock the door and put the key in the box
- The key is needed to lock and unlock the door
- Once the key is in the box, the robot can’t get it back
Initial State

\[ \text{In(Robot}_1, R_2) \land \text{In(Key}_2, R_2) \land \text{Unlocked(Door)} \]
Goal

Locked(Door) ∧ In(Key,Box)

[The robot’s location isn’t specified in the goal]
Actions

Grasp-Key-in-\(R_2\)
\[
\begin{align*}
P &= \text{In}(\text{Robot}, R_2) \land \text{In}(\text{Key}, R_2) \\
D &= \emptyset \\
A &= \text{Holding(\text{Key})}
\end{align*}
\]

Lock-Door
\[
\begin{align*}
P &= \text{Holding(\text{Key})} \\
D &= \emptyset \\
A &= \text{Locked(\text{Door})}
\end{align*}
\]

Move-Key-from-\(R_2\)-into-\(R_1\)
\[
\begin{align*}
P &= \text{In}(\text{Robot}, R_2) \land \text{Holding(\text{Key})} \land \text{Unlocked(\text{Door})} \\
D &= \text{In}(\text{Robot}, R_2), \text{In}(\text{Key}, R_2) \\
A &= \text{In}(\text{Robot}, R_1), \text{In}(\text{Key}, R_1)
\end{align*}
\]

Put-Key-Into-Box
\[
\begin{align*}
P &= \text{In}(\text{Robot}, R_1) \land \text{Holding(\text{Key})} \\
D &= \text{Holding(\text{Key})}, \text{In}(\text{Key}, R_1) \\
A &= \text{In}(\text{Key}, \text{Box})
\end{align*}
\]
Planning Methods
**Forward Planning**

Initial state

Goal: $\text{Clean} (R_1) \land \text{Clean} (R_2)$

Suck($R_1$)

Right

Left

Suck($R_2$)
Forward Planning

Goal: On(B, A) ∧ On(C, B)

Unstack(C, A))

Pickup(B)
Need for an Accurate Heuristic

- Forward planning simply searches the space of world states from the initial to the goal state.
- Imagine an agent with a large library of actions, whose goal is $G$, e.g., $G = \text{Have(Milk)}$.
- In general, many actions are applicable to any given state, so the branching factor is huge.
- In any given state, most applicable actions are irrelevant to reaching the goal $\text{Have(Milk)}$.
- Fortunately, an accurate consistent heuristic can be computed using planning graphs.
Planning Graph for a State of the Vacuum Robot

- \( S_0 \) contains the state’s propositions (here, the initial state)
- \( A_0 \) contains all actions whose preconditions appear in \( S_0 \)
- \( S_1 \) contains all propositions that were in \( S_0 \) or are contained in the add lists of the actions in \( A_0 \)
- So, \( S_1 \) contains all propositions that may be true in the state reached after the first action
- \( A_1 \) contains all actions not already in \( A_0 \) whose preconditions appear in \( S_1 \), hence that may be executable in the state reached after executing the first action.

Etc...
The smallest value of \( i \) such that \( S_i \) contains all the goal propositions is called the **level cost** of the goal (here \( i=2 \)).

By construction of the planning graph, it is a lower bound on the number of actions needed to reach the goal.

In this case, 2 is the actual length of the shortest path to the goal.
Planning Graph for Another State

- The level cost of the goal is 1, which again is the actual length of the shortest path to the goal
Whenever a new node is generated, compute the planning graph of its state [update the planning graph at the parent node]

Stop computing the planning graph when:
- Either the goal propositions are in a set $S_i$ [then $i$ is the level cost of the goal]
- Or when $S_{i+1} = S_i$ (the planning graph has leveled off) [then the generated node is not on a solution path]

Set the heuristic $h(N)$ of a node $N$ to the level cost of the goal for the state of $N$

$h$ is a consistent heuristic for unit-cost actions

Hence, $A^*$ using $h$ yields a solution with minimum number of actions
Size of Planning Graph

- An action appears at most once
- A proposition is added at most once and each $S_k$ ($k \neq i$) is a strict superset of $S_{k-1}$
- So, the number of levels is bounded by $\text{Min}\{\text{number of actions, number of propositions}\}$
- In contrast, the state space can be exponential in the number of propositions (why?)
- The computation of the planning graph may save a lot of unnecessary search work
Improvement of Planning Graph: Mutual Exclusions

- **Goal:** Refine the level cost of the goal to be a more accurate estimate of the number of actions needed to reach it
- **Method:** Detect obvious exclusions among propositions at the same level (see R&N)
- It usually leads to more accurate heuristics, but the planning graphs can be bigger and more expensive to compute
- Forward planning can still suffer from an excessive branching factor

- In general, there are much fewer actions that are relevant to achieving a goal than actions that are applicable to a state

- How to determine which actions are relevant? How to use them?

- \( \rightarrow \) Backward planning
Goal-Relevant Action

- An action is **relevant** to achieving a goal if a proposition in its add list matches a sub-goal proposition

- For example:

  Stack(B, A)

  \[ P = \text{Holding}(B) \land \text{Block}(B) \land \text{Block}(A) \land \text{Clear}(A) \]

  \[ D = \text{Clear}(A), \text{Holding}(B), \]

  \[ A = \text{On}(B,A), \text{Clear}(B), \text{Handempty} \]

  is relevant to achieving \[ \text{On}(B,A) \land \text{On}(C,B) \]
Regression of a Goal

The regression of a goal $G$ through an action $A$ is the least constraining precondition $R[G,A]$ such that:

If a state $S$ satisfies $R[G,A]$ then:
1. The precondition of $A$ is satisfied in $S$
2. Applying $A$ to $S$ yields a state that satisfies $G$
Example

- $G = \text{On}(B,A) \land \text{On}(C,B)$

- $\textbf{Stack}(C,B)$
  
  $P = \text{Holding}(C) \land \text{Block}(C) \land \text{Block}(B) \land \text{Clear}(B)$
  
  $D = \text{Clear}(B), \text{Holding}(C)$
  
  $A = \text{On}(C,B), \text{Clear}(C), \text{Handempty}$

- $R[G,\text{Stack}(C,B)] =$
  
  $\text{On}(B,A) \land$

  $\text{Holding}(C) \land \text{Block}(C) \land \text{Block}(B) \land \text{Clear}(B)$
Example

- $G = \text{On}(B,A) \land \text{On}(C,B)$

- **Stack($C,B$)**
  
  $P = \text{Holding}(C) \land \text{Block}(C) \land \text{Block}(B) \land \text{Clear}(B)$
  
  $D = \text{Clear}(B), \text{Holding}(C)$
  
  $A = \text{On}(C,B), \text{Clear}(C), \text{Handempty}$

- $R[G,\text{Stack}(C,B)] = \text{On}(B,A) \land \text{Holding}(C) \land \text{Block}(C) \land \text{Block}(B) \land \text{Clear}(B)$
Another Example

- $G = \text{In(key,Box)} \land \text{Holding(Key)}$

- **Put-Key-Into-Box**
  
  $P = \text{In(Robot,R}_1\text{)} \land \text{Holding(Key)}$
  
  $D = \text{Holding(Key), In(Key,R}_1\text{)}$
  
  $A = \text{In(Key,Box)}$

- $\mathcal{R}[G,\text{Put-Key-Into-Box}] = ??$
Another Example

- \( G = \text{In(key,Box)} \land \text{Holding(Key)} \)

- **Put-Key-Into-Box**
  
  \( P = \text{In(Robot,R}_1\text{)} \land \text{Holding(Key)} \)

  \( D = \text{Holding(Key)}, \text{In(Key,R}_1\text{)} \)

  \( A = \text{In(Key,Box)} \)

- \( R[G,\text{Put-Key-Into-Box}] = \text{False} \)

where False is the un-achievable goal

- This means that \( \text{In(key,Box)} \land \text{Holding(Key)} \) can’t be achieved by executing **Put-Key-Into-Box**
Computation of $R[G,A]$

1. If any sub-goal of $G$ is in $A$'s delete list then return False

2. Else
   a. $G' \leftarrow$ Precondition of $A$
   b. For every sub-goal $SG$ of $G$ do
      If $SG$ is not in $A$'s add list then add $SG$ to $G'$

3. Return $G'$
Backward Planning

On(B, A) \land On(C, B)
Backward Planning

\[ \text{On}(B,A) \land \text{On}(C,B) \]
\[ \text{Stack}(C,B) \]
\[ \text{On}(B,A) \land \text{Holding}(C) \land \text{Clear}(B) \]
\[ \text{Pickup}(C) \]
\[ \text{On}(B,A) \land \text{Clear}(B) \land \text{Handempty} \land \text{Clear}(C) \land \text{On}(C,\text{Table}) \]
\[ \text{Stack}(B,A) \]
\[ \text{Clear}(C) \land \text{On}(C,\text{TABLE}) \land \text{Holding}(B) \land \text{Clear}(A) \]
\[ \text{Pickup}(B) \]
\[ \text{Clear}(C) \land \text{On}(C,\text{Table}) \land \text{Clear}(A) \land \text{Handempty} \land \text{Clear}(B) \land \text{On}(B,\text{Table}) \]
\[ \text{Putdown}(C) \]
\[ \text{Clear}(A) \land \text{Clear}(B) \land \text{On}(B,\text{Table}) \land \text{Holding}(C) \]
\[ \text{Unstack}(C,A) \]
\[ \text{Clear}(B) \land \text{On}(B,\text{Table}) \land \text{Clear}(C) \land \text{Handempty} \land \text{On}(C,A) \]
Backward Planning

On(B,A) ∧ On(C,B)
  Stack(C,B)
On(B,A) ∧ Holding(C) ∧ Clear(B)
  Pickup(C)
On(B,A) ∧ Clear(B) ∧ Handempty ∧ Clear(C) ∧ On(C,Table)
  Stack(B,A)
Clear(C) ∧ On(C,Table) ∧ Holding(B) ∧ Clear(A)
  Pickup(B)
Clear(C) ∧ On(C,Table) ∧ Clear(A) ∧ Handempty ∧ Clear(B) ∧ On(B,Table)
  Putdown(C)
Clear(A) ∧ Clear(B) ∧ On(B,Table) ∧ Holding(C)
  Unstack(C,A)
Clear(B) ∧ On(B,Table) ∧ Clear(C) ∧ Handempty ∧ On(C,A)
Backward planning searches a space of goals from the original goal of the problem to a goal that is satisfied in the initial state.

There are often much fewer actions relevant to a goal than there are actions applicable to a state → smaller branching factor than in forward planning.

The lengths of the solution paths are the same.
Consistent Heuristic for Backward Planning

A consistent heuristic is obtained as follows:

1. Pre-compute the planning graph of the initial state until it levels off
2. For each node $N$ added to the search tree, set $h(N)$ to the level cost of the goal associated with $N$

If the goal associated with $N$ can’t be satisfied in any set $S_k$ of the planning graph, it can’t be achieved, so prune it!

A single planning graph is computed
How Does Backward Planning Detect Dead-Ends?

\[ \text{On}(B,A) \land \text{On}(C,B) \]

\[
\text{Stack}(C,B)
\]
How Does Backward Planning Detect Dead-Ends?

\[ On(B, A) \land On(C, B) \]

\[ \downarrow \]

\[ Stack(B, A) \]

\[ \quad \]

\[ Holding(B) \land Clear(A) \land On(C, B) \]

\[ \downarrow \]

\[ Stack(C, B) \]

\[ \quad \]

\[ Holding(B) \land Clear(A) \land Holding(C) \land Clear(B) \]

\[ \downarrow \]

\[ Pick(B) \quad [\text{delete list contains Clear}(B)] \]

\[ False \]
How Does Backward Planning Detect Dead-Ends?

\[
\text{On}(B, A) \land \text{On}(C, B) \\
\downarrow \quad \text{Stack}(B, A) \\
\text{Holding}(B) \land \text{Clear}(A) \land \text{On}(C, B)
\]

A state constraint such as
\[
\text{Holding}(x) \Rightarrow \neg (\exists y)\text{On}(y, x)
\]
would have made it possible to prune the path earlier.
Some Extensions of STRIPS Language
Extensions of STRIPS

1. Negated propositions in a state

\[ \text{Dump-Dirt}(r) \]

\[ P = \text{In}(\text{Robot}, r) \land \text{Clean}(r) \]
\[ E = \neg \text{Clean}(r) \]

\[ \text{Suck}(r) \]

\[ P = \text{In}(\text{Robot}, r) \land \neg \text{Clean}(r) \]
\[ E = \text{Clean}(r) \]

- \( Q \) in \( E \) means delete \( \neg Q \) and add \( Q \) to the state
- \( \neg Q \) in \( E \) means delete \( Q \) and add \( \neg Q \)

**Open world assumption:** A proposition in a state is true if it appears positively and false otherwise. A non-present proposition is unknown.

Planning methods can be extended rather easily to handle negated proposition (see R&N), but state descriptions are often much longer (e.g., imagine if there were 10 rooms in the above example).
Extensions of STRIPS

2. Equality/Inequality Predicates

Blocks world:

Move(x, y, z)
\[ P = \text{Block}(x) \land \text{Block}(y) \land \text{Block}(z) \land \text{On}(x, y) \land \text{Clear}(x) \land \text{Clear}(z) \land (x \neq z) \]
\[ D = \text{On}(x, y), \text{Clear}(z) \]
\[ A = \text{On}(x, z), \text{Clear}(y) \]

Move(x, Table, z)
\[ P = \text{Block}(x) \land \text{Block}(z) \land \text{On}(x, \text{Table}) \land \text{Clear}(x) \land \text{Clear}(z) \land (x \neq z) \]
\[ D = \text{On}(x, y), \text{Clear}(z) \]
\[ A = \text{On}(x, z) \]

Move(x, y, Table)
\[ P = \text{Block}(x) \land \text{Block}(y) \land \text{On}(x, y) \land \text{Clear}(x) \land \text{On}(x, \text{Table}) \]
\[ D = \text{On}(x, y) \]
\[ A = \text{On}(x, \text{Table}), \text{Clear}(y) \]
Extensions of STRIPS
2. Equality/Inequality Predicates

Blocks world:

\( \text{Move}(x, y, z) \)
\[ P = \text{Block}(x) \land \text{Block}(y) \land \text{Block}(z) \land \text{On}(x, y) \land \text{Clear}(x) \]
\[ \land \text{Clear}(z) \land (x \neq z) \]
\[ D = \text{On}(x, y), \text{Clear}(z) \]
\[ A = \text{On}(x, z), \text{Clear}(y) \]

\( \text{Move}(x, \text{Table}, z) \)
\[ P = \text{Block}(x) \land \text{Block}(z) \land \text{On}(x, \text{Table}) \]
\[ \land \text{Clear}(z) \land (x \neq z) \]
\[ D = \text{On}(x, y), \text{Clear}(z) \]
\[ A = \text{On}(x, z) \]

\( \text{Move}(x, y, \text{Table}) \)
\[ P = \text{Block}(x) \land \text{Block}(y) \land \text{On}(x, y) \land \text{Clear}(x) \]
\[ D = \text{On}(x, y) \]
\[ A = \text{On}(x, \text{Table}), \text{Clear}(y) \]

Planning methods simply evaluate \((x \neq z)\) when the two variables are instantiated. This is equivalent to considering that propositions \((A \neq B), (A \neq C), ...\) are implicitly true in every state.
Extensions of STRIPS

3. Algebraic expressions

Two flasks \( F_1 \) and \( F_2 \) have volume capacities of 30 and 50, respectively.
\( F_1 \) contains volume 20 of some liquid
\( F_2 \) contains volume 15 of this liquid

State:
\[
\text{Cap}(F_1, 30) \land \text{Cont}(F_1, 20) \land \text{Cap}(F_2, 50) \land \text{Cont}(F_2, 15)
\]

Action of pouring a flask into the other:

**Pour\( (f, f') \)**
\[
P = \text{Cont}(f, x) \land \text{Cap}(f', c') \land \text{Cont}(f', y) \land (f \neq f')
\]
\[
D = \text{Cont}(f, x), \text{Cont}(f', y),
\]
\[
A = \text{Cont}(f, \max\{x+y-c', 0\}), \text{Cont}(f', \min\{x+y, c'\})
\]
Extensions of STRIPS

3. Algebraic expressions

Two flasks $F_1$ and $F_2$ have volume capacities of 30 and 50, respectively.
$F_1$ contains volume 20 of some liquid
$F_2$ contains volume 15 of this liquid

State:
- $\text{Cap}(F_1, 30)$
- $\text{Cont}(F_1, 20)$
- $\text{Cap}(F_2, 50)$
- $\text{Cont}(F_2, 15)$

This extension requires planning methods equipped with algebraic manipulation capabilities

Action of pouring a flask into the other:

Pour($f, f'$)

$P = \text{Cont}(f, x) \land \text{Cap}(f', c') \land \text{Cont}(f', y) \land (f \neq f')$
$D = \text{Cont}(f, x), \text{Cont}(f', y)$,
$A = \text{Cont}(f, \max\{x+y-c', 0\}), \text{Cont}(f', \min\{x+y, c'\})$
Extensions of STRIPS

4. State Constraints

State:
\(\text{Adj}(1,2) \land \text{Adj}(2,1) \land \ldots \land \text{Adj}(8,9) \land \text{Adj}(9,8) \land \text{At}(h,1) \land \text{At}(b,2) \land \text{At}(c,4) \land \ldots \land \text{At}(f,9) \land \text{Empty}(3)\)

\(\text{Move}(x,y,z)\)
\[\begin{align*}
P &= \text{At}(x,y) \land \text{Empty}(z) \land \text{Adj}(y,z) \\
D &= \text{At}(x,y), \text{Empty}(z) \\
A &= \text{At}(x,z), \text{Empty}(y)
\end{align*}\]
Extensions of STRIPS

4. State Constraints

State:
\[ \text{Adj}(1,2) \land \text{Adj}(2,1) \land \ldots \land \text{Adj}(8,9) \land \text{Adj}(9,8) \land \text{At}(h,1) \land \text{At}(b,2) \land \text{At}(c,4) \land \ldots \land \text{At}(f,9) \land \text{Empty}(3) \]

State constraint:
\[ \text{Adj}(x,y) \Rightarrow \text{Adj}(y,x) \]

Move\((x,y,z)\)
\[ P = \text{At}(x,y) \land \text{Empty}(z) \land \text{Adj}(y,z) \]
\[ D = \text{At}(x,y), \text{Empty}(z) \]
\[ A = \text{At}(x,z), \text{Empty}(y) \]
More Complex State Constraints in 1\textsuperscript{st}-Order Predicate Logic

Blocks world:

\[(\forall x)[\text{Block}(x) \land \neg(\exists y)\text{On}(y,x) \land \neg\text{Holding}(x)] \rightarrow \text{Clear}(x)\]

\[(\forall x)[\text{Block}(x) \land \text{Clear}(x)] \rightarrow \neg(\exists y)\text{On}(y,x) \land \neg\text{Holding}(x)\]

Handempty \iff \neg(\exists x)\text{Holding}(x)

would simplify greatly the description of the actions

State constraints require planning methods with logical deduction capabilities, to determine whether goals are achieved or preconditions are satisfied
Some Applications of AI Planning

- Military operations
- Operations in container ports
- Construction tasks
- Machining and manufacturing
- Autonomous control of satellites and other spacecrafts
Deep Space One

Started: January 1996
Launch: October 15th, 1998

http://ic.arc.nasa.gov/projects/remote-agent/pstext.html