The goal of action planning is to choose actions and ordering relations among these actions to achieve specified goals.

Search-based problem solving applied to 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: R₁ and R₂
- A vacuum robot
- Dust

State Representation

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

Propositions that "hold" (i.e., are true) in the state

Logical "and" connective
State Representation

In(Robot, R₁) ∧ Clean(R₁)

- Conjunction of propositions
- No negated proposition, such as ¬Clean(R₂)
- Closed-world assumption: Every proposition that is not listed in a state is false in that state
- No "or" connective, such as In(Robot, R₁) ∨ In(Robot, R₂)
- No variable, e.g., ∃x Clean(x)

Goal Representation

Example: Clean(R₁) ∧ Clean(R₂)

- 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)

In(Robot, R₁) ∧ Clean(R₁) In(Robot, R₁) ∧ Clean(R₂)
Action Representation

Right

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

Sets of propositions

- Same form as a goal: conjunction of propositions

Other Actions

Left

- **P** = In(Robot, R₂)
- **D** = In(Robot, R₂)
- **A** = In(Robot, R₁)

Suck(R₁)  
- **P** = In(Robot, R₁)  
- **D** = ∅  
- **A** = Clean(R₁)

Suck(R₂)  
- **P** = In(Robot, R₂)  
- **D** = ∅  
- **A** = Clean(R₂)
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(\( r \))
• \( P = \text{In}(\text{Robot, } r) \)
• \( D = \emptyset \) (empty list)
• \( A = \text{Clean}(r) \)

Action Schema

It describes several actions, here: Suck(\( R_1 \)) and Suck(\( R_2 \))

Parameter that will get "instantiated" by matching the precondition against a state.

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

Action Schema

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

Suck(\( r \))
• \( P = \text{In}(\text{Robot, } r) \)
• \( D = \emptyset \)
• \( A = \text{Clean}(r) \)
**Action Schema**

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

**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.

**State**

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

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

Goal

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

Goal

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

Goal

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

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

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

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

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

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

**Goal**

\[
\text{Locked(Door)} \land \text{In(Key,Box)}
\]

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

**Grasp-Key-in-R2**
- **P**: $\text{In(Robot,R2)} \land \text{In(Key,R2)}$
- **D**: $\emptyset$
- **A**: Holding(Key)

**Lock-Door**
- **P**: Holding(Key)
- **D**: $\emptyset$
- **A**: Locked(Door)

**Move-Key-from-R2-into-R1**
- **P**: $\text{In(Robot,R2)} \land \text{Holding(Key)} \land \text{Unlocked(Door)}$
- **D**: $\text{In(Robot,R1)}, \text{In(Key,R1)}$
- **A**: $\text{In(Robot,R1)}, \text{In(Key,R1)}$

**Put-Key-Into-Box**
- **P**: $\text{In(Robot,R1)} \land \text{Holding(Key)}$
- **D**: Holding(Key), $\text{In(Key,R1)}$
- **A**: $\text{In(Key,Box)}$

### Planning Methods

- **Forward Planning**

- **Left**
  - Initial state: $\text{Suck(R1)}$
  - Goal: $\text{Clean(R1)} \land \text{Clean(R2)}$

- **Right**
  - Initial state: $\text{Suck(R2)}$
  - Goal: $\text{Clean(R1)} \land \text{Clean(R2)}$
Forward Planning

Goal: On(B,A) ∨ On(C,B)

- Unstack(C,A)
- Pickup(B)

Goal: On(B,A) ∧ On(C,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 = Have(Milk).
- In general, many actions are applicable to any given state, so the branching factor is huge.
- Fortunately, an accurate consistent heuristic can be computed using planning graphs.

Planning Graph for a State of the Vacuum Robot

- S0 contains the state's propositions (here, the initial state).
- A0 contains all actions whose preconditions appear in S0.
- S1 contains all propositions that were in S0 or are contained in the add lists of the actions in A0.
- So, S1 contains all propositions that may be true in the state reached after the first action.
- A1 contains all actions not already in A0, whose preconditions appear in S1, hence may be executable in the state reached after executing the first action. Etc.
Planning Graph for a State of the Vacuum Robot

- The 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.

Application of Planning Graphs to Forward Planning

- 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 \( S_{i+1} = S_i \) (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 \( \min(\text{number of actions, number of propositions}) \).
- In contrast, the state space can be exponential in the number of propositions.
- 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 still suffers 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

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

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

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

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

Another Example

- $G = \text{In}(\text{key},\text{Box}) \land \text{Holding}($Key$)$
- $\text{Put-Key-Into-Box}$
  - $P = \text{In}(\text{Robot \_R}), \text{Holding}($Key$)$
  - $D = \text{Holding}($Key$), \text{In}(\text{Key \_R})$
  - $A = \text{In}($Key\_Box$)$

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

Another Example

- $G = \text{In}(\text{key},\text{Box}) \land \text{Holding}($Key$)$
- $\text{Put-Key-Into-Box}$
  - $P = \text{In}(\text{Robot \_R}), \text{Holding}($Key$)$
  - $D = \text{Holding}($Key$), \text{In}(\text{Key \_R})$
  - $A = \text{In}($Key\_Box$)$

- $R[G, \text{Put-Key-Into-Box}] =$ False
  where False is the un-achievable goal

This means that $\text{In}(\text{key},\text{Box}) \land \text{Holding}($Key$)$ can't be achieved by executing $\text{Put-Key-Into-Box}$
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)

Initial state

Backward Planning

On(B,A) $\land$ On(C,B)$\land$...

Initial state
Backward Planning

- On(B, A) ∧ On(C, B)
- On(B, A) ∧ Holding(C) ∧ Clear(B)
- Pickup(C)
- On(B, A) ∧ Clear(B) ∧ Handempty ∧ Clear(C) ∧ On(C, Table)
- 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)
- Clear(B) ∧ Clear(C) ∧ On(B, Table)
- Clear(B) ∧ On(B, Table) ∧ Clear(C) ∧ Handempty ∧ On(C, A)

Search Tree

- 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, leading to a 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?

1. \( \text{On}(B,A) \wedge \text{On}(C,B) \)
2. \( \text{Stack}(C,B) \)
3. \( \text{Holding}(B) \wedge \text{Clear}(A) \wedge \text{On}(C,B) \)
4. \( \text{Stack}(B,A) \)
5. \( \text{Holding}(B) \wedge \text{Clear}(A) \wedge \text{Holding}(C) \wedge \text{Clear}(B) \)
6. \( \text{Pick}(B) \) [delete list contains Clear(B)]
7. False

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

1. Negated propositions in a state

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

<table>
<thead>
<tr>
<th>Dump-Dirt(r)</th>
<th>Suck(r)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ P = \text{In(Robot, } r) \land \text{Clean}(r) ]</td>
<td>[ P = \text{In(Robot, } r) \land \neg \text{Clean}(r) ]</td>
</tr>
</tbody>
</table>

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

\[ E = \neg \text{Clean}(r) \]

- \( Q \) 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 propositions (see R&N), but state descriptions are often much longer (e.g., imagine if there were 10 rooms in the above example).

2. Equality/Inequality Predicates

Blocks world:

\[ \text{Move}(x, y, z) \]

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

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

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

\[ \text{Move}(x, \text{Table}, z) \]

\[ P = \text{Block}(x) \land \text{Block}(z) \land \text{On}(x, \text{Table}) \land \text{Clear}(x) \land \neg \text{Clear}(z) \land (x \neq z) \]

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

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

\[ \text{Move}(x, y, \text{Table}) \]

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

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

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

...
### Extensions of STRIPS

#### 2. Equality/Inequality Predicates

**Blocks world**

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

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

---

#### 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:

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

This extension requires planning methods equipped with algebraic manipulation capabilities.
Extensions of STRIPS

4. State Constraints

State:

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

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 1st-Order Predicate Logic

Blocks world:

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

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

\[ \text{Handempty} \leftrightarrow \neg (\exists x \text{Holding}(x)) \]

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