Planning

Outside Materials (see Materials page)
What is Planning?

Given:
- A way to describe the world
- An initial state of the world
- A goal description
- A set of possible actions to change the world

Find:
- A prescription for actions to change the initial state into one that satisfies the goal
Planning Applications
Planning Applications
How do you represent change?

In your assignment, does anything change?
As actions change the world OR we consider possible actions, we want to:

- Know how an action will alter the world
- Keep track of the history of world states (have we been here before?)
- Answer questions about potential world states (what would happen if..?)
The situation calculus (McCarthy 63)

- Key idea: represent a snapshot of the world, called a ‘situation’ explicitly.

- ‘Fluents’ are statements that are true or false in any given situation, e.g. ‘I am at home’

- Actions map situations to situations.
S0

\text{holds(at(home), S0)}

\text{holds(color(door, red), S0)}

S1 = \text{result(go(store), S0)}

\text{~ holds(at(home), S1)}

\text{holds(at(store), S1)}

S2

mow_lawn()

\text{holds(at(home), S2)}
I go from home to the store, creating a new situation $S_1$. In $S_1$:

- My friend is still at home
- The store still sells chips
- My age is still the same
- Los Angeles is still the largest city in California...

*How can we efficiently represent everything that hasn’t changed?*
Normally, things stay true from one state to the next -- unless an action changes them:

\[ \text{holds}(\text{at}(X), \text{result}(A, S)) \text{ iff } A = \text{go}(X) \]

\[ \text{or } [\text{holds}(\text{at}(X), S) \text{ and } A \neq \text{go}(Y)] \]

We need one or more of these for every fluent.

Now we can use theorem proving to deduce a plan, right?

*No because of inefficiency and incapability of handling uncertainty, time, etc.*
Strips (Fikes and Nilsson 71)

- For efficiency, separates theorem-proving within a world state from searching the space of possible states

- Highly influential representation for actions:
  - Preconditions (list of propositions to be true)
  - Delete list (list of propositions that will become false)
  - Add list (list of propositions that will become true)
Example problem:

Initial state: at(home), \(\sim\)have(beer), \(\sim\)have(chips)

Goal: have(beer), have(chips), at(home)

Actions:

Buy (X):
- Pre: at(store)
- Add: have(X)

Go (X, Y):
- Pre: at(X)
- Del: at(X)
- Add: at(Y)
The Frame Problem

1. I go from home to the store – new situation $S_1$

2. In $S_1$
   - The store still sells chips
   - My age is still the same
   - Star Trek is still excellent
   - Santa Cruz is still a weird beach town
   - Boston is still cold

3. How can we efficiently represent everything that doesn’t change?
The Ramification Problem

I go from home to the store – new situation $S_1$

In $S_1$

- I am now in West Roxbury
- The store has one more person in it
- My head is now in the store
- My legs are now in the store
- The contents of my pockets are now in the store...

That’s a lot of effects for an action!
Ramification problem:
- Some facts must be inferred based on world state

Frame problem:
- Facts are assumed to persist between states unless changed

All comes down to knowledge engineering!
Example: blocks world

Initial:

State I: (on-table A) (on C A) (on-table B) (clear B) (clear C)

Goal: (on A B) (on B C)

How would you represent the actions?
Searching for a Plan
Starting from the initial state:

- Pick an operator with satisfied preconditions
- Apply it to update state
- Expand tree to include updated state descriptions
- Continue search with updated state as current
- Stop when you find the goal within a state description
- Fail when you’ve checked everything
**Forward State-Space Search Algorithm**

```markdown
**function** fwdSearch(O, s, g)

state $\leftarrow s$

plan $\leftarrow \langle \rangle$

**loop**

if state.satisfies(g) then return plan

applicables $\leftarrow \{\text{ground instances from } O \text{ applicable in } state\}$

if applicables.isEmpty() then return failure

action $\leftarrow$ applicables.chooseOne()

state $\leftarrow \gamma(state, action)$

plan $\leftarrow$ plan $\cdot$ $\langle$ action $\rangle$
```
Example: blocks world

Initial:

Goal:

State I: (on-table A) (on C A) (on-table B) (clear B) (clear C)

Goal: (on B A) (on-table C) (on-table A)

Class Assignment: Trace this using forward search
Finding a Plan: Searching Backwards

- Search **backwards** from goal to find a solution
  - Only looks at relevant actions!

Starting from the **goal**:
- If initial state satisfies the current goal, done!
- Choose an operator with an add list that matches goals
  - Fail if no such operator, or if it has effects that contradict the goals
- Update current goals by subtracting add list, adding preconditions
  - Fail if current goals are a subset of the new goals
- Continue search with updated goals
Initial:

State I: (on-table A) (on C A) (on-table B) (clear B) (clear C)

Goal:

Goal: (on B A) (on-table C) (on-table A)

Class Assignment: Trace this using Backward search
Problems with Backward Search

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

STRIPS is incomplete:

- Once a subgoal ordering is selected, no backtracking is allowed
- cannot find optimal solution for others, e.g. Sussman anomaly:
STRIPS and the Sussman Anomaly (1)

- achieve on(A,B)
  - put C from A onto table
  - put A onto B

- achieve on(B,C)
  - put A from B onto table
  - put B onto C

- re-achieve on(A,B)
  - put A onto B
STRIPS and the Sussman Anomaly (2)

- achieve on(B,C)
  - put B onto C

- achieve on(A,B)
  - put B from C onto table
  - put C from A onto table
  - put A onto B

- re-achieve on(B,C)
  - put A from B onto table
  - put B onto C

- re-achieve on(A,B)
  - put A onto B
HTN (Hierarchical Task Networks)

Outside Materials (see Materials page)
Hierarchical Decomposition

- Build House
  - Obtain Permit
  - Hire Builder
  - Construction
    - Pay Builder
      - Build Foundation
      - Build Frame
        - Build Roof
        - Build Walls
        - Build Interior
Task Reduction

Buy Land → Own Land
Get Loan → Have Money
Build House
Have House → Move In

Buy Land
Get Loan
Own Land
Obtain Permit
Hire Builder
Construction
Pay Builder
Have House → Move In
HTN Formalization (1)

- **State**: list of ground atoms
- **Tasks**:
  - **Primitive tasks**: do\(f(x_1, \ldots, x_n)\)
  - **Non-primitive tasks**:
    - **Goal task**: achieve\(l\) (\(l\) is a literal)
    - **Compound task**: perform\(t(x_1, \ldots, x_n)\)
- **Operator**:
  - \([\text{operator } f(x_1, \ldots, x_n) \text{ (pre: } l_1, \ldots, l_n \text{ ) (post: } l'_1, \ldots, l'_n)]\)
- **Method**: \((\alpha, d)\)
  - \(\alpha\) is a non-primitive task and \(d\) is a task network
- **Plan**: sequence of ground primitive tasks (operators)
HTN Formalization (2)

- Task network: \([(n_1 : \alpha_1) \ldots (n_m : \alpha_m), \phi]\)
  - \(n_i\) = node label
  - \(\alpha_i\) = task
  - \(\phi\) = formula that includes
    - Binding constraints: \((v = v')\) or \((v \neq v')\)
    - Ordering constraints: \((n < n')\)
    - State constraints:
      - \((n, l, n')\): interval preservation constraint (causal link)
      - \((l, n)\): \(l\) must be true in state immediately before \(n\)
      - \((n, l)\): \(l\) must be true in state immediately after \(n\)
Task Network Example

\[
\begin{align*}
  n_1: & \quad \text{achieve}[\text{clear}(v_1)] \\
  n_2: & \quad \text{achieve}[\text{clear}(v_2)] \\
  n_3: & \quad \text{do}[\text{move}(v_1, v_3, v_2)]
\end{align*}
\]
Basic HTN Procedure

1. Input a planning problem $P$

2. If $P$ contains only primitive tasks, then resolve the conflicts and return the result. If the conflicts cannot be resolved, return failure.

3. Choose a non-primitive task $t$ in $P$

4. Choose an expansion for $t$

5. Replace $t$ with the expansion

6. Find interactions among tasks in $P$ and suggest ways to handle them. Choose one.

7. Go to 2
SHOP (Simple Hierarchical Ordered Planner)

- Domain-independent algorithm for Ordered Task Decomposition
  - Sound/complete

- Input:
  - State: a set of ground atoms
  - Task List: a linear list of tasks
  - Domain: methods, operators, axioms

- Output: one or more plans, it can return:
  - the first plan it finds
  - all possible plans
  - a least-cost plan
  - all least-cost plans
Initial task list:  ((travel home park))

Initial state:  ((at home) (cash 20) (distance home park 8))

Methods (task, preconditions, subtasks):
  (:method (travel ?x ?y) ((at x) (walking-distance ?x ?y)) '((!walk ?x ?y)) 1)

Axioms:
  (:-(have-taxi-fare ?x ?y) ((have-cash ?c) (distance ?x ?y ?d) (eval (>= ?c (+ 1.50 ?d))))

Primitive operators (task, delete list, add list)
  (:operator (!walk ?x ?y) ((at ?x)) ((at ?y)))
  …
Initial state:
(at home)
(cash 20)
(distance home park 8)

Precond:
(at home)
(walking-distance Home park)
Succeed

Fail (distance > 5)

(!call-taxi home)
(!ride home park)
(!pay-driver home park)
(!walk home park)

Precond:
(at home)
(have-taxi-fare home park)
Succeed

Succeed (we have $20, and the fare is only $9.50)

Final state:
(at park)
(cash 10.50)
(distance home park 8)
The SHOP Algorithm

procedure SHOP (state $S$, task-list $T$, domain $D$)
  1. if $T$ = nil then return nil
  2. $t_1$ = the first task in $T$
  3. $U$ = the remaining tasks in $T$
  4. if $t$ is primitive & an operator instance $o$ matches $t_1$ then
     5. $P$ = SHOP ($o(S)$, $U$, $D$)
     6. if $P$ = FAIL then return FAIL
     7. return cons($o$, $P$)
  8. else if $t$ is non-primitive
     & a method instance $m$ matches $t_1$ in $S$
     & $m$’s preconditions can be inferred from $S$ then
     9. return SHOP ($S$, append ($m(t_1)$, $U$), $D$)
  10. else
  11. return FAIL
end if
end SHOP
RAPS: how to deal with uncertainty or dynamic environments

Outside Materials (see Materials page)
• **Execution monitoring** is required to maintain an up-to-date current world model. Every action executed must return some form of feedback about its success or failure.
• **RAP** encapsulates checking related to its own tasks (inferences when failure happens).
• RAP is selected for execution
• If RAP is primitive – send to hardware
• Else
  • If RAP: goal = success finish
  • Else choose task net and send to execution queue
  • RAP is set to wait till plan is executed
  • Meanwhile another RAP can execute

Dealing with interference between task nets:
Check after execution if the assumptions of other RAPs are still true, if not then fail the RAPs affected
no, I think it looks fine!

http://www.youtube.com/watch?v=GmuLV9eMTkg
Façade

Drama Manager (sequences beats)

- Bag of beats
  - Beat
  - Beat
  - Beat

- Desired value arc(s)

- Selected beat

Story World

- Player
- Trip
- Grace

Story Memory

- Current story values
  - Beat
  - Beat
  - Beat
  - Beat
- Previous action time
  - Beat

Activity not part of a beat

Natural Language Processing

- Surface text → discourse acts
- Discourse acts → reactions
Oz Project (90s)
Agent Architecture

theory of activity
( active plan tree )

subgoal indexing

plan memory

acts

sensing

the body
Goal:
open <door>
success-test:
<door> is open

Plan(sequential):
1. get <key> from <purse>
2. unlock <door> with <key>
3. open <door>
context-condition:
    have <purse> or have <key>

Goal:
get <key> from <purse>
success-test:
    have <key>

Goal:
unlock <door> with <key>
success-test:
    <door> is unlocked

Goal:
open <door>
success-test:
    <door> is open