Human Planning & Acting

- acting without (explicit) planning:
  - > when purpose is immediate
  - > when performing well-trained behaviors
  - > when course of action can be freely adapted
- acting after planning
  - > when addressing a new situation
  - > when tasks are complex
  - > when the environment imposes high risk/cost
  - > when collaborating with others

* in general, people only plan when strictly necessary → if there is no benefit, we don't plan

Planning → explicit deliberation process that chooses & organizes actions by anticipating their outcomes → conscious thought process
  ↳ aims at achieving some pre-stated objectives

AI Planning → computational study of this deliberation process

Two types of planning: (Both types complement each other)

Domain Specific

Uses specific representations and techniques adapted to each problem.

E.g. Path Planning
Perception Planning
NLG → Utterance Planning

Domain Independent → this is the focus of this course

Uses generic representations and techniques to solve the generic planning problem.

Conceptual Model for Planning: State-Transition System

- A state-transition system is a 4-tuple: $\Sigma = (S, A, E, \gamma)$, where:

$S = \{s_1, s_2, \dots\}$ is a finite or (infinite) recursively enumerable set of states

$A = \{a_1, a_2, \dots\}$ is a finite or recursively enumerable set of actions ← things an agent can do to change the state of the world

$E = \{e_1, e_2, \dots\}$ is a finite or recursively enumerable set of events ← things that happen that change the state of the world (not under an agent's control)

$\gamma : S \times (A \cup E) \to 2^S$ is a state transition function
  ↳ power set

if $a \in A$ and $\gamma(s,a) \neq \emptyset$, then a is applicable in s

applying a in s will take the system to $s' \in \gamma(s,a)$

State-Transition Systems as Graphs

* These systems cannot represent actions in parallel.

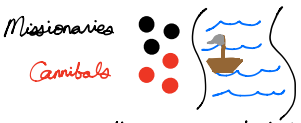
A state-transition system $\Sigma = (S, A, E, \gamma)$ can be represented by a directed labeled graph $G = (N_G, E_G)$, where:

- > nodes correspond to states in S
  - i.e. $N_G = S$
- > there is an arc from $s \in N_G$ to $s' \in N_G$,
  - i.e. $s \to s' \in N_G$ with label $u \in (A \cup E)$ iff $s' \in \gamma(s,u)$
    - ↳ s' is the result of $\gamma(s,u)$

Toy Problem: Cannibals and Missionaries

If the cannibals ever outnumber the missionaries, the missionaries will get eaten.

The boat cannot cross by itself.



There is one boat available that can hold up to two people that they would like to use to cross the river.

I will be using a short-hand notation to define the world states

Define this problem as a state-transition system.

$\Sigma = (S, A, E, \gamma)$

left of river | right of river

$S_1 = \{ \bullet\bullet\bullet \; \bullet\bullet\bullet \; \bullet \; | \; \}$ $S_5 = \{ \bullet\bullet \; \bullet \; | \; \bullet\bullet \; \bullet \}$

$S_2 = \{ \bullet\bullet \; \bullet\bullet\bullet \; \bullet \; | \; \bullet \}$

$S_3 = \{ \bullet \; \bullet\bullet\bullet \; \bullet \; | \; \bullet\bullet \}$

$S_4 = \{ \bullet \; \bullet\bullet \; | \; \bullet\bullet \; \bullet \}$

The key thing to realize is that some states of the world are not possible due to the problem itself.

this state is actually not possible; it seems that only successful states are noted in the state-transition diagram

$A = \{ 1m, 1c, 2m, 2c, 1m1c \}$

number of people on boat — type of person on the boat

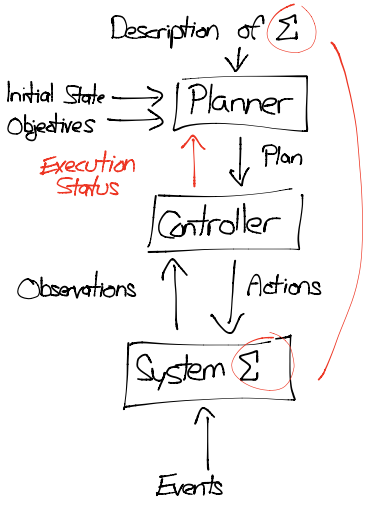
$E = \emptyset$

- A state-transition system describes all ways in which a system may evolve.
- A plan is a structure that navigates the system

  it gives appropriate actions to apply in order to achieve some objective when starting from a given state.

types of objectives:
- > goal state s_g or set of states S_g
- > satisfy some conditions over the sequence of states
- > optimize utility function attached to states
- > task to be performed (e.g. going on a holiday)

Planning & Plan Execution



≠

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

* items in red are in the domain of "dynamic planning"

Planning & Search

Toy Problems v. Real World Problems

Toy Problems
- concise & exact description
- used for illustration purposes
- used for benchmarking
- two properties:
  - > rich (not trivial)
  - > simple

Real World Problems
- no single, agreed-upon solution
- people care about the solution

Search Problems

Characterized by 4 elements:

1) initial state → starting state of the world.
2) set of possible actions / applicability conditions
   - > successor function:
     state → set of <action, state> pairs
     the function returns all actions that are applicable in state, alongside the state that results when you apply action in state.
   - > successor function + initial state = state space
     corresponds roughly to the state transition graph we mentioned earlier.
     The state space is a directed graph, with states as nodes and actions as labels on arcs.
3) goal → can be a state or a function which tests for a goal condition to be met.
   - > a solution to a search problem is simply a path from the initial state to the goal (or final) state.
4) path cost function → a function that assigns a cost value to each possible path in the search space.
   - > this component is used when finding the optimal plan (min(cost))
   - > assumption:
     path cost = sum of step costs

* as opposed to state-transition systems, search problems do not require states to be recursively enumerable.

To define all components of a search problem is its own problem:

problem formulation

Essentially, we have to decide what actions we want to consider and what states you want to consider.

This involves picking an appropriate level of abstraction through which to look at the world.

↳ most important decision!

To manage the complexity of problem formulation, we will make some simplifying assumptions about the environment:

- > finite and discrete states only
- > fully observable
- > deterministic → for every state and action, there is at most one successor
- > static (no events)
- > restricted goals (no goal functions)
- > sequential plan solutions only → our solution is a linear list of actions (no parallel activity)
- > implicit time → actions have zero duration
- > offline planning → the state transition system that underlies our planning does not change while we are planning

Search Algorithms

Data structure that we manipulate during the search process: **Search Nodes**

The search nodes are the nodes in the search tree.

→ it is truly a bookkeeping structure that encapsulates a state.

The data structure is as follows:

- state: a state in the state space corresponds to a world configuration
  - we will use trees for simplicity
  - * two search nodes may contain the same state!
    - For the case of graphs and graph search, we need a Hash Table to avoid searching through a cycle.
- parent node: the immediate predecessor in the tree
  - the root node is the only node without a parent
- action: the action that gets us from the parent node's state to this state.
- path cost: the total cost of the path leading to this node
- depth: the depth of this node in the search tree

General Tree Search Algorithm:

the problem structure as defined previously

```
function treeSearch(problem, strategy):
    fringe := { new searchNode(problem.initialState) }
    loop
        if empty(fringe) then return FAIL
        node := selectFrom(fringe, strategy)
        if problem.goalTest(node.state) then
            return pathTo(node)
        fringe := fringe + expand(problem, node)
```

→ this is a set

potentially infinite

strategy determines how a node is selected from the fringe

expansion is done using the successor f'n applied to the current node's state.

a new node is created for each resulting state.

Subtleties of this algorithm:

→ Given the following tree...

...the algorithm will never finish.

→ The algorithm's performance depends heavily on the application of the strategy that is used.

the method selects the next node to be expanded on the fringe.

ideally, this decision can be done in constant time

an effective method for scheduling the application of the successor function to expand nodes.

Our selection should aim to find the goal state as quickly as possible.

This selection determines the order in which nodes are expanded.

Example criteria of selection:

- LIFO/FIFO queue
- Alphabetical Ordering

LIFO → Depth-first search

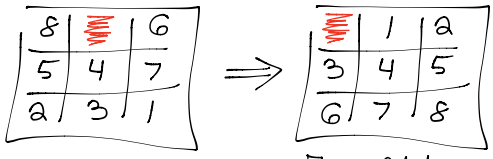
FIFO → Breadth-first search

The general search algorithm is considered deterministic if the search strategy is deterministic. Without the strategy, it is considered a non-deterministic algorithm.

Example Problems:

> Sliding Block Puzzle: (toy problem)

Path-cost is assumed to be 1 unit cost per unit of action; i.e. it costs 1 to slide the tile once.



Random initial state → Goal State

actions: depends on the config. of the tiles. There are four basic actions:

{move-up, move-left, move-down, move-right}

Executing these actions depends on the empty tile at the moment of consideration.

> n-queens problem: (toy problem)

Place n queens on an n × n chess board such that none of the queens attacks any of the others

We can set this up as a planning problem by thinking of the problem as one of "placement actions"

Initial State: Empty chessboard
Actions: Placing a queen anywhere on the board
Goal State: Any configuration of the board where the queens don't attack each other

> The Dock Worker Robots (DWR) Domain (real-world problem)

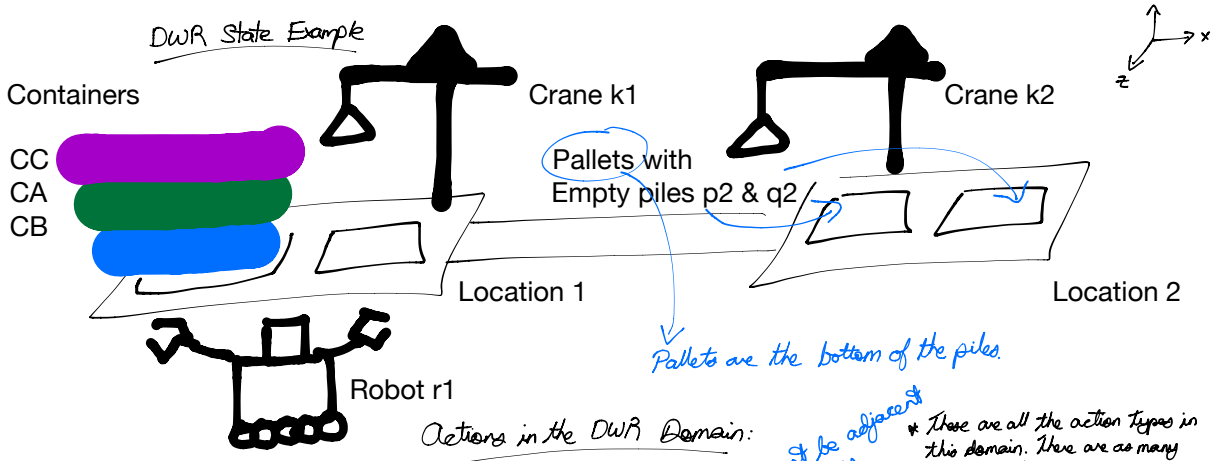
Informal Description:

environment
- harbour w/ several locations (docks)
- docked ships
- storage areas for containers
- parking areas for trucks & trains

actors
- cranes to load/unload ships (and other objects)
- robot carts to move containers around

* What we're interested in are plans for cranes and robots that achieve certain container configurations that we designate as goal states.

DWR State Example



Pallets are the bottom of the piles.

Actions in the DWR Domain:

robot action: *move*(robot, loc, loc') — must be adjacent locations

crane locations:
- *take*(container, crane, pile, location)
- *put*(container, crane, pile, location)
- *load*(container, crane, robot, location)
- *unload*(container, crane, robot, location)

* These are all the action types in this domain. There are as many actions as there are unique ways to instantiate them.

Constraints:
> robot can only carry one container at a time

AI Planning in Context

NONLIN → used for plans for power stations
DEVISER → action sequences for Voyager spacecraft
NONLIN HTN → used for early Japanese robot research
O-PLAN → used for European space missions
also used widely for emergency response & search and rescue operations

Practical AI Planners

- STRIPS, Fikes & Nilsson 1971 ⇒ Mobile Robot Control
- HACKER, Sussman 1973 ⇒ Simple Program Generation
- NOAH, Sacerdoti 1977 ⇒ Mech. Engineers Apprentice Sys.
- NONLIN, Tate 1977 ⇒ Electricity Turbine Overhaul
- O-PLAN, Currie & Tate, 1991 ⇒ Search and Rescue Ops.

used to control SHAKEY

Course Readings:

> Review of AI Planners to 1990: Hendler, Tate, and Drummond: "AI Planning: Systems and Techniques"

> Knowledge-Based Planners: Wilkins, and desJardins: "A Call for Knowledge-based Planning"

> O-Plan Paper: Tate and Dalton "O-Plan: a Common Lisp Planning Web Service"

About the NONLIN planner: (1974-1977)

- * Was an HTN planner
- Was also a POP planner
- NONLIN searched for solutions in the space of plans
- Had ways for answering the question of whether a proposition has a certain truth value at a certain point in the plan → this is known as a QA (Question-Answering) module; today more formally known as the model-truth criterion
- Had condition "types" to limit search
- * Had structure-based planning

* denotes features kept for O-Plan

About the O-Plan planner: (1983-1999)

- Followed on from NONLIN and had many of the same features
- Had tools for knowledge elicitation and modeling
- Supported rich plan representation and use
- Dynamic issue handling
  - > New objectives/tasks could arise while planning & execution is taking place.
- Plan repair
  - > done when time was available or the situation demanded rapid response
- Had interfaces for users with different roles
  - > several different users could be refining the plan simultaneously
  - > they could also share information about the plan between them

Typical Features of Practical AI Planners

Practical planners typically support these features:

- Hierarchical Task Network Planning
- Partial Order Planning
- Rich Domain Model
- Integration with other systems

Question
- What is the difference between refinement planning and partial order planning?