Intelligent Systems: Reasoning and Recognition

James L. Crowley

MOSIG M1 Second Semester 2019/2020

Lesson 18 9 April 2020

Planning and Problem Solving

Planning

Problem Spaces

Planning is formalized using a Problem space.

A problem space is defined as

1) A set of states {U} (the Universe),
2) A set of operators for changing states {A} (Operations or Actions).

A state is defined using a conjunction of predicates.

A problem is {U}, {A} plus

an initial state $i \in \{U\}$

a set of Goal States $\{G\} \subset \{U\}$

A plan creates a sequence of actions A_1, A_2, A_3, A_4,… that lead from the state S to one of the states $g \in \{G\}$

With this approach, problem solving is formalized as Planning: The search for a sequence of actions leading to a goal.

The core concept is the notion of "State".
A state is a "partial" description of the environment represented as a conjunction of predicates.

State: A conjunction of predicates (truth-valued functions) over entities.

A state is defined as a conjunction of predicates whose arguments are entities.
Entities represent perceived phenomena. Predicates represent relations.
Relations associate entities.

Predicates express relations that associate entities (spatial, temporal, part-of, category inclusion, etc). Predicates can be negated. The negation of a predicate is a predicate.

Blocks World

Blocks world is an abstract, toy world for exploring problems of reasoning and intelligence. We will use Blocks world to illustrate different principals and techniques concerning knowledge representation.

Blocks world is composed of:

- A table
- A set of blocks
- An agent (robot hand) that can act on (move) the blocks

The blocks, tables and hand are the primitive concepts that make up a blocks world. They are primitive because they are directly perceivable.

Classic Definition:
1) A universe composed of a set of cubic blocks and a table
2) Blocks are mobile, the table is immobile
3) The agent is a mobile hand,
4) A block can sit on a table, on another block, or in the hand.
5) There cannot be more than one block on another block
6) The table is large enough for all blocks to be on the table.
7) The hand can move only one block at a time.

The state of the universe is formalized using predicates.
Blocks are represented by Capital Letters {A, B, C, …}
Variables (lower case letters) represent sets of blocks
This are specified by Quantifiers: for-all x (∀x), There-exists y: (∃y)

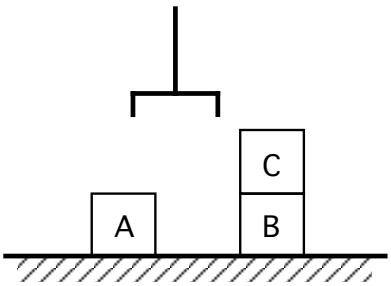
Predicates

Typical predicates used to define states in blocks world are

On(y, x)	O(y,x)	Block x is on Block y
OnTable(x)	OT(x)	Block x is on the table.
Held(x)	H(x)	Block x is in the hand.
Free(x)	F(x)	No block is On x : ¬∃y: (On(y, x)) or ∀y: (¬On(y, x))
HandFree()	HF()	The hand is empty, or ¬∃x (H(x))

Predicates are complex concepts that build on the primitive concepts.

For example: HF ∧ OT(A) ∧ OT(B) ∧ On(B,C) ∧ F(C) ∧ F(A)



This is an example of a situation. We will look schema for representing and reasoning about situations in the coming weeks.

Actions

The system can move from one state to another by performing actions.
Actions are typically represented as condition-action rules.
Actions can be defined by arguments, Preconditions, and Post-conditions.

Action (<blocks>)

Preconditions: Predicates that must be true to execute the action

Post-conditions: Predicates that are made be true or false by the action

Nilsson defined four actions for blocks world.

Grasp(x):

Precondition: HF() ∧ F(x) ∧ OT(x)

Postcondition: ¬HF() ∧ ¬OT(x) ∧ H(x)

Pose(x):

Precondition: H(x)

Postcondition: ¬H(x) ∧ HF() ∧ OT(x)

Stack(x, y): Stack block x on block y

Precondition: H(x) ∧ F(y)

Postcondition: ¬H(x) ∧ ¬F(y) ∧ F(x) ∧ O(y, x) ∧ HF()

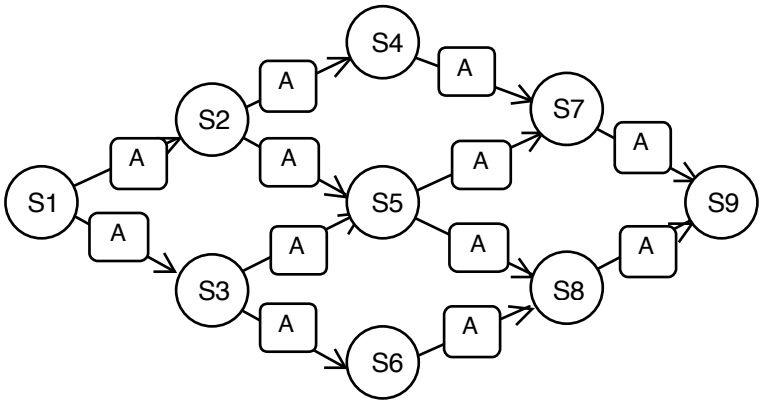
Unstack (x, y)

Precondition: F(x) ∧ O(y, x) ∧ HF()

Postcondition: ¬O(y, x) ∧ ¬HF() ∧ H(x) ∧ F(y)

As defined by Nilsson, in Blocks World all precondition predicates are rendered false by execution of an action. However, this is not true of most problem domains.

To solve a problem we search for a path through the state space from an initial state to a target state.



We will use blocks world in the coming lectures to illustrate various techniques.

Comments on Blocks World and Search

Note that blocks world is a "Closed" world. It has a finite number of states.
Real problems tend to be open, with a very large branching factor (possible actions) and an infinite number of states.

Planning as Search

The "paradigm" for planning is "Generate and Test".
Planning is the generation of a sequence of actions to transform i to a state g ∈ {G}
Planning requires search for a path through a graph of states.

Nils Nilsson defined a taxonomy of graph search algorithms includes the following
1) Breadth first search
2) Depth first search
3) Heuristic Search
4) Hierarchical Search

Nilsson unified Depth-First, Breadth First and Heuristic Search a single algorithm named GRAPHSEARCH.

The GRAPHSEARCH algorithm requires maintaining a list of "previously visited" states {C} (Closed list) and a list of available states to explore {O} (Open list).
As each state is tested, its neighbors are generated and added to the open list.
The state is then added to the closed list.

GRAPHSEARCH:

{O}←intial state
WHILE {O} ≠ empty DO
    Extract a new state s from {O}
    IF s ∈ {G} THEN halt ELSE add s to {C}
    Generate all neighbor states {N} of s.
    ∀ n ∈ {N} IF n ∉ {C} THEN add n to open states {O}.

Breadth-first, depth-first and heuristic search are all variations on the same GRAPHSEARCH algorithm, depending on whether the Open list is a stack, queue or sorted.

1) Breadth first search - The Open list {O} is a Queue (First In, First Out)
2) Depth first search - The Open list {O} is a Stack (First In, Last Out)
3) Heuristic Search - The open list {O} is sorted by a Cost f(s) = g(s) + h(s)

Algorithmic Complexity of Search for Planning

We estimate algorithm complexity with the Order operator O().
Algorithm complexity order is equivalent for all linear functions.

$O(AN+B) = O(N)$

The algorithm complexity of graph search depends on

b: The branching factor; The average number of actions {A} possible in a state.
$b = E\{card(\{A\})\}$ (E{} is expectation)
d: Depth. The minimum number of actions from an initial state to a goal state.

Breadth First search: {O} is a queue (FIFO)

For breadth first search, finding the optimal path requires exhaustive search.
Computation Cost $O(b^d)$, memory $O(b^d)$.

This is a problem for humans, because human cognition has an important physiological limit: The size of working memory.

To use breadth first planning a human must use a memory aid, such as a note pad.
Clearly human's use something more clever.

Depth First search: {O} is a stack (LIFO).

For depth first search, finding the optimal path requires exhaustive search, however
Computation Cost $O(b^d)$, memory $O(d)$.

This can be done by humans for limited depth problems. ($d \leq 7$)
Depth first requires setting a maximum depth d_{max}.

However, both Breadth first and Depth search are exhaustive!
A cost of $O(b^d)$ is not acceptable for most real world problems.

Heuristic search: {O} is sorted based on the cost f(s) of a path through the state.

For Heuristic search, we reduce the order by reducing the branching factor:
This give computation and memory costs of $O(c^d)$ where $c \leq b$.

Heuristic Search is NOT exhaustive. The search avoids unnecessary branches.

For heuristic search, the cost of path through a state is f(s) = g(s) + h(s) where
Where g(s) is the cost of a path from the start state to the state s (easily calculated)
And h(s) is the cost of a path from the state to the goal state (unknown)

$$f(s) = g(s) + h(s)$$

Because h(s) is not known, it must be approximated with an estimate h*(s)

Optimality requires that both the cost function g(s) and the estimate, h*(s) meet the "optimality conditions".

Nilsson's Conditions for Optimal Search

Nilsson demonstrated the conditions under which a heuristic search algorithm is guaranteed to find the "optimal" plan.

Notation :

i : initial state
g : goal state $g \in \{G\}$
$k(s_i, s_j)$: the minimal theoretical cost between states s_i and s_j
$g^*(s) = k(i, s)$: The true cost of the shortest path from i to s.
$h^*(s) = k(s, g)$: The true cost of the shortest path from s to $g \in \{G\}$.

$f^*(s) = g^*(s) + h^*(s)$ The true cost of the shortest path from i to g passing by s.

Problem: If we do not know the shortest path, how can we know h*(s)?
Solution estimate the costs.

Define:

g(s) : estimated cost from i et s.
h(s) : estimated cost from s to g
f(s) = g(s) + h(s) estimated total cost of a path through s

Nilsson showed that whenever the estimated cost $f(s) \leq f^*(s)$,
the first path that is found from s_1 to s_2 will always be the shortest.
Thus g(s)=g*(s) because the first path from i to s has the least cost.

$f(s) \leq f^*(s)$ requires two conditions:

Condition 1: that the heuristic UNDER-ESTIMATES the cost.

$$h(s) \leq h^*(s)$$

Condition 2: that estimate h(s) is "monotonic". That is :

$$h(s_i) - h(s_j) \leq k(s_i, s_j)$$

This is almost always true whenever $h(s) \leq h^*(s)$!!

Nilsson called this the A* condition.
A* is "optimal" because the first path found is the shortest path.

Cost and Optimality of Heuristic Search

A key problem is defining heuristic search is the concept of numerical "cost" for executing an action.
Cost, $k(s_i, s_j)$, can be any numerical value, but must respect the optimality conditions for GRAPHSEARCH to be optimal.

Examples of cost: distance, time, Euros, risk, number of actions.

Whenever the cost metric is proportional to the length of the path, then Euclidean distance to the goal provides an "optimal" heuristic!

This is true for scalar multiples of distance, for example, time traveled or risk or cost or energy expended. (assuming constant speed, time = distance x 1/speed)

Note that for $h(s) = 0$, $h(s)$ meets the optimality condition because $h(s) \leq h^*(s)$!!

This is dijkstra's algorithm, used for network routing.
We can speed up the search by using a better h(s) that 0! However, the first solution found is always the best solution.

Hierarchical Planning, Subgoals and Chunking

Cost of Search vs Optimality of Result

Note that there are TWO notions of cost!

1) Cost of executing the resulting plan.
2) Algorithmic complexity of the search (computational cost).

"Optimal" search means that the resulting plan is the cheapest possible plan.

For many real problem domains, the cost of search is MORE EXPENSIVE than the cost of executing the resulting plan.

In solving problems, humans sacrifice optimality to reduce the effort required for planning.

Subgoals

Subgoals can strongly reduce the algorithmic complexity search for a path through a network. A "Subgoal" is an abstract state that represents a set of states.

For example, to plan a route from campus to the train station.
1) Plan a route from Campus to the Quai d'Isere
2) Plan a route from the Quai d'Isere to the train station.

Define:

d : the minimal number for actions from the start state, i, to a goal state, g.
d_{is}: the minimal number for actions from the start state, i, to a subgoal s.
d_{sg}: the minimal number for actions from the subgoal s, to a goal state.

The subgoal divides the problem into two smaller subproblems: Plan a path from it to s, and plan a path from s to g.

$$O(b^{d_{is}} + b^{d_{sg}}) = O(b^{\max\{d_{is}, d_{sb}\}})$$

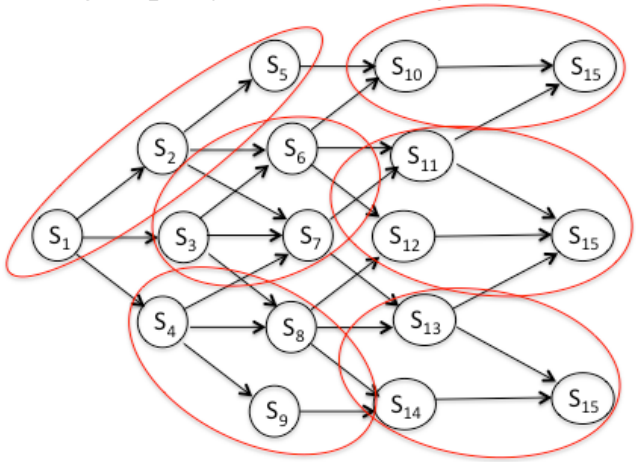
if $\max\{d_{is}, d_{sb}\} < d$, then the complexity is reduced.

Hierarchical search sacrifices autonomy for scope.
Hierarchical search provides solutions to more complex problems, but there is no guarantee that the resulting solution is "optimal".

Two possible approaches to hierarchical planning are
1) Build a hierarchy of super-states
2) Define a hierarchy of operators. (meta-operators)

Hierarchy of states

An obvious approach is to group adjacent states together to form Super states.



Korf set this up as an optimisation problem and found that an "optimal" size for super-states was "e" (2.71828...)

Possible ways to group states

1) Group sets of states connected by a single action to a privileges state
2) Define arbitrary connected sets of states
3) Group states by eliminative predicates.

A state is a conjunction of predicates $P_1() \wedge P_2() \wedge P_3() \wedge P_4()$

The state $P_1() \wedge P_2()$ represents the states $P_1() \wedge P_2() \wedge P_3()$ AND $P_1() \wedge P_2() \wedge \neg P_3()$

We can group states by deleting predicates and divide states into substates by adding predicates.

Sussman tried this with Blocks world and found that it was dependent on which states were eliminated. For example replacing

On(B,C) ∧ On(A,B) ∧ OT(A) with On(B,C) is not helpful.

Operators

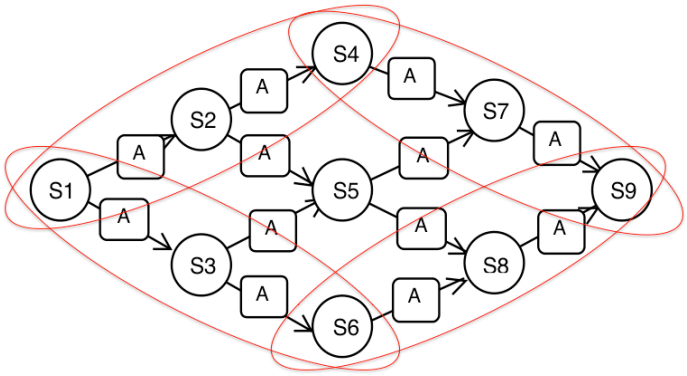
A more effective means is to group sequences of states and actions into "operators"

$S_1 - A_{12} \rightarrow S_2 - A_{23} \rightarrow S_3 - A_{34} \rightarrow S_4$

Is an operator to go from S_1 to S_4

The States translate to perceptual actions P() to verify the results of each action and to verify the pre-condition for the next action.

Operator $(S_1, S_4) = P(S_1) - A_{12} \rightarrow P(S_2) - A_{24} \rightarrow P(S_4)$



Human's do this to reduce the load on short term working memory. Simon called this "Chunking".

Operators are composed hierarchically, as needed, to accommodate the limits of human working memory.

Chunking

To speed up search and to overcome limits to short term memory, humans use a technique called "chunking".

Chunking is a process by which individual pieces of information are bound together into a meaningful whole. A chunk is a concept that represents a collection of concepts.

Chunking states into sets of states enables hierarchical planning.
Chunking can be applied hierarchically, with groups of states at each level represented by a single at the next level.

Example: Travel Planning

Consider the problem of planning a trip to Oxford.
First we choose whether to go by plane, train or bus.
Choosing plane, we select company and flights : Air France LYS-CDG-LHR
Then we plan the trip to Lyon Airport: Bus, Train or car
Then we plan the trip from Heathrow: Bus, Train or car
Then we plan the trip to the train station to catch the bus to LYS, etc.