Intelligent Systems: Reasoning and Recognition

James L. Crowley

ENSIMAG 2 and MoSIG M1 Winter Semester 2010

Lecture 3 10 February 2010

Problem Solving and Planning as State Space Search

The Intelligent Agent 2

Planning and Problem Solving 2

Blocks World 4

Planning as Graph Search 6

State Spaces 8

Categories of Graph Search 10

Algorithmic Complexity 11

Nilsson's GRAPHSEARCH Algorithm 12

The Intelligent Agent

To provide a formal basis for studying intelligence, Nils Nilsson has proposed the Intelligent Agent as a fundamental concept for formalizing intelligence.

The Intelligent Agent has 3 components: (A, B, C)
A) Actions; The ability to act; A physical body;
B) Goals. (In French "Buts")
C) Knowledge; The ability to choose actions to accomplish goals.

The "Intelligent Agent" acts based on the principle of Rationality.

Rational behavior: Actions are chosen to accomplish goals

Nilsson proposed to define intelligence as rationality:

Rational Intelligence: The ability to choose actions to accomplish goals.

An agent is intelligent if it 1) can act, 2) has goals, and 3) Can choose its actions to accomplish it's goals.

Rational intelligence leads to a formulation of intelligence as problem solving and planning.

Planning and Problem Solving

Planning: The search for a sequence of actions leading to a goal.

Rationality leads to a formulation of intelligence as planning
Rational intelligence is formalized using a Problem space.

A problem space is defined as
1) A set of states {U},
2) A set of operators for changing states {A} (Actions).

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 $g \in \{G\}$

States: A state, s, is a "partial" description of the real universe.

A state is defined as a conjunction of predicates (Truth functions) based on measured (observed) values. The measured values are called "observations".

Examples:

Mobile Robotics: Near(x, y, t)

Blocks World: OnTable (A) ∧ On (A, B) ∧ HandEmpty

Blocks World

Blocks world is an abstract, toy world for exploring problems. Blocks world is a "Closed" world. It has a finite number of states.

Blocks world is composed of a finite number of blocks in a finite number of states.

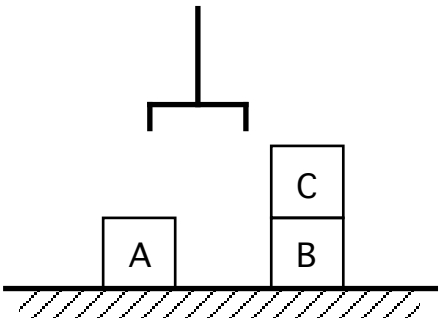
Blocks world is composed of:

- A set of blocks
- An agent that can act on blocks to change their state

Classic Definitions:
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 first order predicates.

For example:



Blocks: { A, B, C }

Predicates:

On(A, B)	S(A, B)	Block A is On Block B.
OnTable(A)	OT(A)	Block A is On the Table.
Held(A)	H(A)	Block A is in the hand.
Free(A)	F(A)	No block is On A : ¬∃x (On(x,A)) or ∀x (¬On(x,A))
HandFree	HF	The hand is empty, or ¬∃x (H(x))

The state of the universe is expressed as a conjunction of predicates:

HF ^ OT(A) ^ OT(B) ^ O(C, B) ^ F(C) ^ F(A)

Actions:

Actions are state change operators. Actions are atomic.

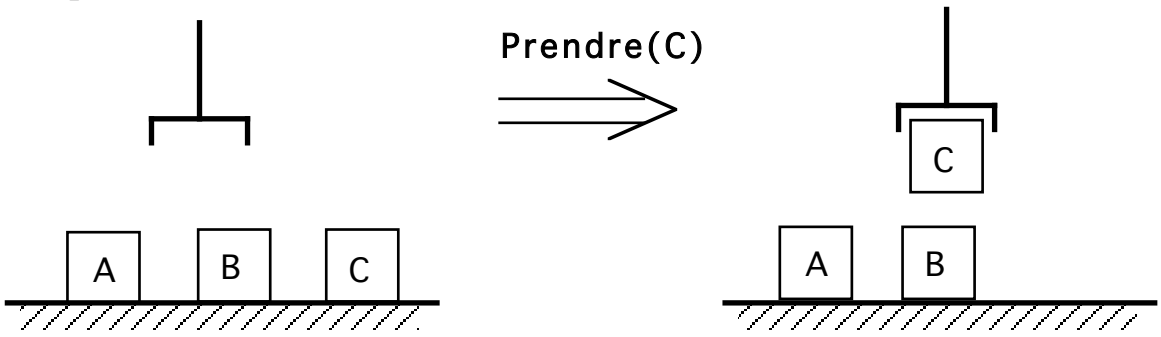
Nilsson proposed to formalize actions with STRIPS :
(Stanford Research Institute Problem Solver) (1971).

Principle: explicitly list all state changes.

Action: Name(Variables)

Precondition: Must be true for the action to operate
Retract: rendered false by the action
Add: rendered true by the action.

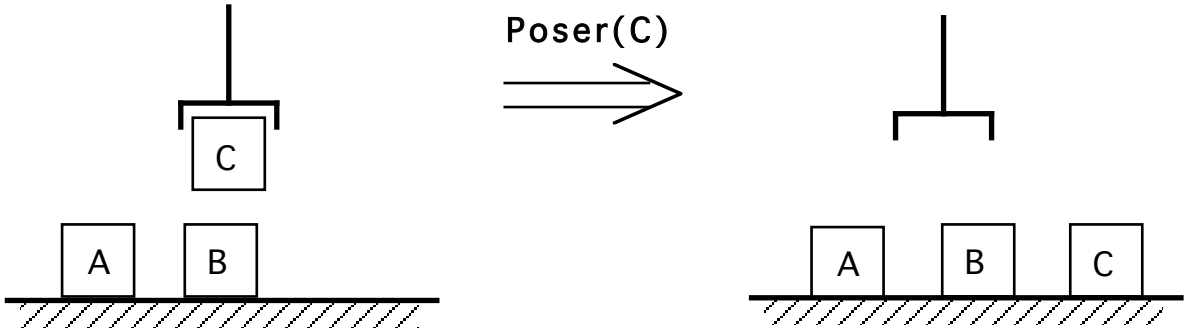
Grasp(x)



Grasp(x)

Pre: HF ^ F(x) ^ OT(x)
Ret: HF ^ F(x) ^ OT(x)
Add: H(x)

Pose(x)

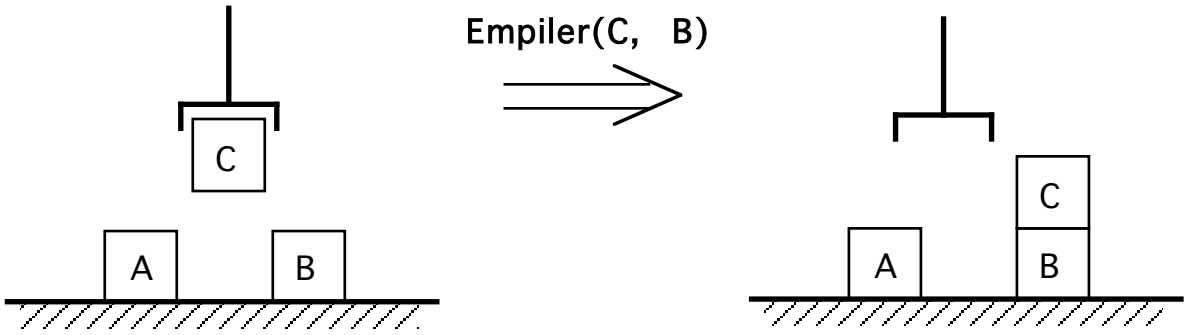


Pose (x)

Pre: H(x)
Ret: H(x)

Aj: F(x) ∧ OT(x) ∧ HF

Stack(x, y)



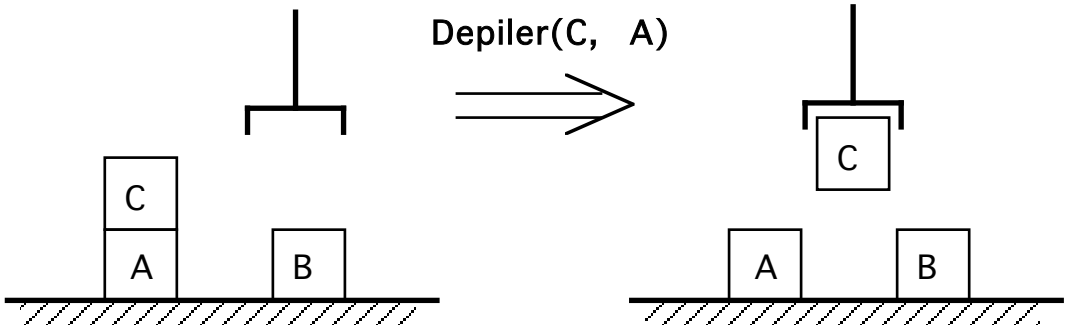
Stack (x, y)

Pre: T(x) ∧ F(y)

Ret: T(x) ∧ F(y)

Add: F(x) ∧ O(x, y) ∧ HF

Unstack (x, y)



Unstack(x, y)

Pre: F(x) ∧ O(x, y) ∧ HF

Ret: F(x) ∧ O(x, y) ∧ HF

Add: T(x) ∧ F(y)

Question: Why do we need Pose(x). Is not Stack(x, table) equivalent?

Response: If we execute Stack(x, table) the predicate Free(table) is not true.

Planning as Graph Search :

A problem is defined by

a universe, {U},

an initial state, i

A set of Goal states, {G}.

Planning is the generation of a sequence of actions to transform i to a state g ∈ {G}

The "paradigm" for planning is "Generate and Test".

Given a current state, s
1) Generate all neighbor states {N} reachable via 1 action.
2) For each n ∈ {N} test if n ∈ {G}. If yes, exit
3) Select a next state, s ∈ {N} and loop.

Planning requires search over a graph for a path.

A taxonomy of graph search algorithms includes the following

1) Depth first search
2) Breadth first search
3) Heuristic Search
4) Hierarchical Search

The first three are unified within the GRAPHSEARCH algorithm of Nilsson.

Graph searching has exponential algorithm complexity.
"knowledge" can be used to reduce the complexity.

State Spaces

In the absence of domain knowledge, an agent needs to explore the entire state space to discover the shortest path from i to {G}.

Domain knowledge can be used to guide this search.

To illustrate this, consider the problem of path planning for a mobile robot.

Navigation requires a "map". The classic map for path planning is a "network of places".

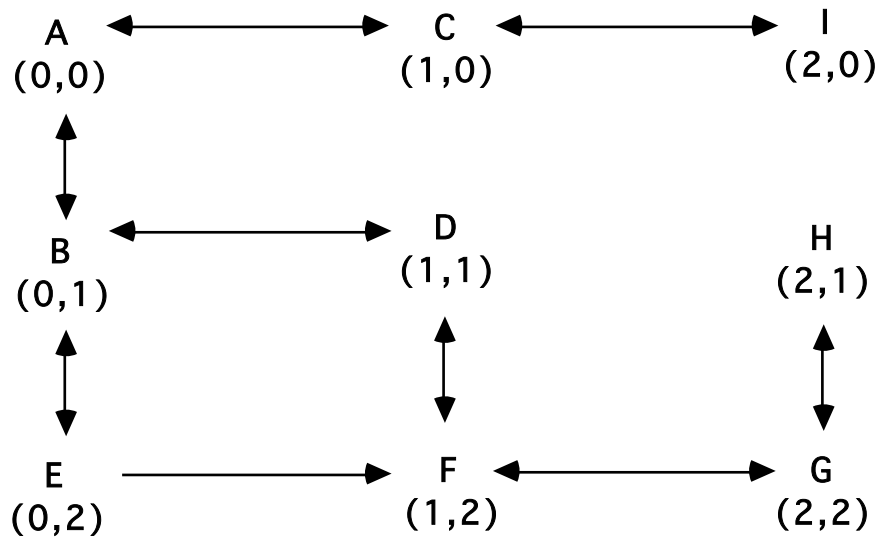
A place is defined as
1) A name
2) An inclusion test (a predicate At(x))
3) A list of "adjacent" places that can be reached by a single action.

The set of places compose a network. ("A network of places").

Navigation planning requires finding a sequence of places that lead from i to {G}.
The search for a path generates a tree of possible paths.

There are 3 forms of search algorithms that can be used to generate this tree:
Depth first, breadth first, and heuristic.

Let us use the following graph to illustrate these three algorithms.
Note that the places are labeled with coordinates (x,y) on a Cartesian map.
This allows us to define a metric to evaluate the "cost" of alternative paths.



Given that the robot is at place "E" and we require it to go to H.

Terminology and notation:

1) A graph is composed of nodes and arcs.
2) Arcs are directed: n_i --> n_j
2) Paths through the graph are represented by a tree.
3) Within the tree:
   n_j is the successor (or daughter) of n_i,
   n_i is the predecessor (or father) of n_j.

Categories of Graph Search

Breadth first, depth first and heuristic search are all variations on the same 4 step algorithm. The algorithm requires maintaining a list of "previously visited" nodes {C} (Closed list) and a list of available nodes to explore {O} (Open list).

$n \in \{G\}$

Given a node s:
1) Create the list of successors of s: {N} = Neighbors(s)
FOR ALL $n \in \{N\}$
2) Eliminate any previously visited nodes from {N}
IF $n \in \{C\}$ THEN {N} := {N} – n
ELSE
3) If Not, then Add n to the closed list
{C} <- {C} + n
4) Test if n is a goal node:
If $n \in \{G\}$ then EXIT with Success
5) Add n to the open list
{O} <- {O} + n
END
6) Choose a node, s, from {O}. Remove s from {O}. Go to 1.

Steps 5 (and 6) determine the nature of the search:

Breadth First search: {O} is a queue (FIFO)
Depth First search: {O} is a stack (LIFO).
Heuristic search: {O} is sorted based on a cost, f.

Algorithmic Complexity

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 neighboring states {N}
$b = E\{card(\{N\})\}$ (E{} is expectation)
d: Depth. The minimum number of nodes from i to {G}.

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 $O(b^d)$.

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).
However, depth first requires setting a maximum depth d_{max}.

Heuristic search: {O} is sorted based on a cost, f.

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

Heuristic Search is NOT exhaustive. We avoid unnecessary branches.

Nilssons GRAPHSEARCH provides Heuristic search in two forms.

Algorithm A : uses an arbitrary cost estimate.
Algorithm A* : uses an "optimal" cost estimate to produce "optimal" search.

A^* requires that the cost function and cost estimate meet the "optimality conditions".

Hierarchical Search: Hierarchical search sacrifices optimality to reduce the depth by dividing search into two parts d_1 and $d_2 < d$. $d_1 + d_2 \geq d$

$O(c^{Min(d1, d2)})$

Nilsson's GRAPHSEARCH Algorithm

Symbols :

T : Search Treee
G : Set of Goal States
S : Departure State
M : List of Neighbor States
Open : List of Open States
Closed : List of Closed States
n, e : Nodes representing states

Algorithm :

1) Create T, Open and Closed, (initially empty).
2) Place S in Open, and as root to T.
3) LOOP: if OPEN then EXIT with failure
4) Extract n from Open Open <- Open - n, Closed <- Closed + n

5) If n is an element of G, then
Construct a Solution stack with states from N to S.
Unstack solution stack. This is best path.
EXIT with Success.

6) M gets neighbors of n M <- Neighbors(n)

7) For each e in M
IF e is in CLOSED, then Remove M <- M - e.
else
a) For A and A* :Calculate cost f(e) of path from s to G through e
$f(e) = g(e) + h(e) = K(s,e) + h(e)$
b) add e to Open as LIFO (depth first), FIFO (breadth First), sorted list (A*
c) add 2 to T as successor to n

8) Go to step 3.