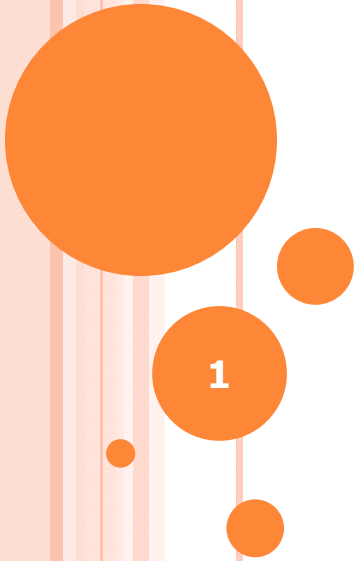
Unit 3



Constraint Satisfaction Problems

ACKNOWLEDGEMENT

This Ppt's are prepared using

1. Stuart Russell and Peter Norvig (1995), "Artificial Intelligence: A Modern Approach", Third edition, Pearson, 2003.

2. Parag Kulkarni and Prachi Joshi, "Artificial Intelligence Building Intelligent Systems", PHI learning Pvt. Ltd., ISBN 978-81-203-5046-5, 2015. 5.

3. Online source

Outline

- Constraint Satisfaction Problems (CSP)
- Backtracking search for CSPs
- Local search for CSPs

CONSTRAINT

- Constraint is something that restricts movement, arrangement, possibilities and solution
- It is a sort of bottleneck
- It is mathematical/ logical relationship among the attribute of one or more objects.

CONSTRAINT SATISFACTION PROBLEMS (CSPS)

- Standard search problem:
  - It is represented by an arbitrary data structure that can accessed by only problem specific routines: successor function, heuristic function, and goal test
- CSP:
  - state is defined by variables X_i with values from domain D_i
  - Set of constraints C1, C2, …., Cn
  - A state of the problem is defined by an assignment of values to some or all of the variables, *{Xi = vi, Xj =vj, ..}*
  - An assignment that does not violate any constraints is called a consistent or legal CONSISTENT assignment.
  - goal test is a set of constraints specifying allowable combinations of values for subsets of variables
- Allows useful general-purpose algorithms with more power than standard search algorithms

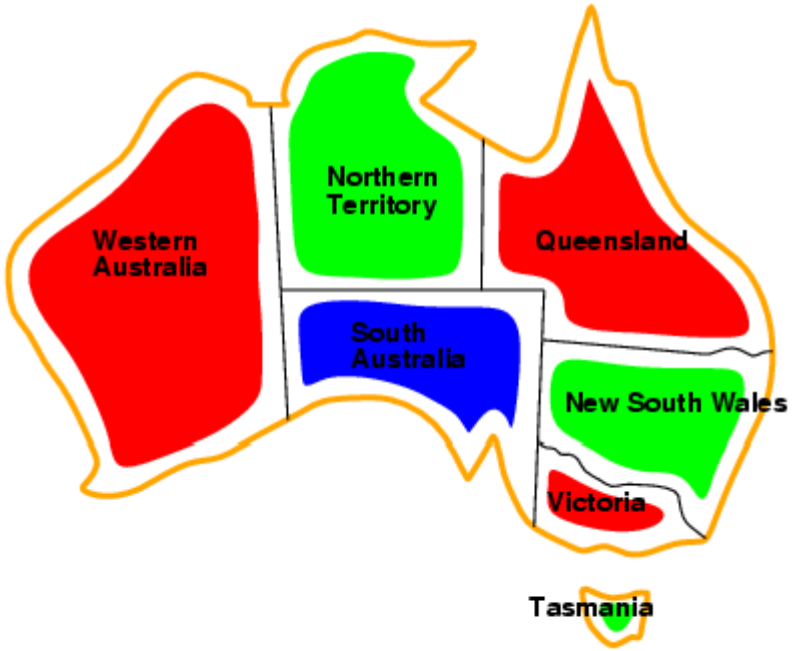
Advantage of representing a problem as CSP

- Standard and known pattern
- Helps to develop generic heuristic hence limit the need for domain specific experts
- Some proven methods can be used (ex. Constraint graph)

EXAMPLE: MAP-COLORING

- Variables *WA, NT, Q, NSW, V, SA, T*
- Domains D_i = {red,green,blue}
- Constraints: adjacent regions must have different colors
- e.g., WA ≠ NT, or (WA,NT) in {(red,green),(red,blue),(green,red), (green,blue),(blue,red),(blue,green)}

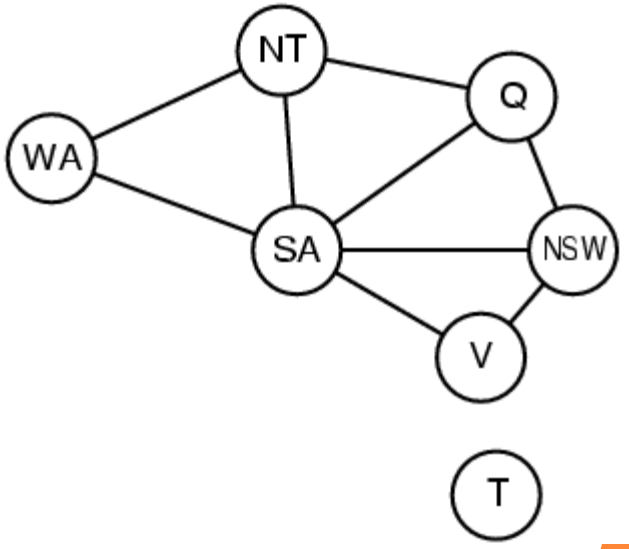
EXAMPLE: MAP-COLORING



- Solutions are complete and consistent assignments, e.g., WA = red, NT = green,Q = red,NSW = green,V = red,SA = blue,T = green

CONSTRAINT GRAPH

- Binary CSP: each constraint relates two variables
- Constraint graph: nodes are variables, arcs are constraints

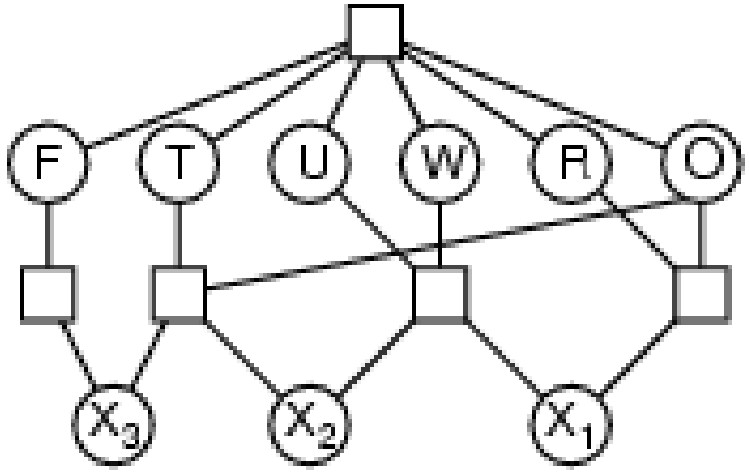


Varieties of constraints

- Unary constraints involve a single variable,
  - e.g., SA ≠ green
- Binary constraints involve pairs of variables,
  - e.g., SA ≠ WA
- Higher-order constraints involve 3 or more variables,
  - e.g., SA ≠ WA ≠ NT

EXAMPLE: CRYPTARITHMETIC

$$\begin{array}{r} \text{T W O} \\ +\ \text{T W O} \\ \hline \text{F O U R} \end{array}$$



- Variables: $F\ T\ U\ W\ R\ O\ X_1\ X_2\ X_3$
- Domains: $\{0,1,2,3,4,5,6,7,8,9\}$
- Constraints: $Alldiff\ (F,T,U,W,R,O)$
  - $O + O = R + 10 \cdot X_1$
  - $X_1 + W + W = U + 10 \cdot X_2$
  - $X_2 + T + T = O + 10 \cdot X_3$
  - $X_3 = F,\ T \neq 0,\ F \neq 0$

Cryptarithmetic Problem

A. Digit ranges from 0 to 9 only.

B. Each Variable should have unique and distinct value.

C. Each Letter, Symbol represents only one digit throughout the problem.

D. You have to find the value of each letter in the Cryptarithmetic.

E. There must be only one solution to the problem.

F. The Numerical base, unless specifically stated , is 10.

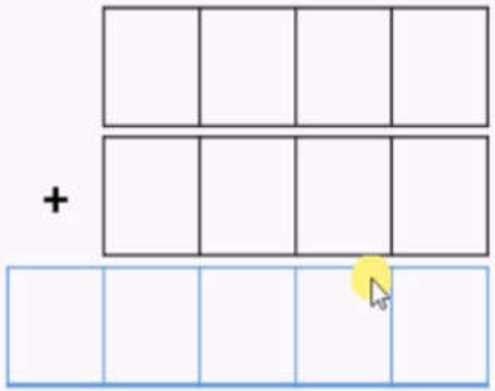
G. Numbers must not begin with zero i.e. 0123 (wrong) , 123 (correct).

H. After replacing letters by their digits, the resulting arithmetic operations must be correct.

Cryptarithmetic Problem

	C3	C2	C1	
	S	E	N	D
+	M	O	R	E
M	O	N	E	Y

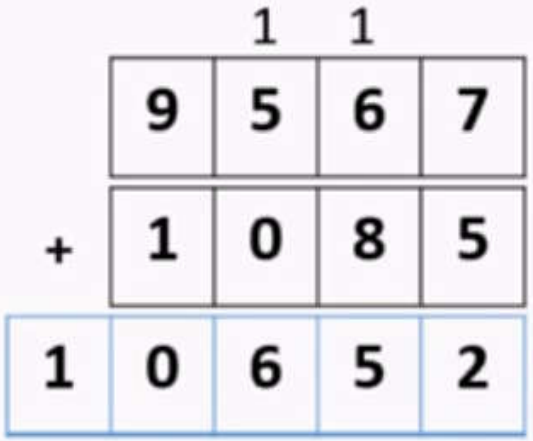
Cryptarithmetic Problem



C4	C3	C2	C1	
	S	E	N	D
+	M	O	R	E
M	O	N	E	Y

Character	Code
S	
E	
N	
D	
M	
O	
R	
Y	

Cryptarithmetic Problem



C4 C3 C2 C1

S E N D

\+ M O R E

M O N E Y

Character	Code
S	9
E	5
N	6
D	7
M	1
O	0
R	8
Y	2

TUTORIAL PROGRAM

Implementation of Constraint Satisfaction Problem for Cryptarithmatic

Real-world CSPs

- Assignment problems
  - e.g., who teaches what class
- Timetabling problems
  - e.g., which class is offered when and where?
- Transportation scheduling
- Factory scheduling

Standard search formulation

Let's start with the straightforward, dumb approach, then fix it

States are defined by the values assigned so far

- Initial state: the empty assignment, $\{\}$
- Successor function: assign a value to an unassigned variable that does not conflict with current assignment.
  $\Rightarrow$ fail if no legal assignments (not fixable!)
- Goal test: the current assignment is complete

1) This is the same for all CSPs!
2) Every solution appears at depth n with n variables
   $\Rightarrow$ use depth-first search
3) Path is irrelevant, so can also use complete-state formulation
4) $b = (n - \ell)d$ at depth ℓ, hence $n!d^n$ leaves!!!!

Solving CSPs
Search:Depth Limited

- CSP with n variables with domain size d
- Branching factor at top = nd
- At next level: $(n-1)d$
- In the end $n!d^n$ leaves. But only d^n possible complete assignments

BACKTRACKING SEARCH

Variable assignments are commutative, i.e.,

$[WA = red \text{ then } NT = green]$ same as $[NT = green \text{ then } WA = red]$

Only need to consider assignments to a single variable at each node

$\Rightarrow \quad b = d$ and there are d^n leaves

Depth-first search for CSPs with single-variable assignments is called backtracking search

Backtracking search is the basic uninformed algorithm for CSPs

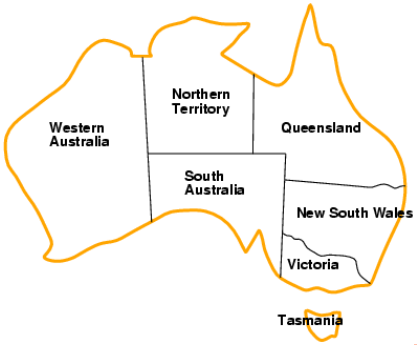
Can solve n-queens for $n \approx 25$

BACKTRACKING SEARCH

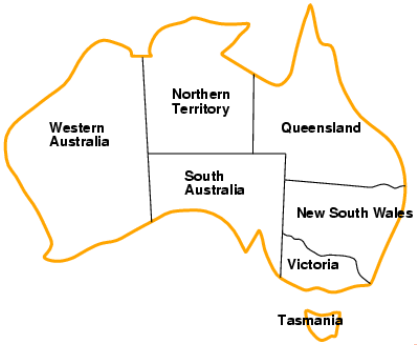
```
function BACKTRACKING-SEARCH(csp) returns solution/failure
    return RECURSIVE-BACKTRACKING({ }, csp)

function RECURSIVE-BACKTRACKING(assignment, csp) returns soln/failure
    if assignment is complete then return assignment
    var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
    for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
        if value is consistent with assignment given CONSTRAINTS[csp] then
            add {var = value} to assignment
            result ← RECURSIVE-BACKTRACKING(assignment, csp)
            if result ≠ failure then return result
            remove {var = value} from assignment
    return failure
```

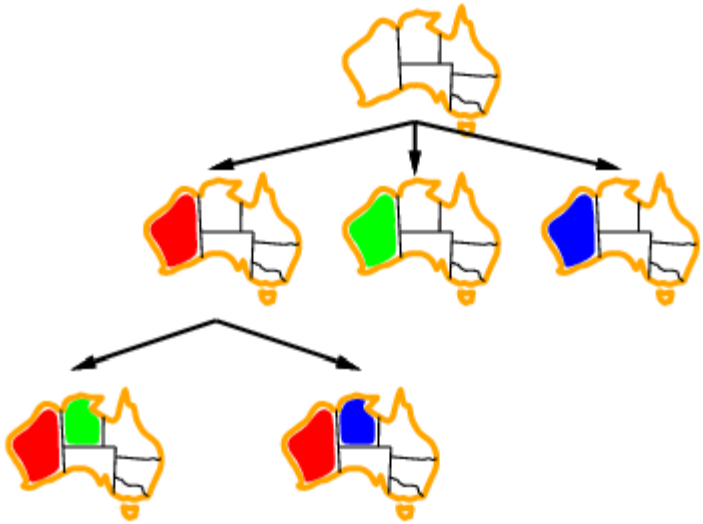
Backtracking example

Backtracking example

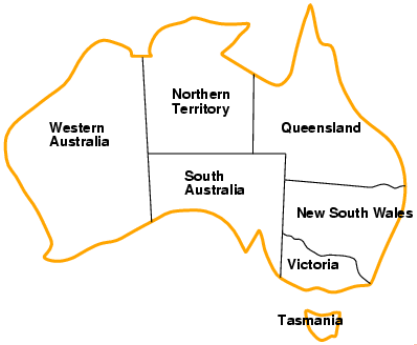


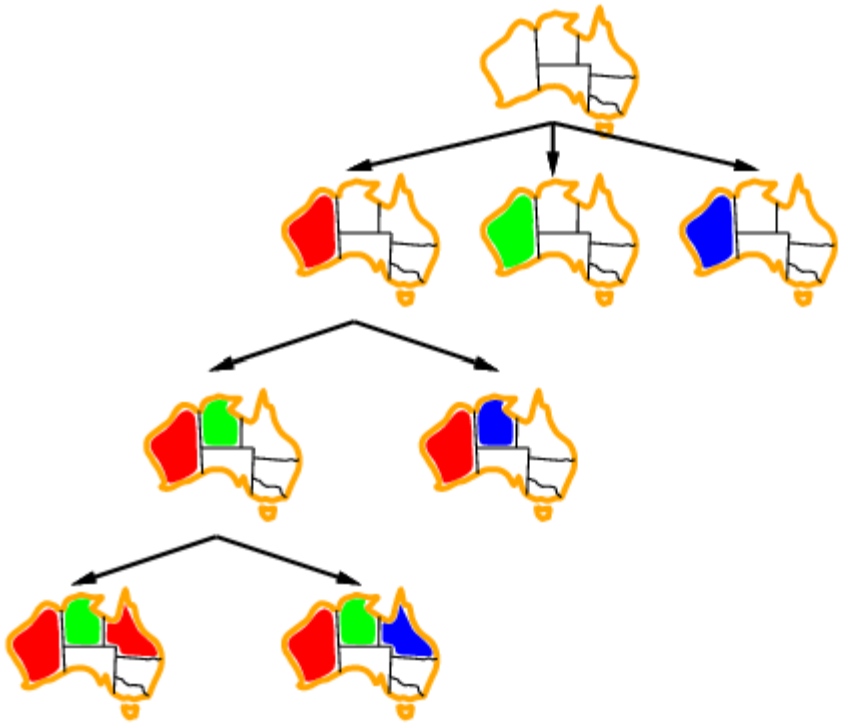
Backtracking example





Backtracking example





IMPROVING BACKTRACKING EFFICIENCY

- General-purpose methods can give huge gains in speed:
  - Which variable should be assigned next?
  - In what order should its values be tried?
  - Can we detect inevitable failure early?

VARIABLE AND VALUE ORDERING

- The backtracking algorithm contains the line
  - var←SELECT-UNASSIGNED-VARIABLE(VARIABLE[csp],*assignment,csp)*
- The idea of choosing variable with fewest legal values is called the **Minimum remaining value heuristic.**
- It also has been called the most constrained variable or fail first heuristic.
- Because it picks a variable that is most likely to cause a failure soon, thereby pruning the search tree.

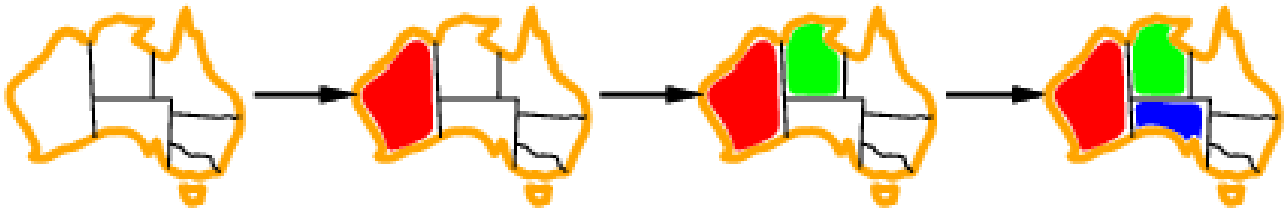
LEAST CONSTRAINING VALUE

- Once a variable has been selected the algorithm must decide on the order in which to examine its values.
- For this Least-constraining-value heuristic can be effect
- Given a variable, choose the least constraining value: the one that rules out the fewest values in the remaining variables

Most constrained variable

- Most constrained variable:

  choose the variable with the fewest legal values

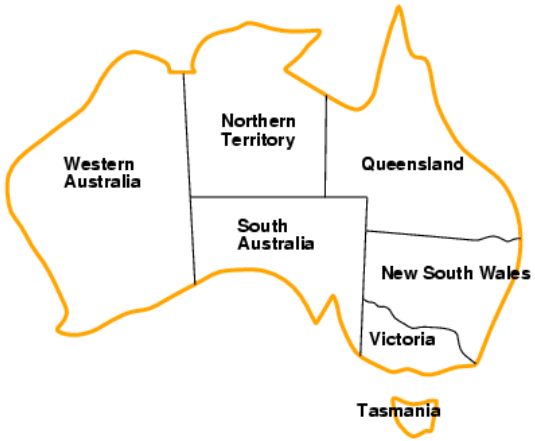


- minimum remaining values (MRV) heuristic

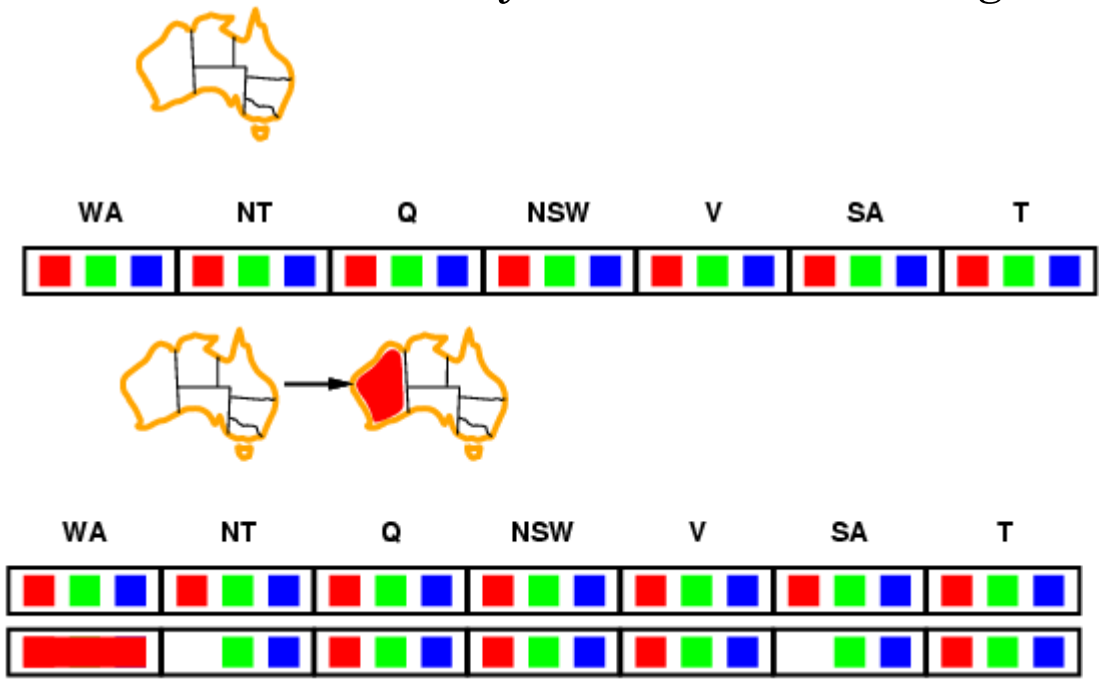
FORWARD CHECKING

- Idea:
  - One way to make better use of constraints during search is called forward checking
  - Whenever a variable X is assigned the forward checking process looks at each unassigned variable Y that is connected to X by a constraint and deletes from Y's domain any value that is inconsistent with value chosen for X
  - Keep track of remaining legal values for unassigned variables
  - Terminate search when any variable has no legal values

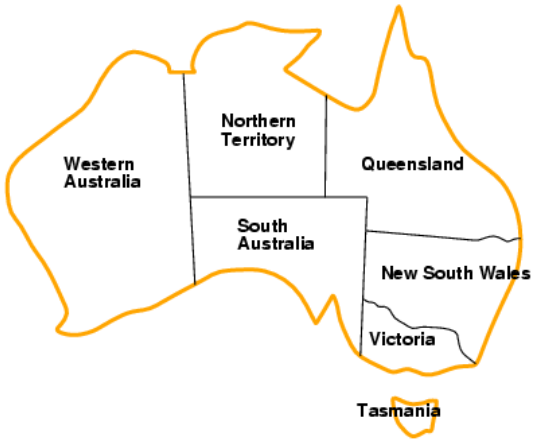
FORWARD CHECKING



- Idea:
  - Keep track of remaining legal values for unassigned variables
  - Terminate search when any variable has no legal values

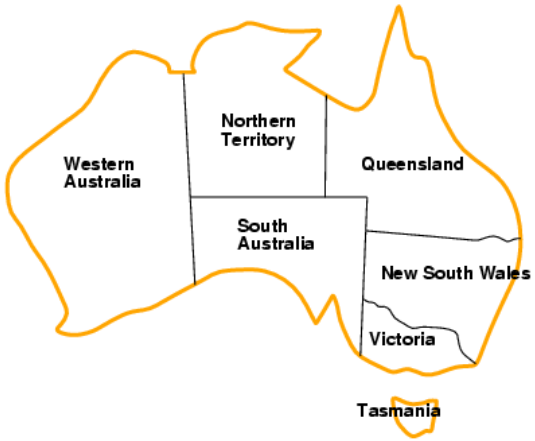


FORWARD CHECKING

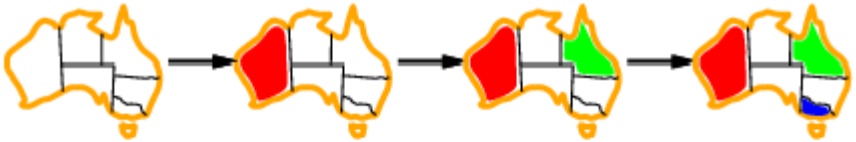


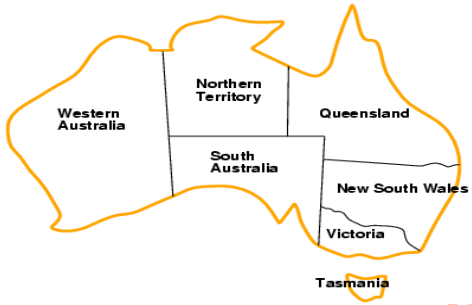
- Idea:
  - Keep track of remaining legal values for unassigned variables
  - Terminate search when any variable has no legal values

FORWARD CHECKING



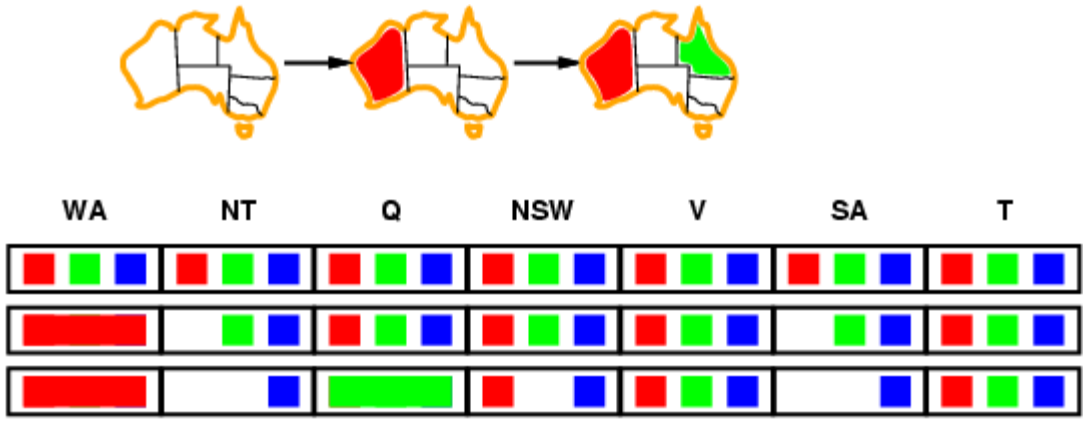
- Idea:
  - Keep track of remaining legal values for unassigned variables
  - Terminate search when any variable has no legal values





CONSTRAINT PROPAGATION

- Forward checking propagates information from assigned to unassigned variables, but doesn't provide early detection for all failures:
- when WA is *red* and Q is *green,* both NT and SA are forced to be blue. But they are ,adjacent and so cannot have the same value. Forward checking does not detect this as an inconsistency, because it does not look far enough ahead.

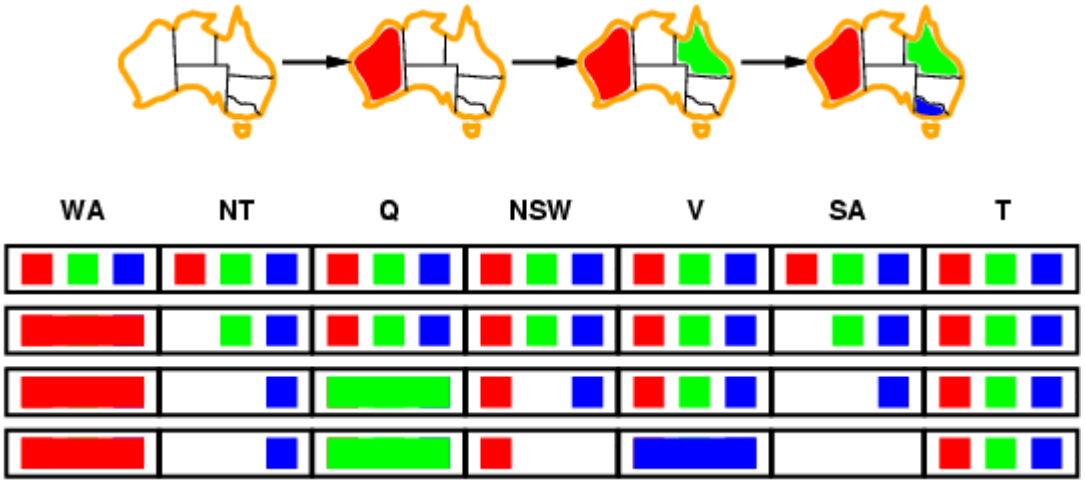


CONSTRAINT PROPAGATION

- **Constraint propagation** is the general term for propagating the implications of a constraint on one variable onto other variables;
- In previous example we need to propagate from
- WA and Q onto NT and SA, (as was done by forward checking) and then onto the constraint between NT and SA to detect the inconsistency.
- We want to do this fast: it is no good reducing the amount of search if we spend more time propagating constraints than we would have spent doing a simple search

ARC CONSISTENCY

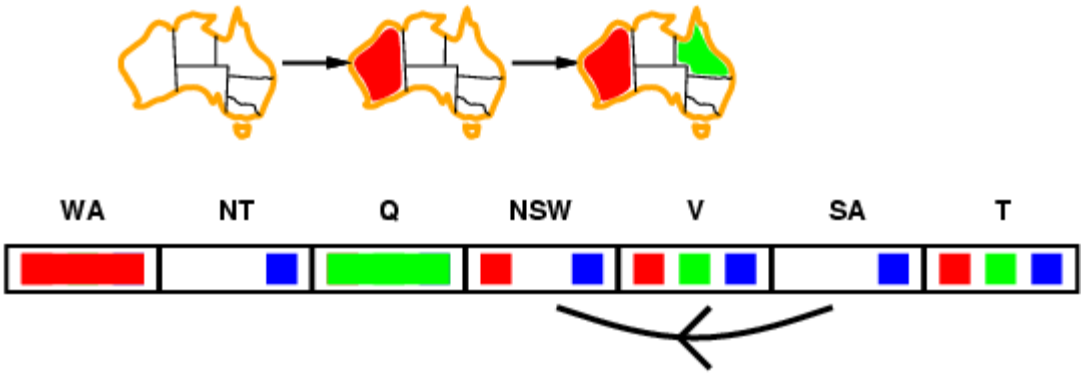
- The idea of **arc consistency** provides a fast method of constraint propagation that is substantially stronger than forward checking
- "arc" refers to a *directed* arc in the constraint graph, such as the arc from SA to NSW



ARC CONSISTENCY

- The idea of arc consistency provides a fast method of constraint propagation that is stronger than forward checking
- Simplest form of propagation makes each arc consistent
- $X \rightarrow Y$ is consistent iff

  for every value x of X there is some value y of Y that is consistent with x



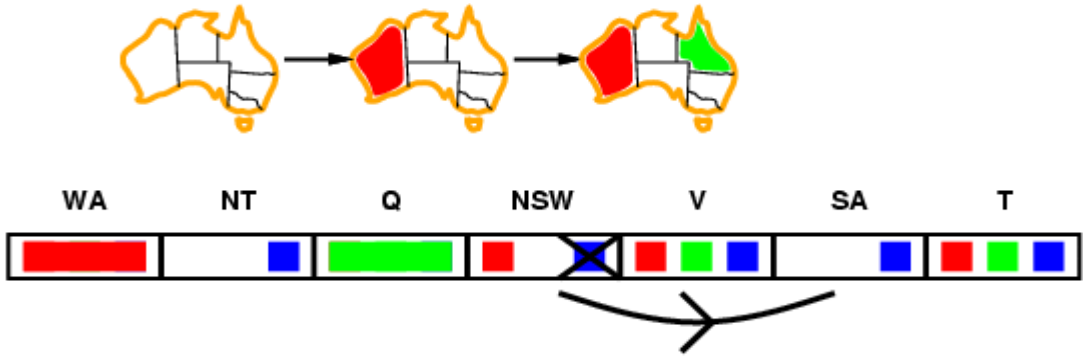
ARC CONSISTENCY

- Simplest form of propagation makes each arc consistent
- $X \rightarrow Y$ is consistent iff

  for every value x of X there is some allowed y

  Arc from SA to NSW is consistent but vice versa is not true

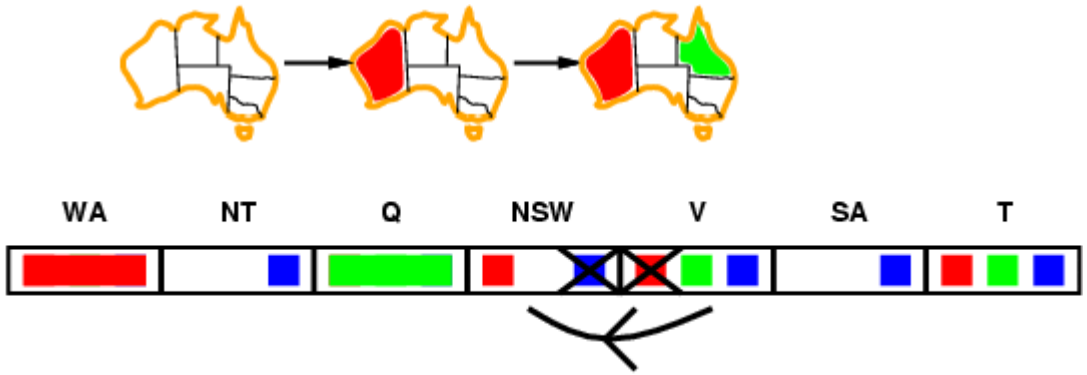
  The arc can be made consistent by deleting the value of blue from the domain of NSW



ARC CONSISTENCY

- Simplest form of propagation makes each arc consistent
- $X \rightarrow Y$ is consistent iff

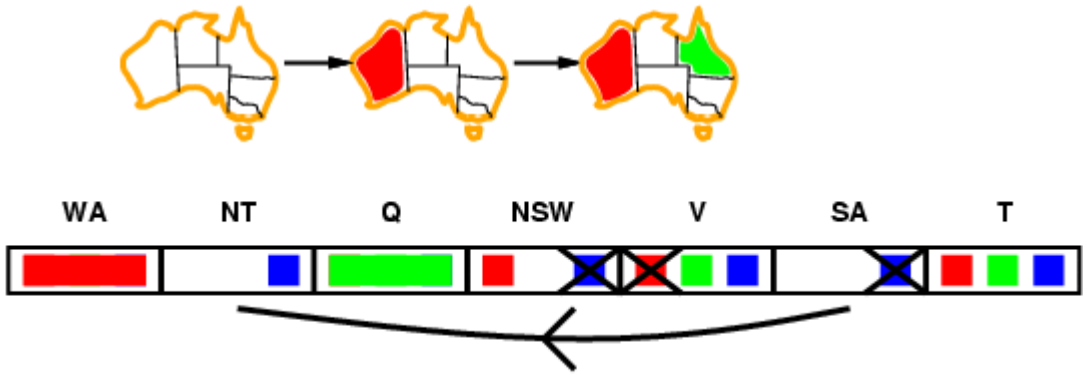
  for every



- If X loses a value, neighbors of X need to be rechecked

ARC CONSISTENCY

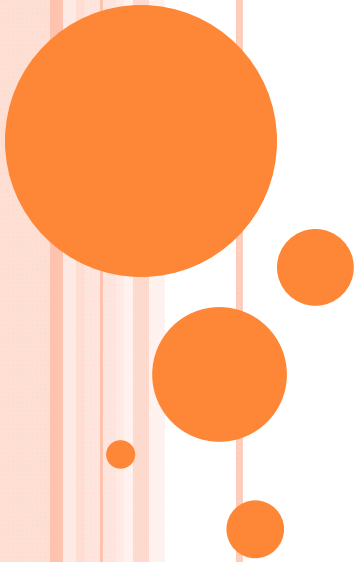
- Simplest form of propagation makes each arc consistent
- $X \rightarrow Y$ is consistent iff
  for every value x of X there is some allowed y



- If X loses a value, neighbors of X need to be rechecked
- Arc consistency detects failure earlier than forward checking
- Can be run as a preprocessor or after each assignment

AI Planning

In which we see how an agent can take advantage of the structure of a problem to construct complex plans of action



PLANNING

- The methods which focus on ways of decomposing the original problem into appropriate subparts and on ways of recording and handling interactions among the subparts as they are detected during the problem-solving process are often called as planning.
- **Planning refers to the process of computing several steps of a problem-solving procedure before executing any of them.**

Search vs. Planning

- Planning Systems do the following:
  - Open up action and goal representation to allow selection
  - Divide-and-conquer by sub-goaling
  - Relax requirement for sequential construction of solutions

Planning Languages

- Languages must represent..
  - States
  - Goals
  - Actions
- Languages must be
  - Expressive for ease of representation
  - Flexible for manipulation by algorithms

State Representation

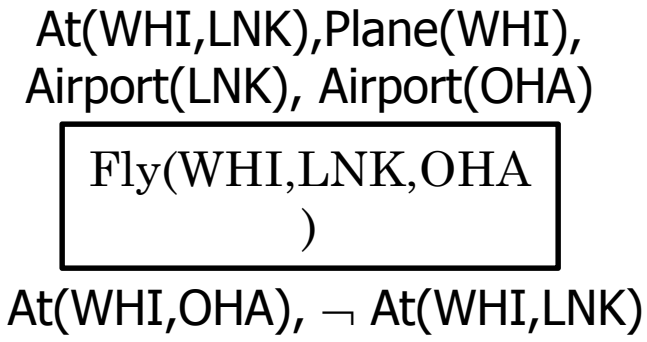
- **A state is representation of the facts**
- A state is represented with a conjunction of positive literals
- Using
  - Logical Propositions: *Poor* $\wedge$ *Unknown*
  - FOL literals: *At(Plane1,OMA)* $\wedge$ *At(Plan2,JFK)*
- Closed World Assumption
  - What is not stated are assumed false

GOAL REPRESENTATION

- **Goal is a <u>partially</u> specified state**, represented as a conjunction of positive ground literals
- A proposition satisfies a goal if it contains all the objects/atoms of the goal and possibly others..
  - Example: Rich $\wedge$ Famous $\wedge$ Miserable satisfies the goal Rich $\wedge$ Famous

Action Representation

- Action Schema
  - Action name
  - Preconditions
  - Effects
- Example



Action(Fly(p,from,to),

PRECOND: At(p,from) ∧ Plane(p) ∧ Airport(from) ∧ Airport(to)

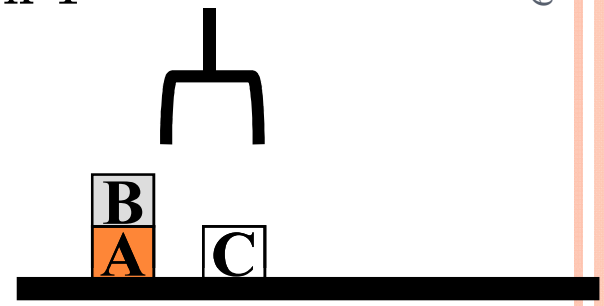
EFFECT: ¬At(p,from) ∧ At(p,to))

- Sometimes, Effects are split into ADD list and DELETE list

BLOCKS WORLD OPERATORS

- Here are the classic basic operations for the blocks world:
  - stack(X,Y): put block X on block Y
  - unstack(X,Y): remove block X from block Y
  - pickup(X): pickup block X
  - putdown(X): put block X on the table
- Each will be represented by
  - a list of preconditions
  - a list of new facts to be added (add-effects)
  - a list of facts to be removed (delete-effects)
  - optionally, a set of (simple) variable constraints
- Predicates:

  On(x , y), OnTable(x), Holding(x), ARMEmpty, Clear(x), OnTable(x).



BLOCK WORLD PROBLEM

- Example

Languages for Planning Problems

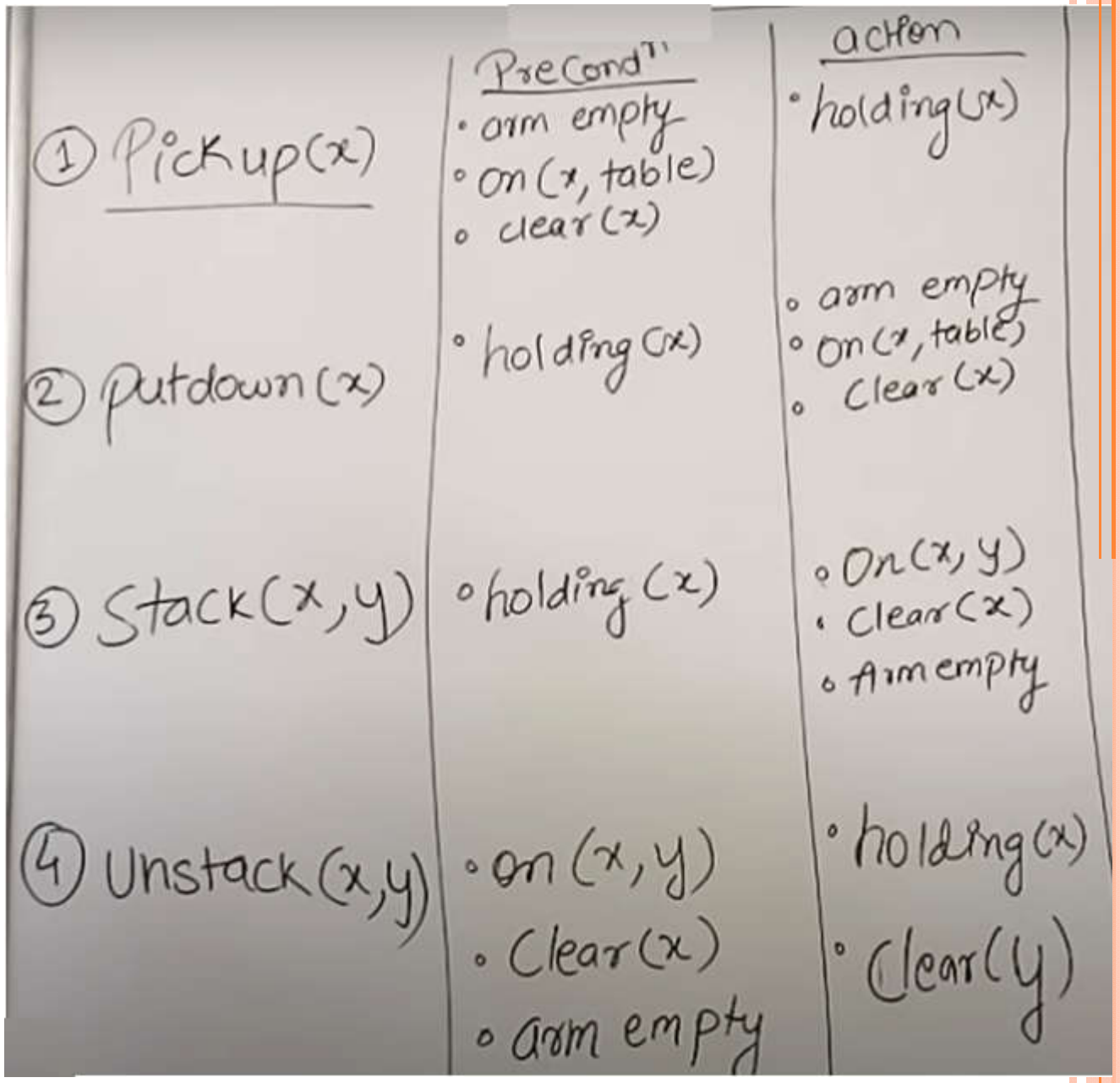
- STRIPS
  - Stanford Research Institute Problem Solver
  - Historically important
- ADL
  - Action Description Languages
- PDDL
  - Planning Domain Definition Language
  - Revised & enhanced for the needs of the International Planning Competition
  - Currently version 3.1

GOAL STACK PLANNING

- One of the earliest techniques is planning using goal stack.
- Problem solver uses single stack that contains sub goals and operators both sub goals are solved linearly and then finally the conjoined sub goal is solved.
- Plans generated by this method will contain complete sequence of operations for solving one goal followed by complete sequence of operations for the next etc.

RULES

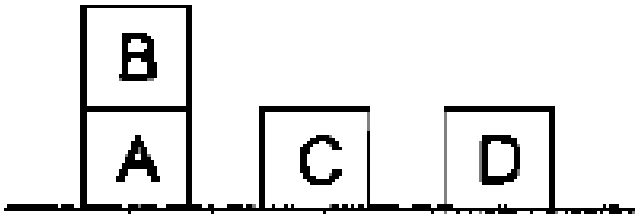
Goal Stack Planning



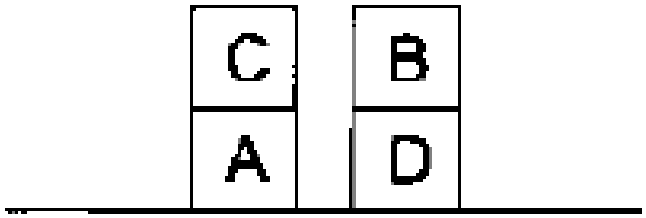
GOAL STACK PLANNING (STEPS)

- Find an operator that satisfies sub goal G1 (makes it true) and replace G1 by the operator.
- If more than one operator satisfies the sub goal then apply some heuristic to choose one.
- In order to execute the top most operation, its preconditions are added onto the stack.
- Once preconditions of an operator are satisfied, then we are guaranteed that operator can be applied to produce a new state.
- New state is obtained by using ADD and DELETE lists of an operator to the existing database.
- Problem solver keeps tract of operators applied.
- This process is continued till the goal stack is empty and problem solver returns the plan of the problem

Goal Stack Planning



start: ON(B, A) ∧
ONTABLE(A) ∧
ONTABLE(C) ∧
ONTABLE(D) ∧
ARMEMPTY



goal: ON(C, A) ∧
ON(B, D) ∧
ONTABLE(A) ∧
ONTABLE(D)

Goal Stack Planning

Example

Planning with State-Space Search

- The most straightforward approach of planning algorithm, is state-space search
  - Forward state-space search (Progression)
  - Backward state-space search (Regression)
- The descriptions of actions in a planning problem, and specify both preconditions and effects
- It is possible to search in both direction: either forward from the initial state or backward from the goal
- We can also use the explicit action and goal representations, to derive effective heuristics automatically.

Problem Formulation for Progression

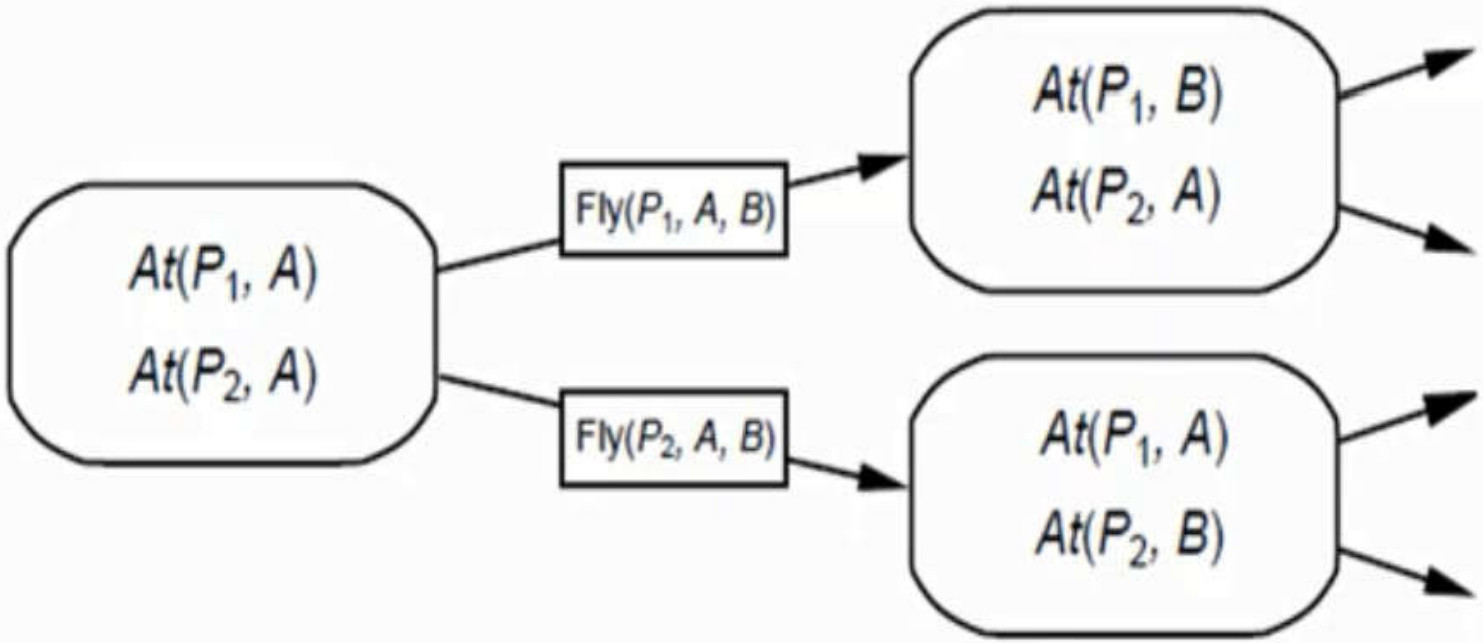
- Initial state:
  - Initial state of the planning problem
- Actions:
  - Applicable to the current state.
  - First actions' preconditions are satisfied, Successor states are generated
  - Add positive literals to add list and negative literals to delete list.
- Goal test:
  - Whether the state satisfies the goal of the planning
- Step cost:
  - Each action is 1 (assumed)

Progression

- From initial state, search forward by selecting operators whose preconditions can be unified with literals in the state
- New state includes positive literals of effect; the negated literals of effect are deleted
- Search forward until goal unifies with resulting state
- This is forward state-space search using STRIPS operators

Forward (progression) state-space search

- Starting from the initial state and using the problem's actions to search forward for the goal state.



FSSS Algorithm

- (1) compute whether or not a state is a goal state,
- (2) find the set of all actions that are applicable to a state, and
- (3) compute a successor state, that is the result of applying an action to a state
- Algorithm takes as input the statement P = (O, s0, g) of a planning problem P. (O contains a list of actions)
- If P is solvable, then Forward-search(O, s0, g) returns a solution plan; otherwise it returns failure.

Problem Formulation for Regression

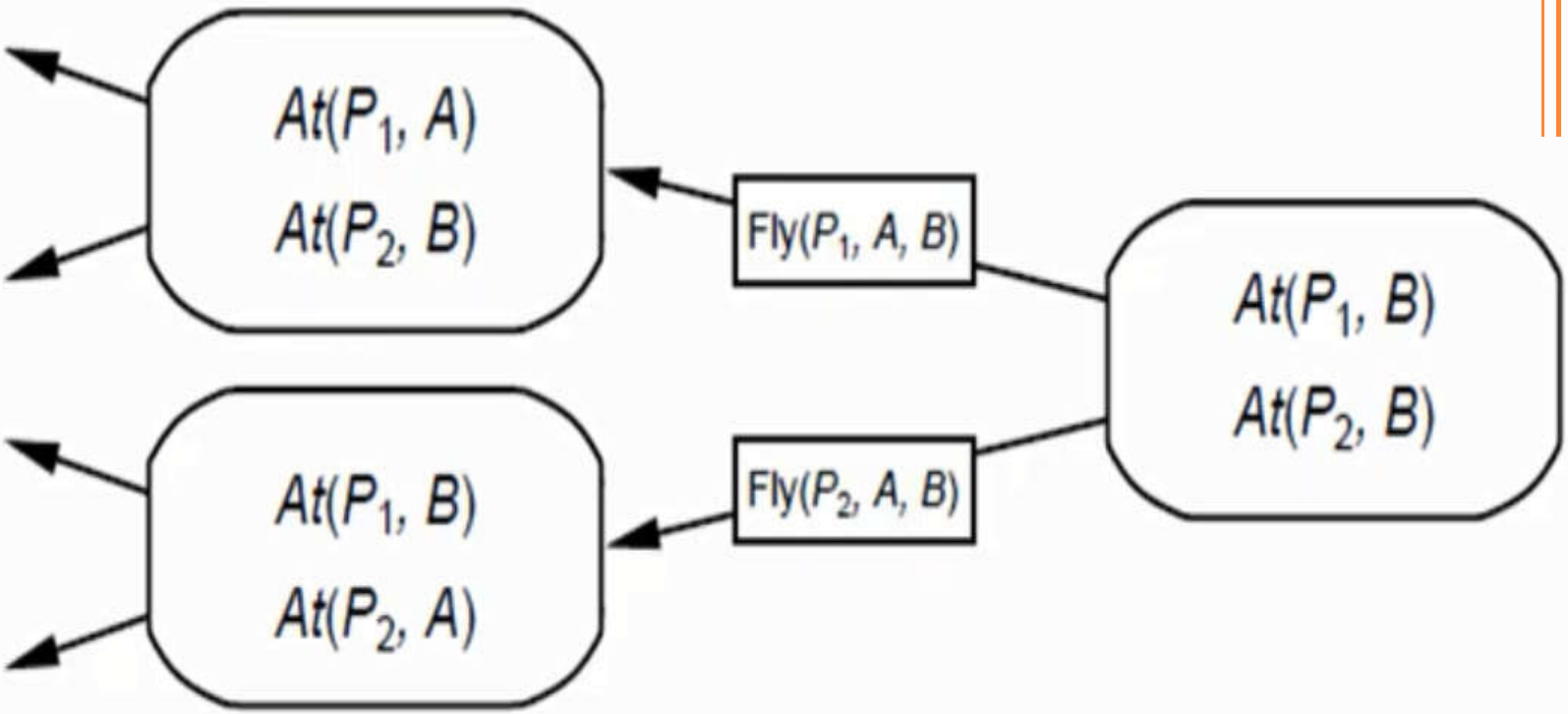
- Initial state:
  - Initial state of the planning problem
- Actions:
  - Applicable to the current state.
  - First actions' preconditions are satisfied, Successor states are generated
  - Add positive literals to add list and negative literals to delete list.
- Goal test:
  - Whether the state satisfies the goal of the planning
- Step cost:
  - Each action is 1 (assumed)

Regression

- The goal state must unify with at least one of the positive literals in the operator's effect
- Its preconditions must hold in the previous situation, and these become subgoals which might be satisfied by the initial conditions
- Perform backward chaining from goal
- Again, this is just state-space search using STRIPS operators

Backward (regression) state-space search:

- Backward state search starting from the goal state(s)
- Using the inverse of the actions to search backward for the initial state.



BSSS Algorithm

- Start at the goal,
- Test the goal is initial state, otherwise
- Apply inverses of the planning operators to produce subgoals,
- The algorithm will stop, if we produce a set of subgoals that satisfies the initial state.

Thank you

Prepared by: Dr. Nilima Kulkarni, MIT SOE.