

# Path Problems

solution is a sequence of operators  
transforming current into desired state

## Problem Space

S: set of possible states

G: goal space  
subset of S representing  
desired situation

I: initial state  
element of S representing  
current situation

O: operators  
functions mapping  $O: S \rightarrow S$   
specified by sets of  
preconditions and effects

## Water-jug Problem

Given three jugs with capacities A, B, and C  
an unlimited source and sink for water  
an empty vat with capacity 100  
a GOAL amount of water to be in

find a sequence of  
pour-ins and bail-outs of the jugs  
realizing a GOAL amount in the vat.  
such that no jug is used  
more than three times.

## Problem Space

S:  $\langle \text{uses}(A), \text{uses}(B), \text{uses}(C), \text{amount} \rangle$   
 $0 \leq \text{uses}(A), \text{uses}(B), \text{uses}(C) \leq 3$   
 $0 \leq \text{amount} \leq 100$

G:  $\langle ?, ?, ?, \text{GOAL} \rangle$

I:  $\langle 0, 0, 0, 0 \rangle$

state represented as vector of attribute values

## Water-jug Problem Space (continued)

O:

pour-in(?jug):

preconditions:

uses(?jug) < 3

amount + capacity(?jug) ≤ 100

effects:

add 1 to uses(?jug)

add capacity(?jug) to amount

bail-out(?jug):

preconditions:

uses(?jug) < 3

amount - capacity(?jug) ≥ 0

effects:

add 1 to uses(?jug)

subtract capacity(?jug) from amount

amount in jug is part of state

# General Search Method

(forward-directed: from I to G)

Given I, G, and O           ;; S implicit

1. set {Sobtained} to {I};  
   set Scurrent to I.
2. Until [Scurrent is in G or Scurrent is nil]  
   **select-operators** {Ocurrent} from O,  
   **expand Scurrent**, using  
       {Ocurrent} to generate {Snew},  
   **update {Sobtained}** according to {Snew},  
   **select Scurrent** from {Sobtained};
3. If [Scurrent is nil]  
   then report-failure  
   else retrieve-solution

# General Search Method

## procedures

### expand $S_{current}$ :

apply selected operators to  $S_{current}$   
to generate set of new states  $\{S_{new}\}$

for each operator  $o$  with satisfied preconditions,  
for each way the preconditions can be satisfied,  
generate new state  $s$  according to effects of  $o$ ;  
add  $s$  to plan following  $S_{current}$

search in space of "nodes"  
each node includes

- (i) state representation
- (ii) plan so far -or- pointer to prior node  
with prior state and operator applied
- (iii) other search control information  
(depth, cost so far, heuristic value)

# General Search Method

## procedures

**update** {Sobtained}:

options:

simply add new elements,  
allowing duplicate states  
recognize and eliminate  
some or all duplicate states

maintain nodes as {Open} and {Closed} subsets

search strategies

limit size of {Sobtained},  
keeping only k best  
k = 1 memoryless, k > 1 beam search

treat as stack or queue  
depth-first or breadth-first search

order according to heuristic information

hash or ancestor search to eliminate duplicates

# General Search Method

## procedures

**select** Scurrent:

interacts with means of maintaining {Sobtained}

only consider states in {Open};  
once expanded, put on {Closed}

take first state in an ordered {Sobtained}

# Basic Search Strategies

## uninformed search

breadth-first

depth-first

iterative deepening

compare

completeness

solution optimality

time complexity

space complexity

# Basic Search Methods

## breadth-first

treat {Sobtained} as a queue

complete method  
finds optimal depth solution

treat {Sobtained} as a priority queue  
based on path cost (so far)

minimum cost solution  
Dijkstra's algorithm for shortest path in a graph

## depth-first

treat {Sobtained} as a stack

potentially incomplete method  
potentially non-optimal solution

# Basic Search Methods

## complexity comparison

worst case

assuming depth **d** of solution

assuming branching factor of **b**

## breadth-first search

time complexity  $O(b^d)$

space complexity  $O(b^d)$

always finds shallowest solution

## depth-first search (with depth-limit at d)

time complexity  $O(b^d)$

space complexity  $O(bd)$

wins on space measure,

with same order time complexity

when limit depth set to solution depth

# Iterative Deepening

combine depth-first and breadth-first advantages

method:

search by depth-first method  
to successive depths 1, 2, 3, ....  
until find solution

analysis:

complete method  
optimal depth solution

time complexity  
ratio of iterative deepening  
to breadth-first is  
 $\sim\sim b/(b-1)$

space complexity  
 $db$  ;;; linear in  $b$  and  $d$

# General Search Method

**select**  $S_{current}$ :

heuristic opportunity,  
reflecting structure of state space

## **informed (heuristic) search strategies**

based on an evaluation function  $f: S \rightarrow R$

$$f(s) = g(s) + h(s), \text{ for a state } s$$

$g(s)$  (estimated) cost to reach  $s$  from  $I$

$h(s)$  estimated cost to reach  $G$  from  $s$

$h$  is heuristic function

select state with minimum value of  $f$

$\Rightarrow$

maintain set  $\{S_{obtained}\}$  as priority queue,  
ordered by increasing  $f$  value

$f$  reflects knowledge of the structure of state space

# Heuristic Search Methods

"best-first" methods

$h(s) = 0$  ===== Dijkstra's Algorithm  
only cost

$g(s) = 0$  ===== greedy algorithm  
only heuristic value considered

$h(s) \leq H(s)$  ===== A\* Search,  
where  $H(s)$  is "actual"  
cost from  $s$  to  $G$

if  $h$  never overestimates total cost;  
such an  $h$  is said to be  
**admissible**

if  $h_1(s) \leq h_2(s) \leq H(s)$  =====  
 $h_2$  is **more informed** than  $h_1$

# Heuristic Search

## Important Results

A\* Search

{using evaluation  $f = g+h$ ,  
where heuristic  $h$  is admissible}

A\* is also said to be admissible

if we select the state with lowest  $f$  then:  
will find a minimum-cost path, if one exists;  
(complete and optimal)

if  $h_1$  and  $h_2$  are both admissible  
with  $h_1 \geq h_2$  ( $h_1$  is more informed)  
then a less or equal number of states  
will be developed  
when using  $h_1$  during search

# Heuristic Search

## Examples

### A\*

admissible heuristic functions  
more informed heuristics

### 8-Puzzle

1	5	7		1	2	3
3	6	2	-->	4	5	6
4	8			7	8	

### marker problem

white white white <blank> black black black

operators:

slide to neighbor blank

hop neighbor to blank

goal state: reverse initial pattern

# Heuristic Search

A\*

heuristic search lowers "effective branching factor"  
not depth of solution

problem is the size of {Sobtained}  
i.e., the space requirement

## Iterative Deepening A\*

Can we take the idea of depth-first  
iterative deepening and apply  
it in heuristic search to lessen the  
memory burden of best-first search?

# Iterative Deepening A\*

search with score of  $f$  of root  
until forced to expand node with  
greater  $f$  value

search with next highest  $f$  score  
(seen on fringe of past search)  
until forced to go higher..

## problems

only maintains its best first character  
when have a monotone  $f$  function  
increasing values:  $f(\text{child}) \geq f(\text{parent})$

increasing  $f$  limit may only allow a few  
more states to be searched during an iteration  
implies time order squared the time of basic A\*

## solutions

increase  $f$  limit by some fixed amount  $\epsilon$   
when find first solution know it is within  $\epsilon$   
of optimal cost solution  
can complete search with that  $\epsilon$   
to guarantee minimum cost solution

# Planning and Means-Ends Analysis

## Operator Selection

### Means-Ends Analysis

only select those operators  $\{O_{current}\}$   
that are expected to reduce the difference  
between  $S_{current}$  and  $G$

### Example

In water jug problem:  
only pour-in when above goal  
only bail-out when below

<b>G</b>	<b>A</b>	<b>B</b>	<b>C</b>
25	10	3	8
17	8	6	7

# GPS and Means-Ends Analysis

when problem solving, always have one of  
three goal types with associated methods

**transform** state1 into state2

**reduce** difference between state1 and state2

**apply** operator op to state1

transform(s1, s2) :

diff(s1, s2)? d -> reduce(s1, d) s3 -> transform(s3, s2)  
|  
done

reduce(s1, d) :

op(d)? o-> apply(o, s1)  
|  
fail

apply(o, s1) :

diff(s1, pre(o))? d-> reduce(s1, d) s2-> apply(o, s2)  
|  
do(o, s1)

# GPS

## Control

maintains a current goal ---  
executes method associated with type  
of current goal

always tries to reduce difference  
of greatest difficulty first ---  
if a new goal has difference greater  
than "prior" goal, will backup

if no operator available then backup

## "Knowledge" Representation

table of connections  
represents relevance of operators  
to reducing differences

difference ordering  
represents relative difficulty of  
reducing differences

# Means-Ends Analysis

## STRIPS Planner

important step in implementing  
these ideas in a first planning system

introduces a proposition-based state representation

an algorithm (ala GPS) for solving problems  
by means-ends analysis

## State Representation

states as sets of propositions of form

(relation-name arg1 arg2 ....)  
zero or more arguments, all are constants..

## state example

### Blocks World

(clear A)	A
(on A B)	B
(on B C)	C
(on C Table)	_____

# STRIPS

## Operator Representation

(operator-name arg1..)

arguments are variables

**preconditions** (propositions in current state)

**adds** (propositions to add/positive)

**deletes** (propositions to remove/negate)

propositions can contain arguments variables

## operator example

### Blocks World

moveBlocktoBlock(?b1 ?b2 ?b3)

pre: ( (on ?b1 ?b2) (clear ?b1) (clear ?b3) )

add: ( (on ?b1 ?b3) (clear ?b2) )

del: ( (on ?b1 ?b2) (clear ? b3) )

movetoTable(?b1 ?b2) // Table always clear

movetoBlock(?b1, ?b2) // from Table

# Means-Ends Analysis

## STRIPS search

1. push G onto goal stack, followed by  
    its individual propositions in some order on top;  
    set S<sub>current</sub> to I;
  
2. Until goal stack empty  
    if top is operator o,  
        then pop stack and apply o,  
            generating new S<sub>current</sub>  
            adding o to solution  
            or put preconditions on stack;  
    if top is a proposition p  
        if p in S<sub>current</sub>  
            then pop goal stack;  
        else  
            select operator o that adds p,  
            and push o, followed by  
            its preconditions,  
            on top of goal stack;  
    if top is a set of propositions,  
        if all in G, pop stack  
        else push propositions on stack
  
3. Report solution  
    simplified.... no backtracking...

# Means-ends Analysis

## Strips Search

Consider this "laundromat problem":

I:  
((AT washer) (INHAND bill) (WASHER off))

G:  
((WASHER on))

O: (((START-WASHER)  
    (P: (AT washer) (INHAND quarter)  
        (WASHER off))  
    (A: (WASHER on))  
    (D: (WASHER off) (INHAND quarter))))

((GET-CHANGE)  
    (P: (AT changer) (INHAND bill))  
    (A: (INHAND quarter))  
    (D: (INHAND bill)))

((WALK x y)  
    (P: (AT x))  
    (A: (AT y))  
    (D: (AT x))))

# STRIPS search

## laundromat problem

CS: ((AT washer) (INHAND bill) (WASHER off))

Stack: ((WASHER on))

Stack: ((AT washer) (INHAND quarter) (WASHER off)  
(do START-WASHER) (WASHER on))

Stack: ((INHAND quarter) (WASHER off)  
(do START-WASHER) (WASHER on))

Stack: ((AT changer) (INHAND bill) (do GET-CHANGE)  
(INHAND quarter) (WASHER off)  
(do START-WASHER) (WASHER on))

Stack: ((AT washer) (do (WALK washer changer))  
(AT changer) (INHAND bill) (do GET-CHANGE)  
(INHAND quarter) (WASHER off)  
(do START-WASHER) (WASHER on))

Stack: ((do (WALK washer changer))  
(AT changer) (INHAND bill) (do GET-CHANGE)  
(INHAND quarter) (WASHER off)  
(do START-WASHER) (WASHER on))

# STRIPS search

## laundromat problem (continued)

CS: ((AT washer) (INHAND bill) (WASHER off))

Stack: ((do (WALK washer changer))  
(AT changer) (INHAND bill) (do GET-CHANGE)  
(INHAND quarter) (WASHER off)  
(do START-WASHER) (WASHER on))

..... check and can do (WALK washer changer) .....

CS: ((AT changer) (INHAND bill) (WASHER off))

Stack: ((AT changer) (INHAND bill) (do GET-CHANGE)  
(INHAND quarter) (WASHER off)  
(do START-WASHER) (WASHER on))

Stack: ((INHAND bill) (do GET-CHANGE)  
(INHAND quarter) (WASHER off)  
(do START-WASHER) (WASHER on))

Stack: ((do GET-CHANGE)  
(INHAND quarter) (WASHER off)  
(do START-WASHER) (WASHER on))

.... check and find can do GET-CHANGE

CS: ((AT changer) (INHAND quarter) (WASHER off))

etc... to be finished

# Blocks World

## Sussman Anomaly

Goal (on A B)  
(on B C)

A	
B	
C	

---

Initial (on C A)  
(on A TABLE)  
(on B TABLE)  
(clear C) (clear B)

C	
A	B

---

If choose goal (on B C) as difference

get to

B	
C	
A	

---

If choose goal (on A B) first

get to

A	
B	C

---

# Non-Linear Planning

develop a partial-order plan

only ordered by linking additions  
to later preconditions

must maintain (protect) that precondition over  
the interval between being added and used

initial state adds conditions  
goal state uses conditions (like preconditions)

(on C A)		
(on B TABLE)		(on A B)
	[puton C TABLE]	[puton A B]
(clear B)		
	[puton B C]	(on B C)
(clear C)		

# Problem Reduction

Reduce Problem to And-Or Set of Subproblems

notion is that to solve original problem  
must solve one of the or ed  
sets of and ed subproblems

Reduction continues until  
Primitive Problems are reached

Primitive Problem is one  
solvable by known operator or plan

Reduction produces an

and-or tree

and node satisfied  
if all descendants satisfied  
or node satisfied if one satisfied

often gives us a recursive solution to problem