CSC 550: Introduction to Artificial Intelligence

Fall 2008

AI as search

- problem states, state spaces
- uninformed search strategies
  - depth first search
  - depth first search with cycle checking
  - breadth first search
  - breadth first search with cycle checking
  - iterative deepening

AI as search

GOFAI philosophy:

Problem Solving = Knowledge Representation + Search (a.k.a. deduction)

to build a system to solve a problem:

1. define the problem precisely
   i.e., describe initial situation, goals, …
2. analyze the problem
3. isolate and represent task knowledge
4. choose an appropriate problem solving technique

we'll look at a simple representation (states) first & focus on search

more advanced representation techniques will be explored later

Example: airline connections

suppose you are planning a trip from Omaha to Los Angeles

initial situation:	located in Omaha	
goal:	located in Los Angeles	
possible flights:	Omaha → Chicago	Denver → Los Angeles
	Omaha → Denver	Denver → Omaha
	Chicago → Denver	Los Angeles → Chicago
	Chicago → Los Angeles	Los Angeles → Denver
	Chicago → Omaha	

we could define a special-purpose program to find a path

- would need to start with initial situation (located in Omaha)
- repeatedly find flight/rule that moves from current city to next city
- stop when finally reach destination (located in Los Angeles) or else no more paths

note: this is the same basic algorithm as with the automated deduction example

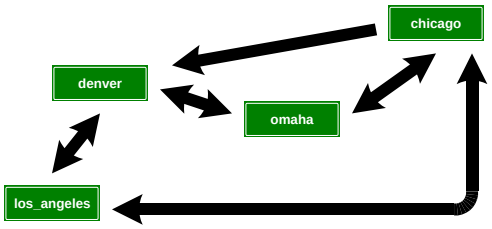
3

State spaces

or, could define a more general solution framework

state : a situation or position

state space : the set of legal/attainable states for a given problem
a state space can be represented as a directed graph (nodes = states)



in general: to solve a problem

- define a state space, then search for a path from start state to a goal state

4

Example: airline state space

define a state by a city name: `Omaha` `LosAngeles`

define state transitions with: `(GET-MOVES state)`

e.g., `(GET-MOVES 'Omaha)` → `(Chicago Denver)`

```
;;; travel.scm      Dave Reed      9/4/08

(define MOVES
  '((Omaha --> Chicago) (Omaha --> Denver)
   (Chicago --> Denver) (Chicago --> LosAngeles) (Chicago --> Omaha)
   (Denver --> LosAngeles) (Denver --> Omaha)
   (LosAngeles --> Chicago) (LosAngeles --> Denver)))


(define (GET-MOVES state)
  (define (get-help movelist)
    (cond ((null? movelist) '())
          ((equal? state (caar movelist))
           (cons (caddar movelist) (get-help (cdr movelist))))
          (else (get-help (cdr movelist)))))
  (get-help MOVES))
```

5

Example: water jug problem

suppose you have two empty water jugs:

- large jug can hold 4 gallons, small jug can hold 3 gallons

starting with empty jugs (and an endless supply of water), want to end up with exactly 2 gallons in the large jug

state?

- start state?
- goal state?

transitions?

6

Example: water jug state space

define a state with: `(largeJugContents smallJugContent)`

define state transitions with: `(GET-MOVES state)`

e.g., `(GET-MOVES '(0 0))` → `((4 0) (0 3))`

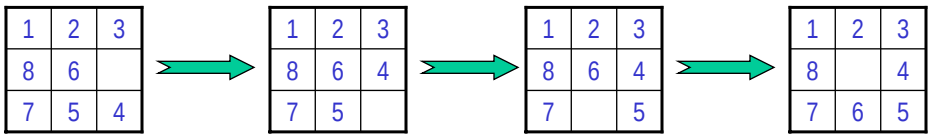
```
;;;  jugs.scm    Dave Reed    9/04/08

(define (GET-MOVES state)

  (define (remove-bad lst)
    (cond ((null? lst) '())
          ((or (equal? (car lst) state)
               (member (car lst) (cdr lst))) (remove-bad (cdr lst)))
          (else (cons (car lst) (remove-bad (cdr lst))))))

  (let ((jug1 (car state)) (jug2 (cadr state)))
    (let ((pour1 (min (- 3 jug2) jug1)) (pour2 (min (- 4 jug1) jug2)))
      (remove-bad (list (list 4 jug2) (list 0 jug2)
                        (list jug1 3) (list jug1 0)
                        (list (- jug1 pour1) (+ jug2 pour1))
                        (list (+ jug1 pour2) (- jug2 pour2)))))))
```

7

Example: 8-tiles puzzle

consider the children's puzzle with 8 sliding tiles in a 3x3 board

- a tile adjacent to the empty space can be shifted into that space
- goal is to arrange the tiles in increasing order around the outside



state?

- start state?
- goal state?

transitions?

define a state with: $(r_1c_1\ r_1c_2\ r_1c_3\ r_2c_1\ r_2c_2\ r_2c_3\ r_3c_1\ r_3c_2\ r_3c_3)$

define state transitions with: `(GET-MOVES state)`

```
e.g., (GET-MOVES '(1 2 3 8 6 space 7 5 4)) →
        ((1 2 space 8 6 3 7 5 4)
         (1 2 3 8 space 6 7 5 4)
         (1 2 3 8 6 4 7 5 space))
```

8

Example: 8-tiles state space

```
;;;  tiles.scm    Dave Reed    9/04/08
;;;  NOTE: this is an UGLY, brute-force implementation

(define (GET-MOVES state)
  (cond ((equal? (list-ref state 0) 'space) (list (swap state 0 1) (swap state 0 3)))
        ((equal? (list-ref state 1) 'space) (list (swap state 1 0) (swap state 1 2)
                                                  (swap state 1 4)))
        ((equal? (list-ref state 2) 'space) (list (swap state 2 1) (swap state 2 5)))
        ((equal? (list-ref state 3) 'space) (list (swap state 3 0) (swap state 3 4)
                                                  (swap state 3 6)))
        ((equal? (list-ref state 4) 'space) (list (swap state 4 1) (swap state 4 3)
                                                  (swap state 4 5) (swap state 4 7)))
        ((equal? (list-ref state 5) 'space) (list (swap state 5 2) (swap state 5 4)
                                                  (swap state 5 8)))
        ((equal? (list-ref state 6) 'space) (list (swap state 6 3) (swap state 6 7)))
        ((equal? (list-ref state 7) 'space) (list (swap state 7 4) (swap state 7 6)
                                                  (swap state 7 8)))
        ((equal? (list-ref state 8) 'space) (list (swap state 8 5) (swap state 8 7)))))


(define (swap lst index1 index2)
  (let ((val1 (list-ref lst index1)) (val2 (list-ref lst index2)))
    (replace (replace lst index1 val2) index2 val1)))

(define (replace lst index item)
  (if (= index 0)
      (cons item (cdr lst))
      (cons (car lst) (replace (cdr lst) (- index 1) item))))
```

9

State space + control

once you formalize a problem by defining a state space:

- you need a methodology for applying actions/transitions to search through the space (from start state to goal state)

  i.e., you need a *control strategy*

control strategies can be: *tentative* or *bold*; *informed* or *uninformed*

- a *bold* strategy picks an action/transition, does it, and commits
- a *tentative* strategy burns no bridges

- an *informed* strategy makes use of problem-specific knowledge (heuristics) to guide the search
- an *uninformed* strategy is blind

10

Uninformed strategies

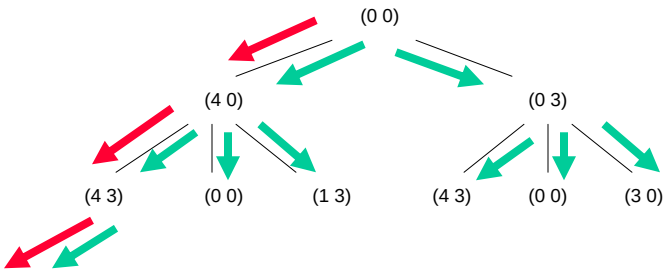
bold + uninformed:

STUPID!

tentative + uninformed:

- depth first search (DFS)
- breadth first search (BFS)

example: water jugs state space (drawn as a tree with duplicate nodes)



11

Depth first search

```
(define (DFS startState goalState)

  (define (extend path moves)
    (cond ((equal? (car path) goalState) path)
          ((null? moves) #f)
          (else (or (extend (cons (car moves) path) (GET-MOVES (car moves)))
                    (extend path (cdr moves))))))

  (extend (list startState) (GET-MOVES startState)))
```

basic idea:

- keep track of the path you are currently searching: `(currState prevState … startState)`
- if the current state is the goal, then SUCCEED
- if there are no moves from the current state, then FAIL
- otherwise, (recursively) try the first move
  if not successful, try successive moves (reminder: OR uses short-circuit eval)

note: this implementation is different from the book's description
not quite as efficient (relies on full recursion instead of destructive assignments)
IMHO, much clearer

12

DFS examples

```
> (DFS 'Omaha 'LosAngeles)
(losangeles denver chicago omaha)

> (DFS 'LosAngeles 'Omaha)
user break
```

first search: path is OK, but not optimal

second search: program hangs

```
> (DFS '(0 0) '(4 0))
((4 0) (0 0))

> (DFS '(0 0) '(2 0))
user break
```

first search: path is OK (although trivial)

second search: program hangs

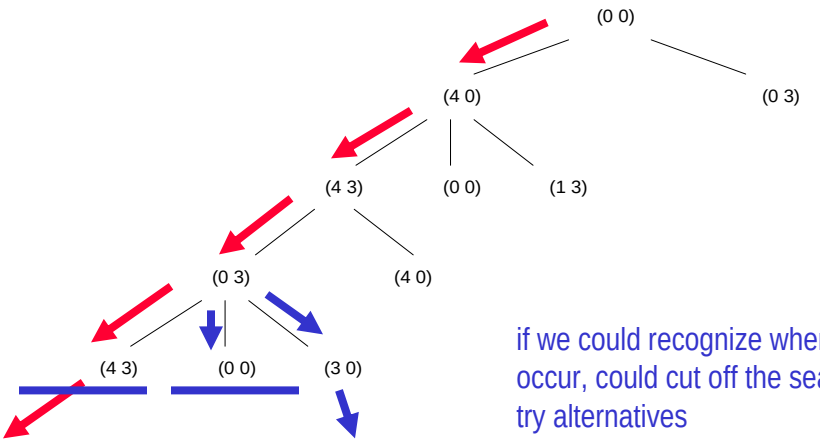
```
> (DFS '(1 2 3 8 6 4 7 space 5) '(1 2 3 8 space 4 7 6 5))
((1 2 3 8 space 4 7 6 5) (1 2 3 8 6 4 7 space 5))

> (DFS '(1 2 3 8 6 space 7 5 4) '(1 2 3 8 space 4 7 6 5))
user break
```

first search: path is OK (although trivial)

second search: program hangs

DFS and cycles

since DFS moves blindly down one path,
looping (cycles) is a SERIOUS problem



if we could recognize when cycles occur, could cut off the search and try alternatives

Depth first search with cycle checking

```
(define (DFS-nocycles startState goalState)

  (define (extend path moves)
    (cond ((equal? (car path) goalState) path)
          ((null? moves) #f)
          (else (or (and (not (member (car moves) path))
                         (extend (cons (car moves) path) (GET-MOVES (car moves))))
                    (extend path (cdr moves))))))

  (extend (list startState) (GET-MOVES startState)))
```

it often pays to test for cycles:

- already have the current path stored, so relatively easy
- before adding next state to path, make sure not already a member

note: again, algorithm described in text is more efficient (can avoid redundant search) but this version is much clearer

15

Examples w/ cycle checking

```
> (DFS-nocycles 'Omaha 'LosAngeles)
(losangeles denver chicago omaha)


> (DFS-nocycles 'LosAngeles 'Omaha)
(omaha denver chicago losangeles)
```

first search: same as before

second search: path is OK, but not optimal

```
> (DFS-nocycles '(0 0) '(4 0))
((4 0) (0 0))

> (DFS-nocycles '(0 0) '(2 0))
((2 0) (0 2) (4 2) (3 3) (3 0) (0 3) (4 3) (4 0) (0 0))
```

first search: same as before

second search: path is OK, but not optimal

```
> (DFS-nocycles '(1 2 3 8 6 4 7 space 5) '(1 2 3 8 space 4 7 6 5))
((1 2 3 8 space 4 7 6 5) (1 2 3 8 6 4 7 space 5))

> (DFS-nocycles '(1 2 3 8 6 space 7 5 4) '(1 2 3 8 space 4 7 6 5))
((1 2 3 8 space 4 7 6 5) (1 2 3 8 4 space 7 6 5) (1 2 space 8 4 3 7
6 5) (1 space 2 8 4 3 7 6 5) (space 1 2 8 4 3 7 6 5) (8 1 2 space 4
3 7 6 5) (8 1 2 7 4 3 space 6 5) (8 1 2 7 4 3 6 space 5) (8 1 2 7
space 3 6 4 5) (8 1 2 space 7 3 6 4 5) (space 1 2 8 7 3 6 4 5) ...
```

first search: same

second search: path is OK, but FAR from optima (length 510)

16

Breadth vs. depth

even with cycle checking, DFS may not find the shortest solution

- if state space is infinite, might not find solution at all

breadth first search (BFS)

- extend the search one level at a time
  i.e., from the start state, try every possible move (& remember them all)
  if don't reach goal, then try every possible move from those states
  . . .
- requires keeping a list of partially expanded search paths
- ensure breadth by treating the list as a queue
  when want to expand shortest path: take off front, extend & add to back

```
( (Omaha) )

( (Chicago Omaha) (Denver Omaha) )

( (Denver Omaha) (Denver Chicago Omaha) (LosAngeles Chicago Omaha) (Omaha Chicago Omaha) )

( (Denver Chicago Omaha) (LosAngeles Chicago Omaha) (Omaha Chicago Omaha)
  (LosAngeles Denver Omaha) (Omaha Denver Omaha) )

( (LosAngeles Chicago Omaha) (Omaha Chicago Omaha) (LosAngeles Denver Omaha)
  (Omaha Denver Omaha) (LosAngeles Denver Chicago Omaha) (Omaha Denver Chicago Omaha) )
```

17

Breadth first search

```
(define (BFS startState goalState)

  (define (BFS-paths paths)
    (cond ((null? paths) #f)
          ((equal? (caar paths) goalState) (car paths))
          (else (BFS-paths (append (cdr paths)
                                   (extend-all (car paths) (GET-MOVES (caar paths))))))))

  (define (extend-all path nextStates)
    (if (null? nextStates)
        '()
        (cons (cons (car nextStates) path)
              (extend-all path (cdr nextStates)))))

  (BFS-paths (list (list startState))))
```

BFS-paths takes a list of partially-searched paths

- if no paths remaining, then FAIL
- if leftmost path ends in goal state, then SUCCEED
- otherwise, extend leftmost path in all possible ways, add to end of path list, & recurse

18

BFS examples

```
> (BFS 'Omaha 'LosAngeles)
(losangeles chicago omaha)

> (BFS 'LosAngeles 'Omaha)
(omaha chicago losangeles)
```

first search:path is optimal

second search: path is optimal

```
> (BFS '(0 0) '(4 0))
((4 0) (0 0))

> (BFS '(0 0) '(2 0))
((2 0) (0 2) (4 2) (3 3) (3 0) (0 3) (0 0))
```

first search: same as before

second search: path is optimal

```
> (BFS '(1 2 3 8 6 4 7 space 5) '(1 2 3 8 space 4 7 6 5))
((1 2 3 8 space 4 7 6 5) (1 2 3 8 6 4 7 space 5))

> (BFS '(1 2 3 8 6 space 7 5 4) '(1 2 3 8 space 4 7 6 5))
((1 2 3 8 space 4 7 6 5)
 (1 2 3 8 6 4 7 space 5)
 (1 2 3 8 6 4 7 5 space)
 (1 2 3 8 6 space 7 5 4))
```

first search path is OK (although trivial)

second search: path is optimal

19

Breadth first search w/ cycle checking

```
(define (BFS-nocycles startState goalState)

  (define (BFS-paths paths)
    (cond ((null? paths) #f)
          ((equal? (caar paths) goalState) (car paths))
          (else (BFS-paths (append (cdr paths)
                                   (extend-all (car paths) (GET-MOVES (caar paths))))))))

  (define (extend-all path nextStates)
    (cond ((null? nextStates) '())
          ((member (car nextStates) path) (extend-all path (cdr nextStates)))
          (else (cons (cons (car nextStates) path)
                      (extend-all path (cdr nextStates))))))

  (BFS-paths (list (list startState))))
```

as before, can add cycle checking to avoid wasted search

- don't extend path if new state already occurs on the path

WILL CYCLE CHECKING AFFECT THE ANSWER FOR BFS?

IF NOT, WHAT PURPOSE DOES IT SERVE?

20

DFS vs. BFS

Advantages of DFS

- requires less memory than BFS since only need to remember the current path
- if lucky, can find a solution without examining much of the state space
- with cycle-checking, looping can be avoided

Advantages of BFS

- guaranteed to find a solution if one exists – in addition, finds optimal (shortest) solution first
- will not get lost in a blind alley (i.e., does not require backtracking or cycle checking)
- can add cycle checking to reduce wasted search

note: just because BFS finds the optimal *solution*, it does not necessarily mean that it is the optimal *control strategy*!

21

Iterative deepening

interesting hybrid: DFS with iterative deepening

- alter DFS to have a depth bound
  (i.e., search will fail if it extends beyond a specific depth bound)

- repeatedly call DFS with increasing depth bounds until a solution is found
  DFS with bound = 1
  DFS with bound = 2
  DFS with bound = 3
  . . .

advantages:

- yields the same solutions, in the same order, as BFS
- doesn't have the extensive memory requirements of BFS

disadvantage:

- lots (?) of redundant search – each time the bound is increased, the entire search from the previous bound is redone!

22

Depth first search with iterative deepening

```
(define (DFS-deepening startState goalState)

  (define (DFS-bounded bound)
    (or (extend (list startState) (GET-MOVES startState) bound)
        (DFS-bounded (+ bound 1))))

  (define (extend path moves depthBound)
    (cond ((> (length path) depthBound) #f)
          ((equal? (car path) goalState) path)
          ((null? moves) #f)
          (else (or (and (not (member (car moves) path))
                         (extend (cons (car moves) path)
                                 (GET-MOVES (car moves)) depthBound))
                    (extend path (cdr moves) depthBound)))))

  (DFS-bounded  1))
```

DFS-bounded performs DFS search with depth bound:

- try to extend path to reach goal with given bound
- if FAIL, then extend bound and try again

extend checks to make sure it hasn't exceeded depth bound:

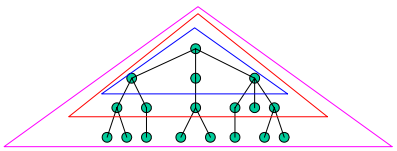
- checks current path length, if > depth bound then FAIL

WHAT WILL HAPPEN IF NO SOLUTION EXISTS? POSSIBLE FIXES?

23

Cost of iterative deepening

just how costly is iterative deepening?



1st pass searches entire tree up to depth 1

2nd pass searches entire tree up to depth 2 (including depth 1)

3rd pass searches entire tree up to depth 3 (including depths 1 & 2)

in reality, this duplication is not very costly at all!

Theorem: Given a "full" tree with constant branching factor B, the number of nodes at any given level is greater than the number of nodes in all levels above.

e.g., Binary tree (B = 2): 1 → 2 → 4 → 8 → 16 → 32 → 64 → 128 → …

repeating the search of all levels above for each pass merely doubles the amount of work for each pass.

in general, this repetition only increases the cost by a constant factor < (B+1)/(B-1)

24