

# CSC 550: Introduction to Artificial Intelligence

Fall 2008

## heuristics & informed search

- heuristics
- hill-climbing
  - bold + informed search
  - potential dangers, variants
- best first search
  - tentative + informed search
  - best-first search vs. DFS vs. BFS
- optimization problems
  - Algorithm A, admissibility, A\*

1

## Search strategies so far...

bold & uninformed: STUPID

tentative & uninformed: DFS (and variants), BFS

bold & informed: *hill-climbing*

- essentially, DFS utilizing a "measure of closeness" to the goal (a.k.a. a *heuristic function*)
- from a given state, pick the *best* successor state
  - i.e., the state with highest heuristic value
- stop if no successor is better than the current state
  - i.e., no backtracking so only need to store current path

EXAMPLE: travel problem, e.g. Omaha → LosAngeles

*heuristic(State) = -(crow-flies distance from goal)*

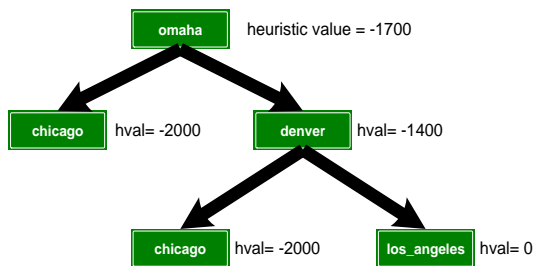
2

## Hill-climbing example

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

```
(define DISTANCES
  '((Omaha (Omaha 0) (Chicago 500) (Denver 400)
          (LosAngeles 1700))
    (Chicago (Chicago 0) (Omaha 500) (Denver 700)
            (LosAngeles 2000))
    (Denver (Denver 0) (Omaha 400) (Chicago 700)
            (LosAngeles 1400))
    (LosAngeles (LosAngeles 0) (Omaha 1700)
                (Chicago 2000) (Denver 1400))))
```

```
(define (HEURISTIC state goalState)
  (- 0 (cadr (assoc goalState (cdr (assoc state DISTANCES))))))
```



the heuristic guides the search, always moving closer to the goal

what if the flight from Denver to L.A. were cancelled?

3

## Hill-climbing implementation

```
(define (hill-climb startState goalState)
  (define (climb-path path)
    (if (equal? (car path) goalState)
        path
        (let ((nextStates (sort (GET-MOVES (car path)) better-state)))
          (if (or (null? nextStates) (not (better-state (car nextStates) (car path))))
              #f
              (climb-path (cons (car nextStates) path))))))
  (define (better-state state1 state2)
    (> (HEURISTIC state1 goalState) (HEURISTIC state2 goalState)))
  (climb-path (list startState)))
```

can utilize the `sort` function

- requires a comparison function (evals to #t if input1 should come before input2)

`climb-path` takes the current path:

- if the current state (car of path) is the goal, then SUCCEED
- otherwise, get all possible moves and sort based on heuristic value (high to low)
- if no moves or "best" move is no better than current state, then FAIL
- otherwise, add "best" move to path and recurse

4

## Hill-climbing in practice

```
> (hill-climb 'Omaha 'LosAngeles)
(losangeles denver omaha)
```

given these flights, actually finds optimal paths

```
;;; REMOVED Denver to L.A. FLIGHT
> (hill-climb 'Omaha 'LosAngeles)
#E
```

WHY?

```
;;; ADDED Omaha ↔ KC, KC ↔ LA FLIGHTS
;;; ASSUME Omaha is 200 mi from KC, KC is 1900 mi from LA
> (hill-climb 'Omaha 'LosAngeles)
(losangeles denver omaha)
> (hill-climb 'LosAngeles 'Omaha)
(losangeles kansascity omaha)
```

get different answers

WHY?

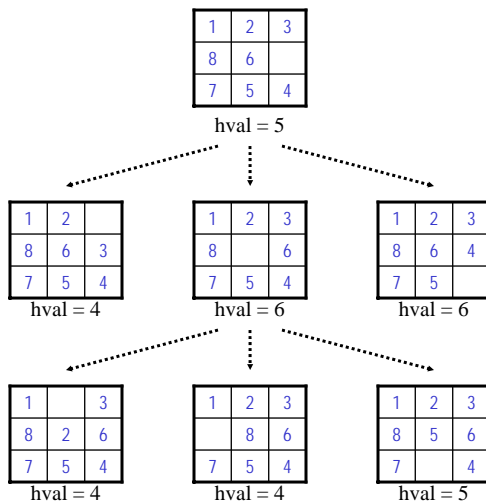
### hill-climbing only works if

- the heuristic is accurate (i.e., distinguishes "closer" states)
- the path to the goal is direct (i.e., state improves on every move)

5

## 8-puzzle example

*heuristic(State) = # of tiles in place, including the space*



start state has 5 tiles in place

of the 3 possible moves, 2 improve the situation

if assume left-to-right preference, move leads to a dead-end (no successor state improves the situation) so STOP!

6

## Intuition behind hill-climbing

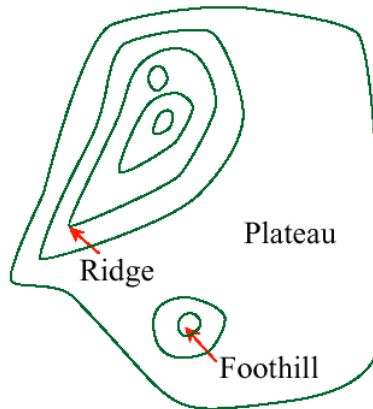
if you think of the state space as a topographical map (heuristic value = elevation), then hill-climbing selects the steepest gradient to move up towards the goal

### potential dangers

*plateau*: successor states have same values, no way to choose

*foothill*: local maximum, can get stuck on minor peak

*ridge*: foothill where N-step lookahead might help



7

## Hill-climbing variants

could generalize hill-climbing to continue even if the successor states look worse

- always choose best successor
- don't stop unless reach the goal or no successors

dead-ends are still possible (and likely if the heuristic is not perfect)

### simulated annealing

- allow moves in the wrong direction on a probabilistic basis
- decrease the probability of a backward move as the search continues

idea: early in the search, when far from the goal, heuristic may not be good  
heuristic should improve as you get closer to the goal

*approach is based on a metallurgical technique*

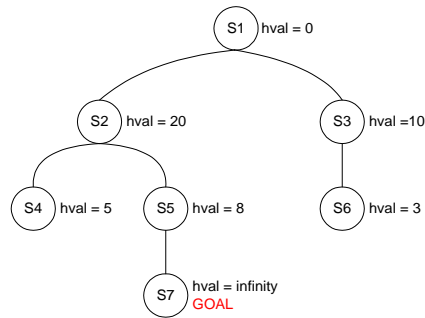
8

## Best first search

since bold, hill-climbing is dependent on a near-perfect heuristic

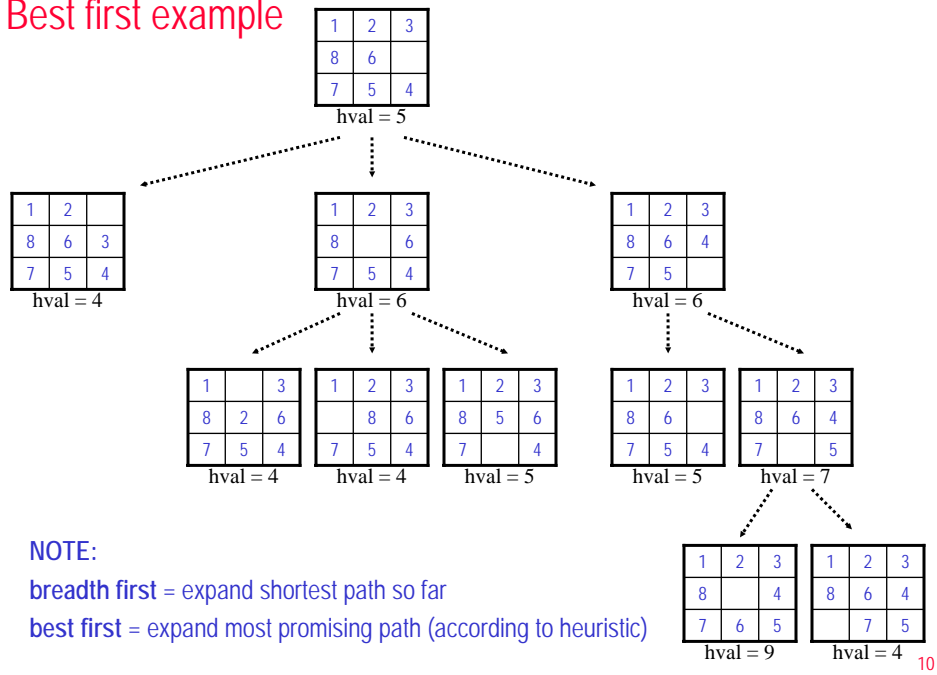
tentative & informed: best first search

- like breadth first search, keep track of all paths searched
- like hill-climbing, use heuristics to guide the search
- always expand the "most promising" path  
i.e., the path ending in a state with highest heuristic value



9

## Best first example



10

## Best first implementation

```
(define (best startState goalState)

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

  (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))))))

  (define (better-path path1 path2)
    (> (HEURISTIC (car path1) goalState) (HEURISTIC (car path2) goalState)))

  (best-paths (list (list startState))))
```

only difference from BFS implementation:

- newly extended path list is sorted by heuristic value of end states (high-to-low)
- since new paths are added at end & sort is stable, ties will favor older/shorter paths

11

## Travel example

```
> (best 'Omaha 'LosAngeles)
(losangeles denver omaha)
```

given these flights, actually finds optimal paths

```
> (best 'LosAngeles 'Omaha)
(omaha denver losangeles )
```

```
;;; REMOVED Denver to L.A. FLIGHT
```

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

WHY?

```
;;; ADDED Omaha ↔ KC, KC ↔ LA FLIGHTS
;;; ASSUME Omaha is 200 mi from KC, KC is 1900 mi from LA
```

```
> (best 'Omaha 'LosAngeles)
(losangeles denver omaha)
```

still different answers

```
> (best 'LosAngeles 'Omaha)
(losangeles kansascity omaha)
```

WHY?

note: best first search does not guarantee an "optimal" solution

- does not take the past into account, simply tries move that gets closest
- would select LA → KC → Omaha → DesMoines even if  
LA → Chicago → DesMoines were possible

12

## 8-puzzle example

```
(define (HEURISTIC state goalState)
  (cond ((null? state) 0)
        ((equal? (car state) (car goalState))
         (+ 1 (HEURISTIC (cdr state) (cdr goalState))))
        (else (HEURISTIC (cdr state) (cdr goalState)))))
```

heuristic counts number of tile matches with goal

```
> (best '(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))
```

- DFS-nocycles required > 400 moves
- BFS found same soln, but 6 times slower

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

- DFS-nocycles hangs
- BFS found same soln, but > 200 times slower

13

## Comparing best first search

unlike hill-climbing, best first search can handle "dips" in the search

→ not so dependent on a perfect heuristic

depth first search and breadth first search may be seen as special cases of best first search

DFS: heuristic value is distance (number of moves) from start state

BFS: heuristic value is inverse of distance (number of moves) from start state

or, procedurally:

DFS: assign all states the same heuristic value  
when adding new paths, add equal heuristic values at the *front*

BFS: assign all states the same heuristic value  
when adding new paths, add equal heuristic values at the *end*

14

## Optimization problems

consider a related search problem:

- instead of finding the shortest path (i.e., fewest moves) to a solution, suppose we want to minimize some cost

EXAMPLE: airline travel problem

- could associate costs with each flight, try to find the cheapest route
- could associate distances with each flight, try to find the shortest route

we could use a strategy similar to breadth first search

- repeatedly extend the minimal cost path  
→ search is guided by the cost of the path so far

but such a strategy ignores heuristic information

- would like to utilize a best first approach, but not directly applicable  
→ search is guided by the remaining cost of the path

IDEAL: combine the intelligence of both strategies

- cost-so-far component of breadth first search (to optimize actual cost)
- cost-remaining component of best first search (to make use of heuristics)

15

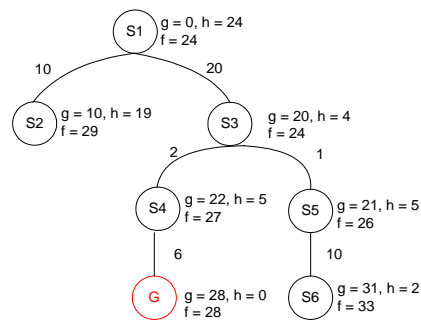
## Algorithm A

associate 2 costs with a path

g            actual cost of the path so far  
h            heuristic estimate of the remaining cost to the goal\*  
f = g + h    combined heuristic cost estimate

\*note: the heuristic value is inverted relative to best first

Algorithm A: best first search using f as the heuristic

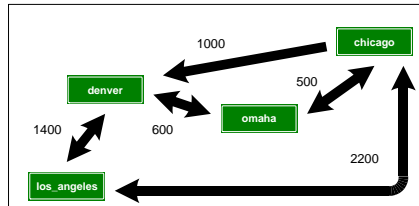


16

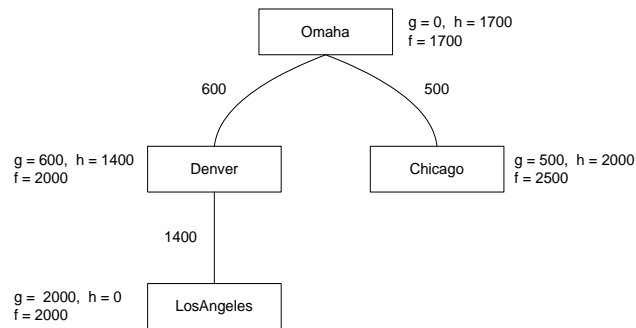
## Travel problem revisited

g: cost is actual distances per flight

h: cost estimate is crow-flies distance



	Omaha	Chicago	Denver	LosAngeles
Omaha	0	500	400	1700
Chicago	500	0	700	2000
Denver	400	700	0	1400
LosAngeles	1700	2000	1400	0



17

## Travel problem reimplemented

```
(define MOVES
  '((Omaha --> Chicago 500) (Omaha --> Denver 600)
    (Chicago --> Denver 1000) (Chicago --> LosAngeles 2200) (Chicago --> Omaha 500)
    (Denver --> LosAngeles 1400) (Denver --> Omaha 600)
    (LosAngeles --> Chicago 2200) (LosAngeles --> Denver 1400)))

(define DISTANCES
  '((Omaha (Omaha 0) (Chicago 500) (Denver 400) (LosAngeles 1700))
    (Chicago (Chicago 0) (Omaha 500) (Denver 700) (LosAngeles 2000))
    (Denver (Denver 0) (Omaha 400) (Chicago 700) (LosAngeles 1400))
    (LosAngeles (LosAngeles 0) (Omaha 1700) (Chicago 2000) (Denver 1400))))

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

(define (H state goalState)
  (cadr (assoc goalState (cdr (assoc state DISTANCES)))))
```

store actual  
cost of each  
move

modify GET-  
MOVES to  
return pairs:  
(state cost)

H function is  
inverse of  
HEURISTIC

18

## Algorithm A implementation

```
(define (a-tree startState goalState)

  (define (a-paths paths)
    (cond ((null? paths) #f)
          ((equal? (cadar paths) goalState) (car paths))
          (else (a-paths (sort better-path
                               (append (cdr paths)
                                       (extend-all (car paths)
                                                  (GET-MOVES (cadar paths))))))))))

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

  (define (better-path path1 path2)
    (< (+ (car path1) (H (cadar path1) goalState))
        (+ (car path2) (H (cadar path2) goalState))))

  (a-paths (list (list 0 startState))))
```

### differences from best first search:

- put actual cost of path (sum of g's) at front of each path
- when path is extended, add cost of move to total path cost
- paths are sorted by (actual cost of path so far + H value for current state)

note: much more efficient to implement using graph, but more complex

19

## Travel example

```
> (a-tree 'Omaha 'LosAngeles)
(2000 losangeles denver omaha)
> (a-tree 'LosAngeles 'Omaha)
(2000 omaha denver losangeles)
```

```
;;; ADDED Omaha ↔ KC, KC ↔ LA FLIGHTS
;;; ASSUME Omaha is 200 mi from KC, KC is 1900 mi from LA

> (a-tree 'Omaha 'LosAngeles)
(2400 losangeles denver omaha)
> (a-tree 'LosAngeles 'Omaha)
(2000 losangeles denver omaha)
```

Algorithm A finds "cheapest" path

unlike best first search, takes total cost into account so guaranteed "cheapest" path

note: Algorithm A finds the path with least cost (here, distance)  
not necessarily the path with fewest steps

```
;;; CHANGE Chicago → LA flight to 2500 mi

> (a-tree 'Chicago 'LosAngeles)
(2400 losangeles denver chicago)
> (a-tree 'LosAngeles 'Omaha)
(2000 omaha denver losangeles)
```

if the flight from Chicago to L.A. was 2500 miles (instead of 2200)

Chicago→Denver→LA (2400 mi)

would be shorter than

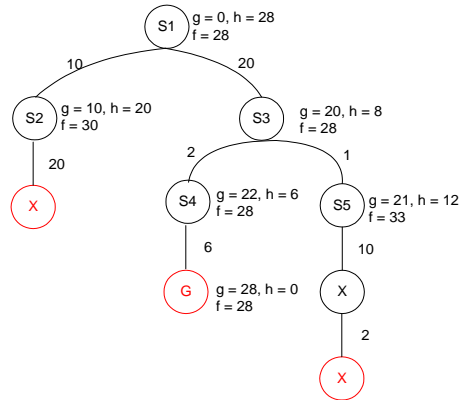
Chicago→LA (2500 mi)

20

## Algorithm A vs. hill-climbing

if the cost estimate function  $h$  is perfect, then  $f$  is a perfect heuristic

→ Algorithm A is deterministic



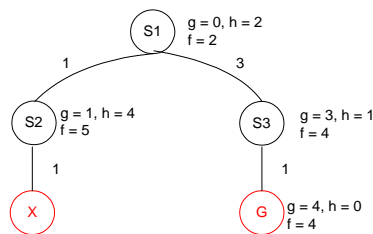
if know actual costs for each state, Algorithm A reduces to hill-climbing

21

## Admissibility

in general, actual costs are unknown at start – must rely on heuristics

if the heuristic is imperfect, NOT guaranteed to find an optimal solution



if a control strategy is guaranteed to find an optimal solution (when a solution exists), we say it is *admissible*

if cost estimate  $h$  never overestimates actual cost, then Algorithm A is admissible (when admissible, Algorithm A is commonly referred to as Algorithm A\*)

22

## Heuristic examples

is our heuristic for the travel problem admissible?

*h: crow-flies distance from Goal*

8-puzzle heuristic?

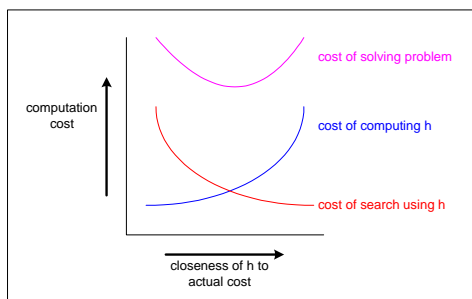
*h: number of tiles out of place, including the space*

23

## Cost of the search

the closer  $h$  is to the actual cost function, the fewer states considered

- however, the cost of computing  $h$  tends to go up as it improves



the best algorithm is one that minimizes the total cost of the solution

also, admissibility is not always needed or desired

**Graceful Decay of Admissibility:** If  $h$  rarely overestimates the actual cost by more than  $D$ , then Algorithm A will rarely find a solution whose cost exceeds optimal by more than  $D$ .

24

## Flashlight example

consider the flashlight puzzle:

Four people are on one side of a bridge. They wish to cross to the other side, but the bridge can only take the weight of two people at a time. It is dark and they only have one flashlight, so they must share it in order to cross the bridge. Assuming each person moves at a different speed (able to cross in 1, 2, 5 and 10 minutes, respectively), find a series of crossings that gets all four across *in the minimal amount of time*.

state representation?

cost of a move?

heuristic?

25

## Flashlight implementation

state representation must identify the locations of each person and the flashlight

```
((1 2 5 10) left ())
```

**note:** must be careful of permutations

e.g., (1 2 5 10) = (1 5 2 10) so must make sure there is only one representation per set

**solution:** maintain the lists in sorted order, so only one permutation is possible

only 3 possible moves:

1. if the flashlight is on left and only 1 person on left, then  
move that person to the right (cost is time it takes for that person)
2. if flashlight is on left and at least 2 people on left, then  
select 2 people from left and move them to right (cost is max time of the two)
3. if the flashlight is on right, then  
select a person from right and move them to left (cost is time for that person)

heuristic:

h: number of people in wrong place

26

## Flashlight code

```
(define (GET-MOVES state)

  (define (generate-all-pairs lst)
    ;; generates a list of all pairs of items from lst
  )

  (define (make-each-move move-list)
    ;; generates a list of all states obtainable from the current
    ;; state by making a move from move-list
  )

  (if (equal? (cadr state) 'left)
      (make-each-move (append (map list (car state)) (generate-all-pairs (car state))))
      (make-each-move (append (map list (caddr state)) (generate-all-pairs (caddr state))))))

(define (H state goalState)
  (+ (length (remove-all (car goalState) (car state)))
     (length (remove-all (caddr goalState) (caddr state)))))
```

Algorithm A finds *an* optimal  
solution  
(more than one are optimal)

```
> (a-tree '((1 2 5 10) left ()) '(() right (1 2 5 10)))
(17
 ((() right (1 2 5 10))
 ((1 2) left (5 10))
 ((1) right (2 5 10))
 ((1 5 10) left (2))
 ((5 10) right (1 2))
 ((1 2 5 10) left ()))
```

27