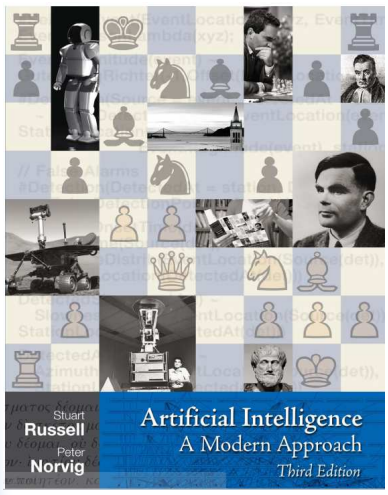
Intelligent Autonomous Agents
and Cognitive Robotics
Solving problems by searching

Rainer Marrone

Hamburg University of Technology

Slides based on Hwee Tou Ng's

Literature



- Chapter 3

2

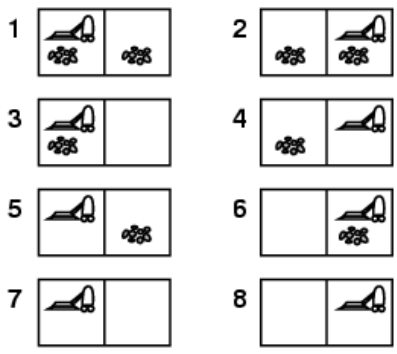
Problem types

- Single-state problem: Deterministic, fully observable
  - Agent knows exactly which state it will be in; can calculate optimal action sequence to reach the goal
- Multiple state problem: Deterministic, partially/not observable
  - Agent must reason about sequences of actions and states assumed while working towards goal state.
- Contingency problem: Nondeterministic and partially observable
  - Percepts provide new information about current state
  - Solution is a contingent plan or policy
  - Often interleave search and execution
- Exploration problem: Unknown state space
  - Discover and learn about environment while taking actions

3

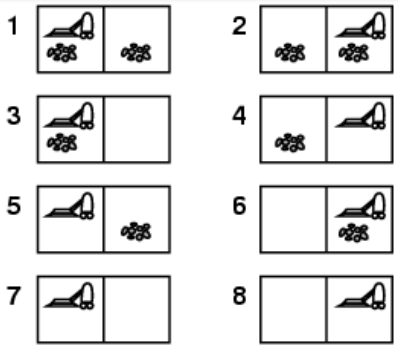
Example: vacuum world

- Single-state, start in #5. Solution?
  [Right, Suck]
- Multiple-state, start in {*1,2,3,4,5,6,7,8*} e.g., *Right* goes to {*2,4,6,8*}
  Solution?
  [Right,Suck,Left,Suck]



4

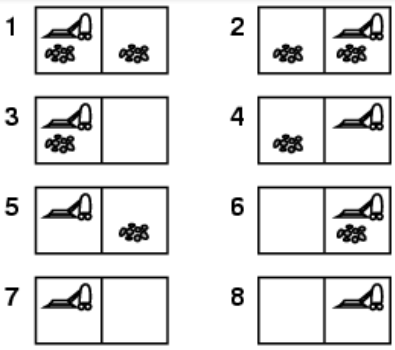
Example: vacuum world



- Contingency
  - Nondeterministic: *Suck* may dirty a clean carpet
  - Partially observable: location, dirt at current location.
  - Percept: *[L, Clean],* i.e., start in #5 or #7
    Solution

5

Example: vacuum world



- Contingency
  - Nondeterministic: *Suck* may dirty a clean carpet
  - Partially observable: location, dirt at current location.
  - Percept: *[L, Clean],* i.e., start in #5 or #7
    Solution? *[Right,* ***if*** *dirt* ***then*** *Suck]*

6

Solving problems by searching

- We will discuss solutions for all the different settings.
- We start with simple searches and modify them for more complex settings

7

Tree search algorithms

- Basic idea:
  - offline, simulated exploration of state space by generating successors of already-explored states (a.k.a. expanding states)

```
Function Tree-Search(problem, strategy) returns a solution, or failure
    initialize the search tree using the initial state problem
    loop do
        if there are no candidates for expansion then return failure
        choose a leaf node for expansion according to strategy
        if the node contains a goal state then return the corresponding solution
        else expand the node and add resulting nodes to the search tree
    end
```

8

Measuring search performance

- A search strategy is defined by picking the order of node expansion
- Strategies are evaluated along the following dimensions:
  - completeness: does it always find a solution if one exists?
  - time complexity: number of nodes generated
  - space complexity: maximum number of nodes in memory
  - optimality: does it always find a least-cost solution?
- Time and space complexity are measured in terms of
  - *b:* maximum branching factor of the search tree
  - *d:* depth of the least-cost solution
  - *m*: maximum depth of the state space (may be ∞)

9

Uninformed search strategies

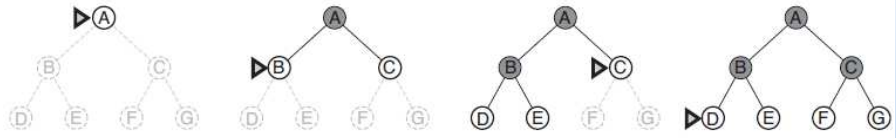
Uninformed search strategies use only the information available in the problem definition

- Breadth-first search
- Depth-first search
- Depth-limited search
- Iterative deepening search

10

Breadth-first search

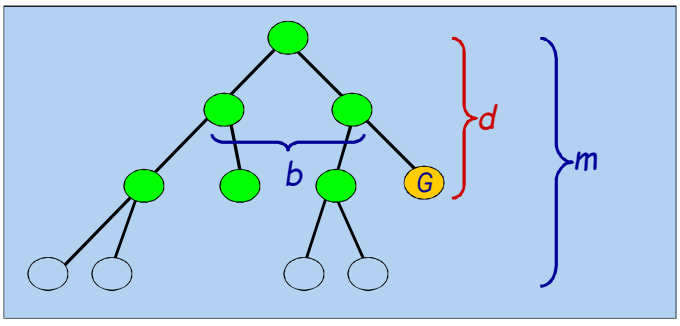
- Expand shallowest unexpanded node
- Implementation:
  - *fringe* is a FIFO queue, i.e., new successors go at end



11

Time complexity of breadth-first search

- *If a goal node is found on depth d of the tree, all nodes up till that depth are created.*

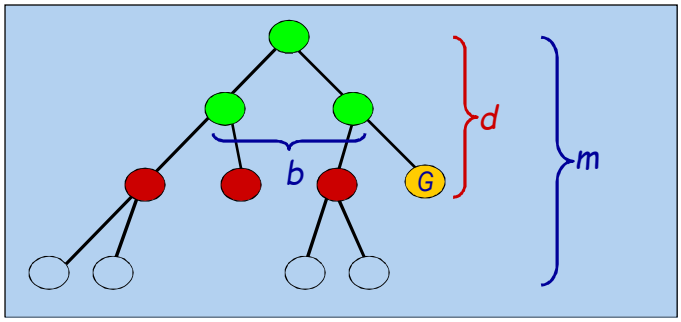


- *Thus:* $O(b^d)$

12

Space complexity of breadth-first

- *Largest number of nodes in QUEUE is reached on the level d of the goal node.*



- *QUEUE contains all*  *and*  *nodes. (Thus: 4) .*
- *In General: b^d*

13

Properties of breadth-first search

- Complete? Yes (if *b* is finite)
- Time? $1+b+b^2+b^3+\ldots+b^d = O(b^d)$
- Space? $O(b^{d+1})$ (keeps every node in memory)
  $O(b^d)$ *(only fringe)*
- Optimal? *Yes (if cost = 1 per step)*

14

Complexity example

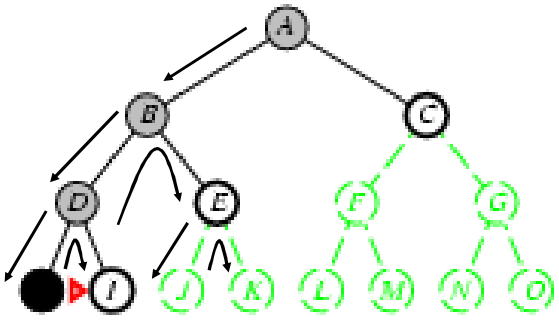
Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabyte
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabyte
14	10^{14}	3.5 years	99 petabytes
16	10^{16}	350 years	10 exabytes

Figure 3.13 Time and memory requirements for breadth-first search. The numbers shown assume branching factor $b = 10$; 1 million nodes/second; 1000 bytes/node.

15

Depth-first search

- Expand deepest unexpanded node



16

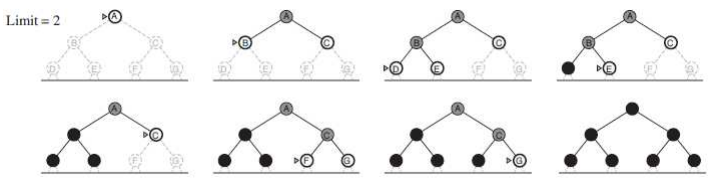
Properties of depth-first search

- Complete? No: fails in infinite-depth spaces
  → complete in finite spaces
- Time? $O(b^m)$: terrible if m is much larger than d
  - but if solutions are dense, may be much faster than breadth-first
- Space? $O(bm)$, i.e., linear space!
- Optimal? No

17

Depth-limited search

- depth-first search with depth limit l, i.e., nodes at depth l have no successors
- Solves infinite path problem
- Incomplete if $l<d$ (shallowest goal node)
- Nonoptimal if $l>d$



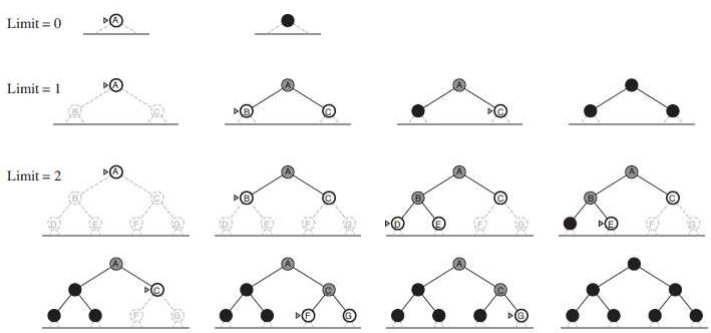
18

Iterative deepening search

```
function ITERATIVE-DEEPENING-SEARCH( problem) returns a solution, or failure
    inputs: problem, a problem
    for depth ← 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH( problem, depth)
        if result ≠ cutoff then return result
```

19

Iterative deepening search



20

Iterative deepening search

- Number of nodes generated in a depth-limited search to depth *d* with branching factor *b*:

  $N_{DLS/BFS} = b^0 + b^1 + b^2 + \ldots + b^{d-2} + b^{d-1} + b^d$

- Number of nodes generated in an iterative deepening search to depth *d* with branching factor *b*:

  $N_{IDS} = (d+1)b^0 + d\,b^1 + (d-1)b^2 + \ldots + 3b^{d-2} + 2b^{d-1} + 1b^d$

- For *b = 10*, *d = 5*,
  - $N_{DLS} = 1 + 10 + 100 + 1{,}000 + 10{,}000 + 100{,}000 = 111{,}111$
  - $N_{IDS} = 6 + 50 + 400 + 3{,}000 + 20{,}000 + 100{,}000 = 123{,}456$

- Overhead = (123,456 - 111,111)/111,111 = 11%

21

Properties of iterative deepening search

- Complete? Yes
- Time? $(d+1)b^0 + d\,b^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$
- Space? $O(bd)$
- Optimal? Yes, if step cost = 1

22

Summary of algorithms

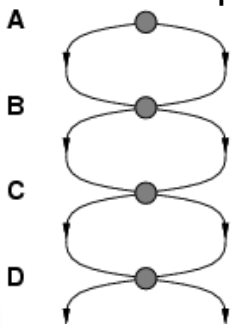
Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[a]	Yes[a,b]	No	No	Yes[a]	Yes[a,d]
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes[c]	Yes	No	No	Yes[c]	Yes[c,d]

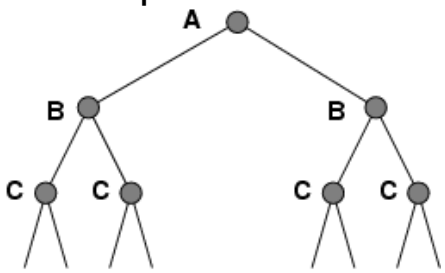
Figure 3.21 Evaluation of tree-search strategies. b is the branching factor; d is the depth of the shallowest solution; m is the maximum depth of the search tree; l is the depth limit. Superscript caveats are as follows: [a] complete if b is finite; [b] complete if step costs $\geq \epsilon$ for positive ϵ; [c] optimal if step costs are all identical; [d] if both directions use breadth-first search.

23

Repeated states

- Failure to detect repeated states can turn a linear problem into an exponential one!





24

Graph search

```
function GRAPH-SEARCH( problem, fringe) returns a solution, or failure
    closed ← an empty set
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
        if STATE[node] is not in closed then
            add STATE[node] to closed
            fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

Remember nodes visited

25

Beyond classical search

- Informed search
  - Greedy best-first search
  - A* search
- Admissible heuristics, creating heuristics
- Local search algorithms
  - Hill-climbing search
  - Simulated annealing search
  - Local beam search
  - Genetic algorithms
- Searching with nondeterministic actions

26

Best-first search

- Idea: use a heuristic evaluation function *f(n)* for each node
  - estimate of "desirability"
  - → Expand most desirable unexpanded node
- Implementation:
  Order the nodes in fringe in decreasing order of desirability
- Special cases:
  - greedy best-first search
  - A* search

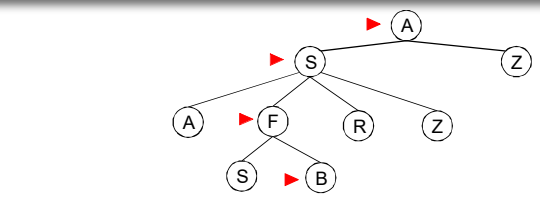
27

Greedy best-first search

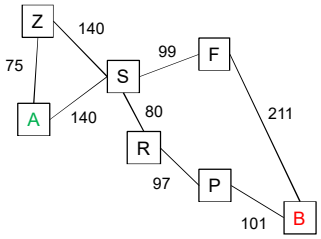
- Evaluation function
  f(n) = *h(n)* (heuristic)= estimate of cost from *n* to *goal*
  e.g., $h_{SLD}(n)$ = straight-line distance from *n* to goal node
- Greedy best-first search expands the node that **appears** to be closest to the goal
- Stop if the goal node appears on the fringe

28

Greedy best-first search example: Go from A to B



140
99
211
450



	B
A	366
Z	374
S	253
R	193
P	100
F	176
B	0

29

Properties of greedy best-first search

- Complete? No – can get stuck in loops,
  but can use graph search
- Time? $O(b^m)$, but a good heuristic can give dramatic improvement
- Space? $O(b^m)$ -- keeps all nodes in memory
- Optimal? No

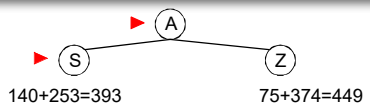
30

A* search

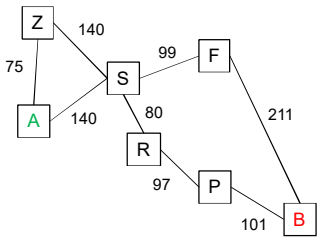
- Idea: avoid expanding nodes that are already expensive
- Evaluation function $f(n) = g(n) + h(n)$
  $g(n)$ = cost so far to reach n
  $h(n)$ = estimated cost from n to goal
- Goal node must also be expanded

32

A* search example: Go from A to B



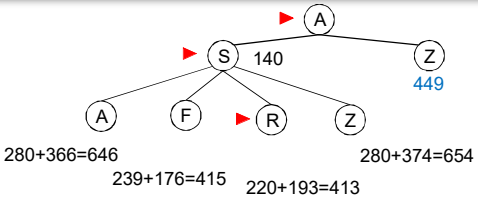
$f(n) = g(n) + h(n)$

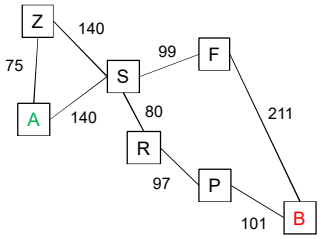


	B
A	366
Z	374
S	253
R	193
P	100
F	176
B	0

33

A* search example: Go from A to B

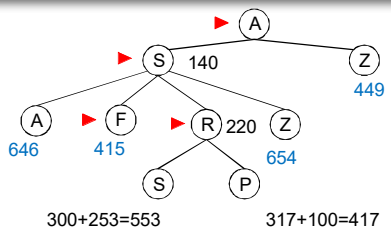


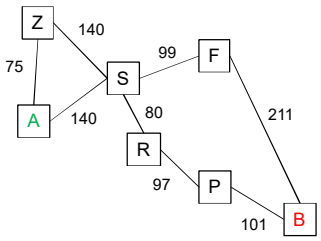


	B
A	366
Z	374
S	253
R	193
P	100
F	176
B	0

34

A* search example: Go from A to B

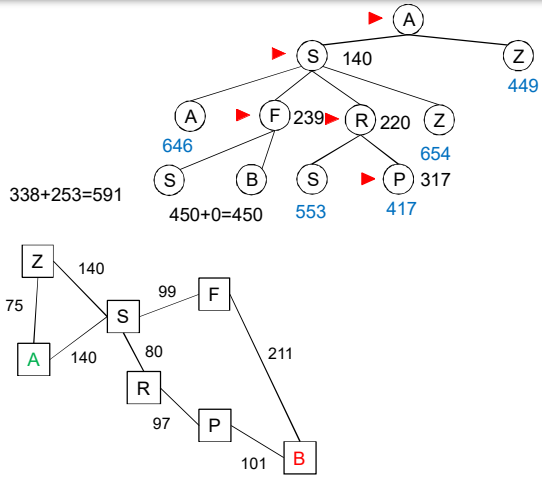




	B
A	366
Z	374
S	253
R	193
P	100
F	176
B	0

35

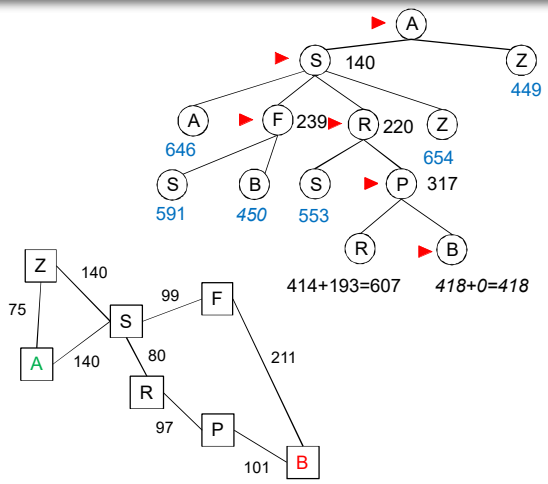
A* search example: Go from A to B



	B
A	366
Z	374
S	253
R	193
P	100
F	176
B	0

36

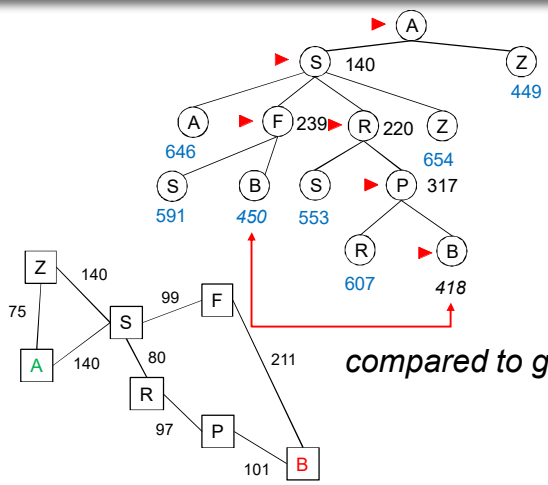
A* search example: Go from A to B



	B
A	366
Z	374
S	253
R	193
P	100
F	176
B	0

37

A* search example: Go from A to B



compared to greedy

	B
A	366
Z	374
S	253
R	193
P	100
F	176
B	0

38

Admissible heuristics

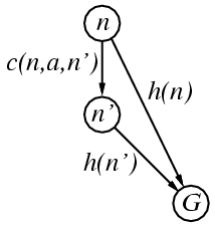
- A heuristic *h(n)* is admissible if for every node *n*,
  h(n) ≤ *h*(n)*, where *h*(n)* is the true cost to reach the goal state from *n*.
  An admissible heuristic never overestimates the cost to reach the goal, i.e., it is optimistic
- Example: $h_{SLD}(n)$ (never overestimates the actual road distance)
- Theorem: If *h(n)* is admissible,
  A* using `TREE-SEARCH` is optimal
- For graph searches we nee a stronger criteria

39

Consistent heuristics

- "*each side of a triangle cannot be longer than the sum of the other two sides*"
- A heuristic is consistent if for every node *n*, every successor *n'* of *n* generated by any action *a*,

  $h(n) \leq c(n,a,n') + h(n')$



- If *h* is consistent, we have

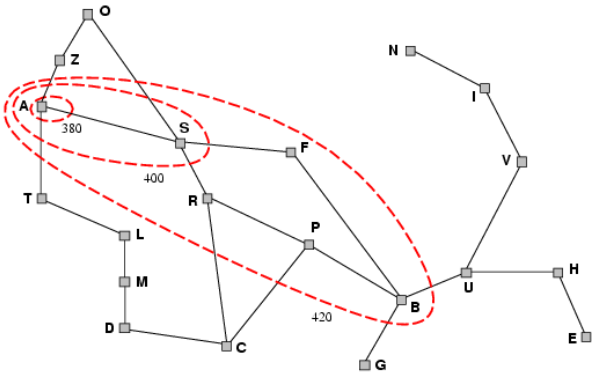
$$
\begin{aligned}
f(n') &= g(n') + h(n') \\
&= g(n) + c(n,a,n') + h(n') \\
&\geq g(n) + h(n) \\
&= f(n)
\end{aligned}
$$

- i.e., *f(n)* is non-decreasing along any path.
- Theorem: If *h(n)* is consistent, A* using `GRAPH-SEARCH` is optimal

40

Optimality of A*

- A* expands nodes in order of increasing *f* value
- A* will search all path with f(n)<C* (completeness)
- A* never expands nodes with f>C* (the true cost)



Map, showing contours at f=380, f=400, and f=420.

41

Properties of A*

- Complete? Yes
- Time? The number of states in the goal contour can still be exponential.
- Space?
  Keeps all generated nodes in memory, as do all graph search algorithms.
- Optimal? Yes

Not practical for very large scale problems

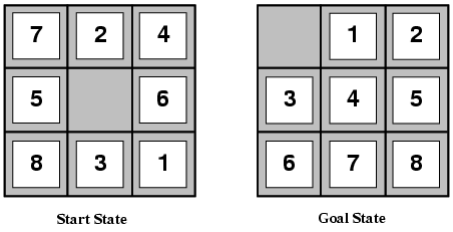
42

Admissible heuristics

E.g., for the 8-puzzle:

- $h_1(n)$ = number of misplaced tiles
- $h_2(n)$ = total Manhattan distance

(i.e., no. of squares from desired location of each tile)



- $h_1(S)$ = ? 8
- $h_2(S)$ = ? 3+1+2+2+2+3+3+2 = 18

43

Empirical Evaluation

- d = distance from goal
- Average over 100 instances
- IDS: Iterative Deepening Search (the best you can do without any heuristic)

nodes expanded

	Search Cost		
d	IDS	A*(h_1)	A*(h_2)
2	10	6	6
4	112	13	12
6	680	20	18
8	6384	39	25
10	47127	93	39
12	364404	227	73
14	3473941	539	113
16	–	1301	211
18	–	3056	363
20	–	7276	676
22	–	18094	1219
24	–	39135	1641

44

Dominance

- If $h_2(n) \geq h_1(n)$ for all n (both admissible) then h_2 dominates h_1
- *Is h_2 always better than h_1?*
- f(n) < C* (true cost)
- Every node h(n) < C* -g(n) will surely get expanded
- Because $h_2(n) \geq h_1(n)$ every node of h_2 will also be expanded from h_1, and h_1 will cause other nodes to be expanded

45

Relaxed problems

- A problem with fewer restrictions on the actions is called a relaxed problem
- The cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem
- If the rules of the 8-puzzle are relaxed so that a tile can move anywhere, then $h_1(n)$ gives the shortest solution
- If the rules are relaxed so that a tile can move to any adjacent square, then $h_2(n)$ gives the shortest solution

46

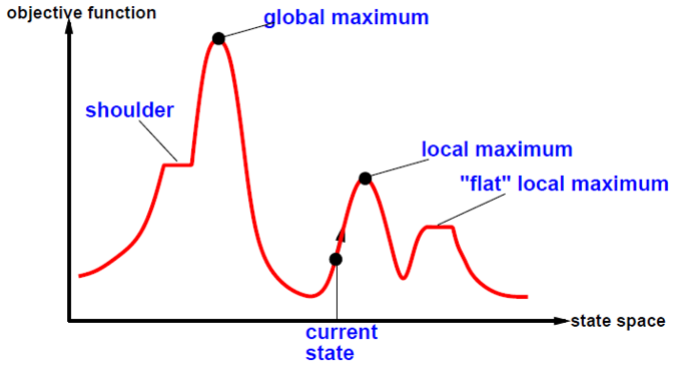
Local search algorithms

- In many optimization problems, the path to the goal is irrelevant; the goal state itself is the solution
- State space = set of "complete" configurations
- Find configuration satisfying constraints, e.g., n-queens; integrated-circuit design; factory-floor layout,
- In such cases, we can use local search algorithms. Keep a single "current" state, try to improve it

47

State space and objective Funtion

Useful to consider state space landscape



49

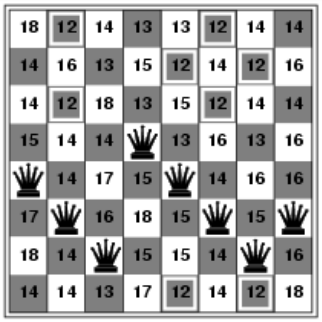
Hill-climbing search

- "Like climbing Everest in thick fog with amnesia“ (Russell, Norvig)

```
function HILL-CLIMBING( problem) returns a state that is a local maximum
   inputs: problem, a problem
   local variables: current, a node
                    neighbor, a node
   current ← MAKE-NODE(INITIAL-STATE[problem])
   loop do
      neighbor ← a highest-valued successor of current
      if VALUE[neighbor] ≤ VALUE[current] then return STATE[current]
      current ← neighbor
   end
```

50

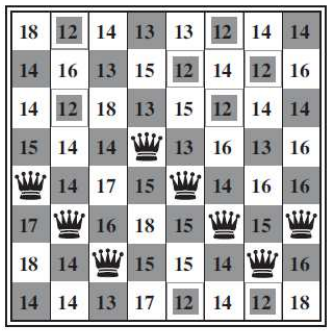
Hill-climbing search: 8-queens problem

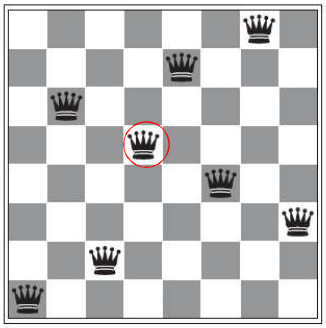


- The successors of a state are all possible states generated by moving a single queen to another square in the same column (so each state has 8 × 7=56 successors)
- Cost function: *h* = number of pairs of queens that are attacking each other, either directly or indirectly
- *h* = *17* for the above state, best moves are marked.

51

Hill-climbing search: 8-queens problem





(a) (b)

Figure 4.3 (a) An 8-queens state with heuristic cost estimate $h = 17$, showing the value of h for each possible successor obtained by moving a queen within its column. The best moves are marked. (b) A local minimum in the 8-queens state space; the state has $h = 1$ but every successor has a higher cost.

52

Observations

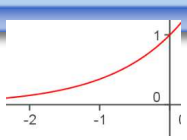
- Get stuck 86% vs 14% success
- Taking 4 steps only if successful
  3 steps if getting stuck (17 Million states)

- If sideways are allowed (100), success in 94%. Increase of cost 21 steps.
- Variants
  - Stochastic hill climbing
  - First-choice hill climbing
  - Random restart

53

Simulated annealing search

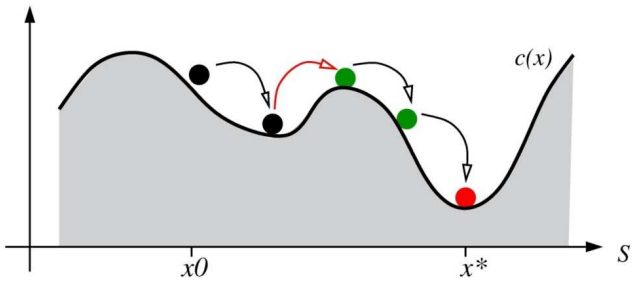
Idea: escape local maxima by allowing some "bad" moves
but gradually decrease their size and frequency




```
function SIMULATED-ANNEALING( problem, schedule) returns a solution state
   inputs: problem, a problem
           schedule, a mapping from time to "temperature"
   local variables: current, a node
                    next, a node
                    T, a "temperature" controlling prob. of downward steps
   current ← MAKE-NODE(INITIAL-STATE[problem])
   for t ← 1 to ∞ do
       T ← schedule[t]
       if T = 0 then return current        Stopping criteria
       next ← a randomly selected successor of current
       ΔE ← VALUE[next] – VALUE[current]
       if ΔE > 0 then current ← next
       else current ← next only with probability e^(Δ E/T)
```

54

Simulated Annealing



55

Properties of simulated annealing search

- One can prove:
  If *T* decreases slowly enough, then simulated annealing search will find a global optimum/minimum with probability approaching 1
- Widely used in VLSI layout, airline scheduling, etc.

56

Local beam search

- Keep track of *k* states rather than just one
- Start with *k* randomly generated states
- At each iteration, all the successors of all *k* states are generated
- If any one is a goal state, stop; else select the *k* best successors from the complete list and repeat.

57

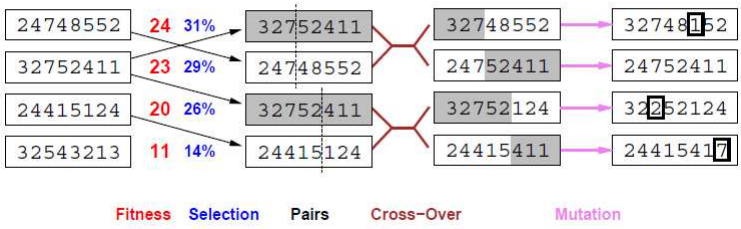
Genetic algorithms

- A variant of stochastic beam search. But a successor state is generated by different operations.
- Start with *k* randomly generated states (population)
- A state is represented as a string over a finite alphabet (often a string of 0s and 1s)
- Evaluation function (fitness function). Higher values for better states.
- Produce the next generation of states by selection, crossover, and mutation

58

Genetic algorithms

= stochastic local beam search + generate successors from pairs of states

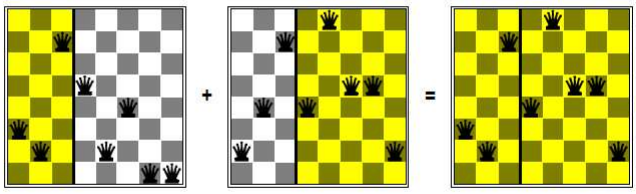


- Fitness function: number of non-attacking pairs of queens (min = 0, max = 7 + 6 + 5 + 4 + 3 + 2 + 1 = 28)
- 24/(24+23+20+11) = 31%
- 23/(24+23+20+11) = 29% etc

59

Genetic algorithms

Crossover helps iff substrings are meaningful components



GAs ≠ evolution: e.g., real genes encode replication machinery!

- *How many crossover, mutations*
- *How to encode the problem, fitness function*
- *One (more popular) vs. two child's*

60

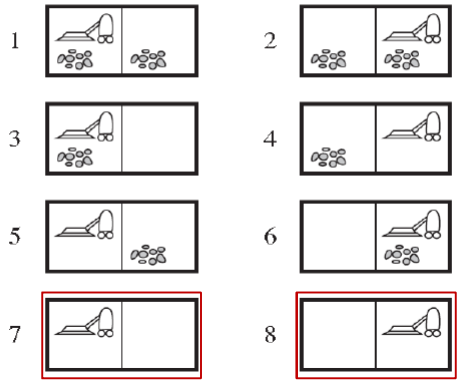
Nondeterministic/Uncertain actions

- What if the outcome of actions is non deterministic
- Erratic vacuum cleaner
  - When applied to a *dirty* square the square is *cleaned and adjacent square sometimes also*.
  - When applied to a *clean square*, *sometimes dirt is deposited* on that square
    ➔ need to have **contingency plan/strategy**

61

Possible states

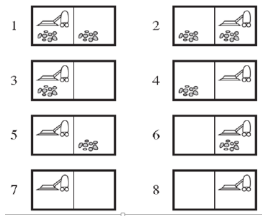
- The eight possible states of the erratic vacuum world – states 7 and 8 are goal states



62

Multiple States

- The result of an action is a set of states
- *Suck* in state 1 returns the set {5,7}
- We also need to generalize the concept of solution, since for example, if we **start** in state **1** there is no single sequence of actions to solve the problem instead we need a contingency plan like:
  [*Suck,* **if** *State*=5 **then** [*Right, Suck*] **else** []]



63

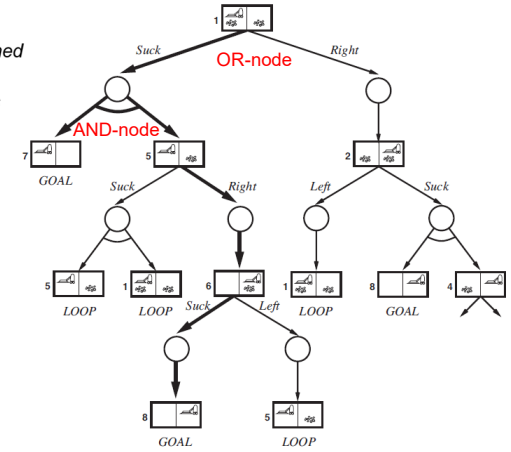
AND-OR Search trees

- Branching is also introduced by the environment choice of the outcome of actions.

When applied to a *dirty* square the square is *cleaned and adjacent square sometimes also*.

When applied to a *clean square*, *sometimes dirt is deposited* on that square

- This leads to AND-OR trees
- The bold path is the current plan



64

AND-OR Search trees

- A solution is a subtree
  - has a goal node at every leaf
  - specifies one action at OR-nodes
  - Includes every outcome branch at AND-nodes
- Leads to *if then else* or *case* if more then two outcomes

65

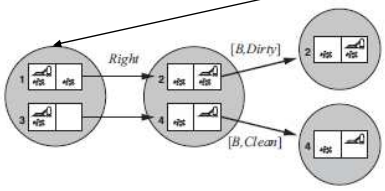
AND-OR Search trees

- Can also be explored by BFS and best-first methods
- Heuristic functions must be modified to estimate cost of a contingent solution rather than a sequence
- The notion of admissibility carries over.

66

Partial Observable Env.

- The vacuum cleaner has only partial information, e.g., if he is in the left square he does not see the state of the right square.
  If the initial state is left and dirt, we have a **belief state** rather than a physical state

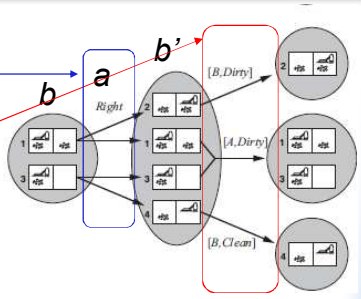


- But we also have uncertain actions: Move action may fail

67

Uncertain actions & partial observable

- Prediction:
  b'=Predict(b, a)
- Possible observations in b'
  Percepts(b')={o: o=PERCEPT(b')}



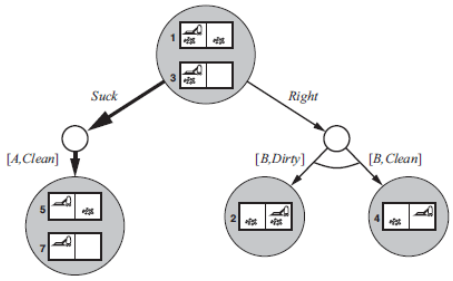
- Update of belief state:
  b_o= UPDATE(b',o)= {s:o = PERCEPT(s) and s∈b'}
- Putting all together:

$$\text{RESULTS}(b,a) = \{b_o : b_o = \text{UPDATE}(\text{PREDICT}(b,a),o) \text{ and } o \in \text{POSSIBLE-PERCEPTS}(\text{PREDICT}(b,a))\}$$

68

Structure

- Can use different search structures
- E.g. And-Or-Graphs



69