Review Search

This material: Chapter 1-2, 3.1-3.7, 4.1-4.2
Next Lecture Chapter 5.1-5.5 (Adversarial Search)
(Please read lecture topic material before and after each lecture on that topic)

- Search: complete architecture for intelligence?
  - Search to solve the problem, “What to do?”
- Problem formulation:
  - Handle infinite or uncertain worlds
- Search methods:
  - Uninformed, Heuristic, Local

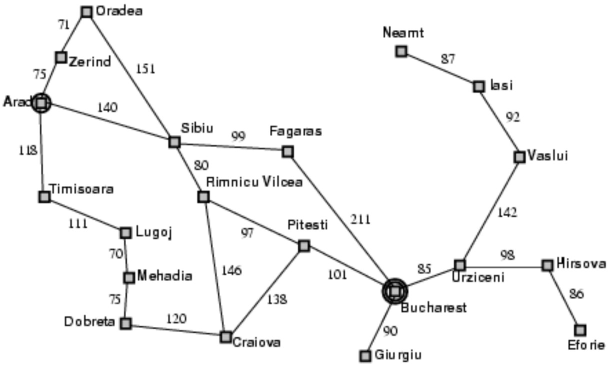
Complete architectures for intelligence?

- Search?
  - Solve the problem of what to do.
- Learning?
  - Learn what to do.
- Logic and inference?
  - Reason about what to do.
  - Encoded knowledge/”expert” systems?
    - Know what to do.
- Modern view: It’s complex & multi-faceted.

Search?
Solve the problem of what to do.

- Formulate “What to do?” as a search problem.
  - Solution to the problem tells agent what to do.
- If no solution in the current search space?
  - Formulate and solve the problem of finding a search space that does contain a solution.
  - Solve original problem in the new search space.
- Many powerful extensions to these ideas.
  - Constraint satisfaction; means-ends analysis; etc.
- Human problem-solving often looks like search.

Problem Formulation

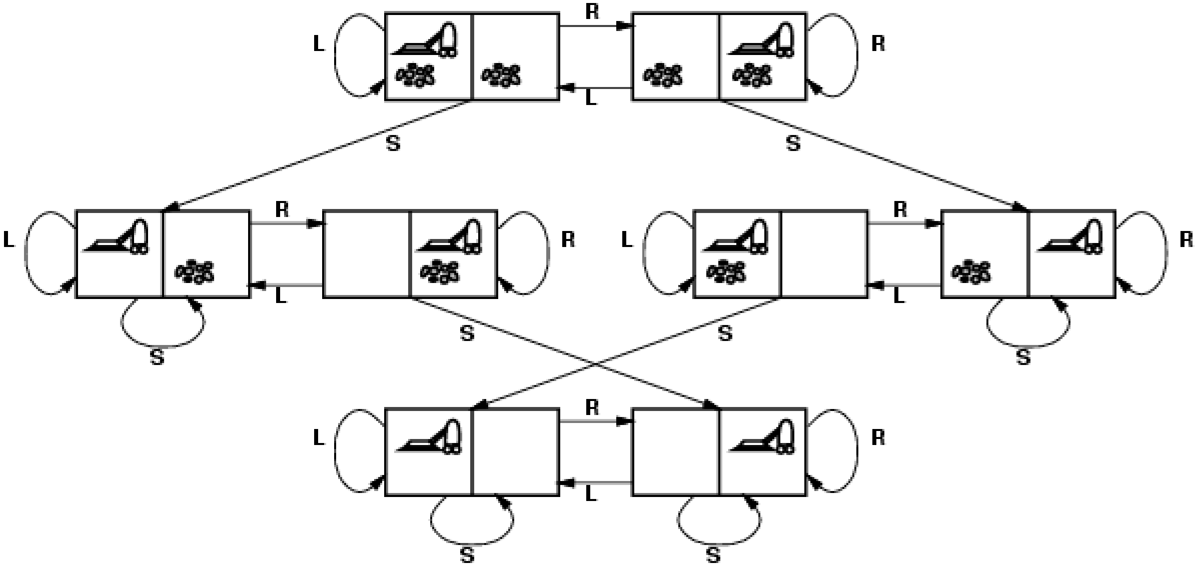


A problem is defined by five items:

- initial state e.g., "at Arad“
- actions
  - Actions(X) = set of actions available in state X
- transition model
  - Result(S,A) = state resulting from doing action A in state S
- goal test, e.g., *x* = "at Bucharest”, *Checkmate(x)*
- path cost (additive, i.e., the sum of the step costs)
  - *c(x,a,y)* = step cost of action *a* in state *x* to reach state *y*
    - assumed to be ≥ 0

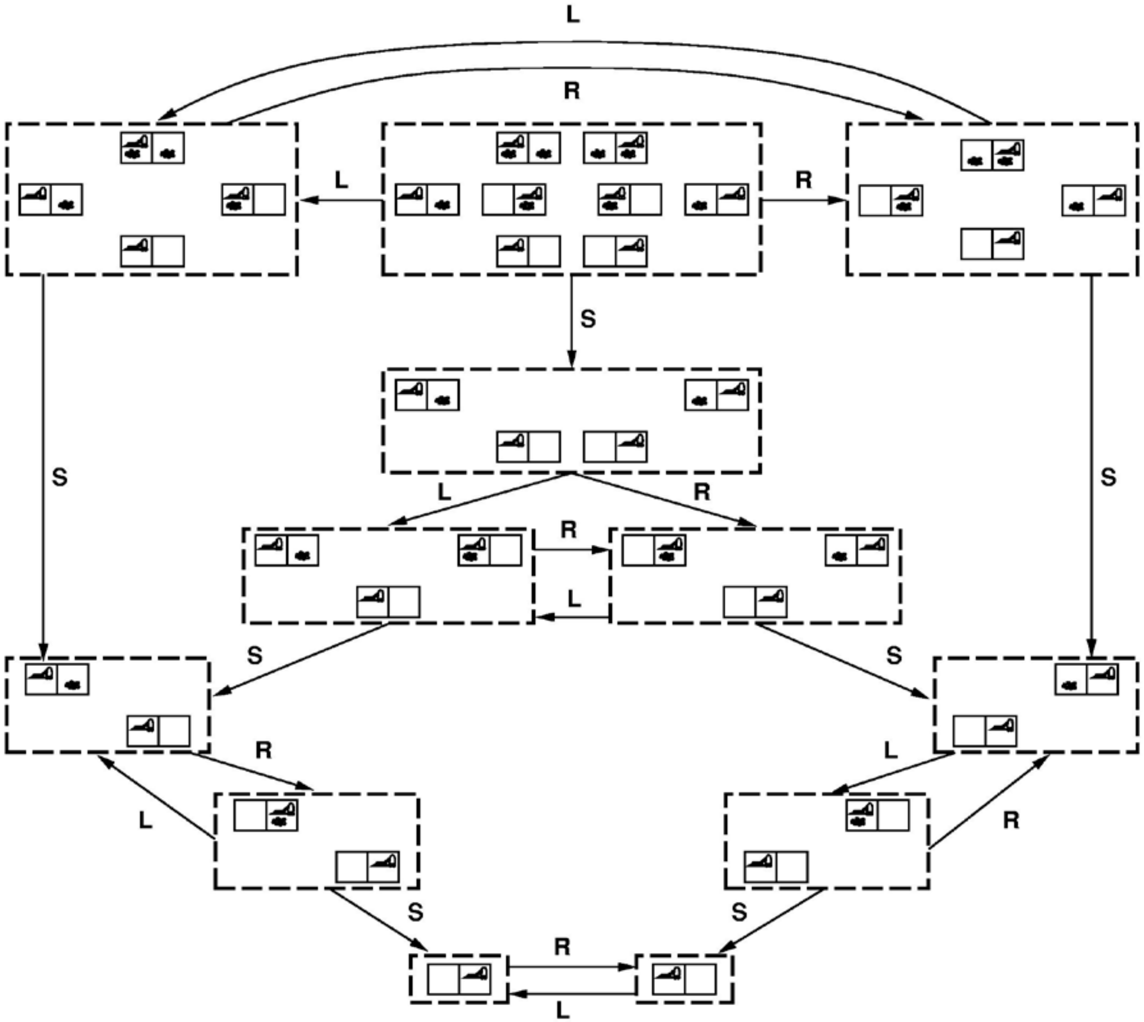
A solution is a sequence of actions leading from the initial state to a goal state

Vacuum world state space graph



- <u>states?</u> discrete: dirt and robot location
- <u>initial state?</u> any
- <u>actions?</u> *Left*, *Right*, *Suck*
    - *<u>Transition Model or Successors as shown on graph</u>*
- <u>goal test?</u> no dirt at all locations
- <u>path cost?</u> 1 per action

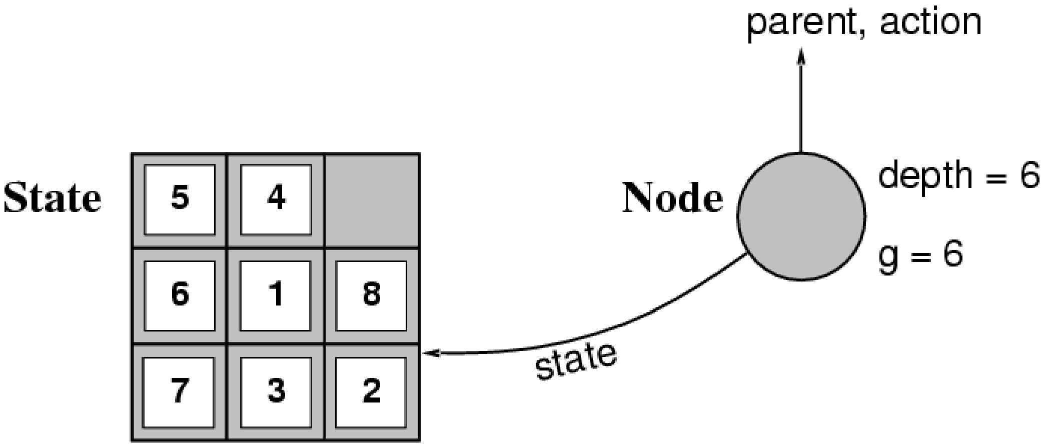
Vacuum world belief states: Agent's belief about what state it's in



Implementation: states vs. nodes

- A state is a (representation of) a physical configuration

- A node is a data structure constituting part of a search tree
- A node contains info such as:
  - state, parent node, action, path cost *g(x)*, depth, etc.

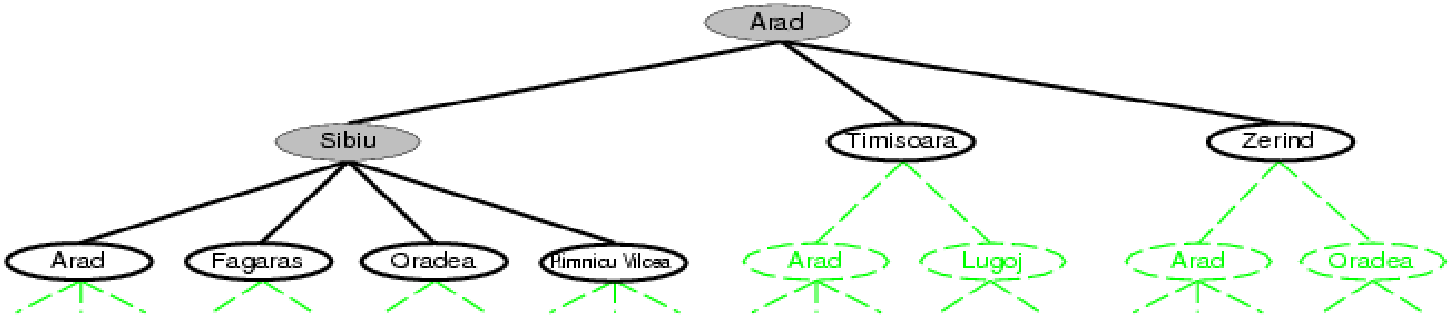


- The `Expand` function creates new nodes, filling in the various fields using the `Actions(S)` and `Result(S,A)` functions associated with the problem.

Tree search algorithms

- Basic idea:
  - Exploration of state space by generating successors of already-explored states (a.k.a.~expanding states).
  - Every generated state is evaluated: *is it a goal state*?

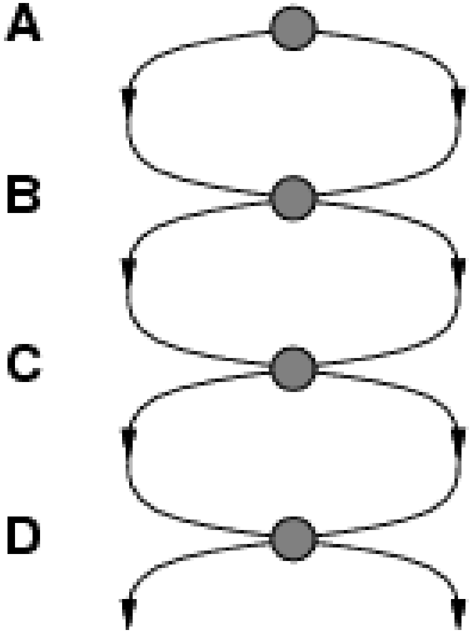
Tree search example

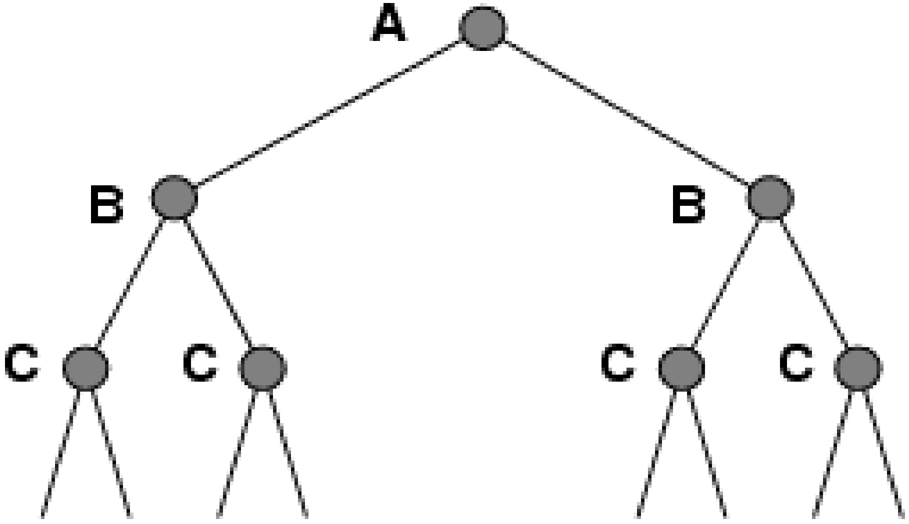


```
function TREE-SEARCH( problem, strategy) returns a solution, or failure
    initialize the search tree using the initial state of problem
    loop do
        if there are no candidates for expansion then return failure
        choose a leaf node for expansion according to strategy
        if the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

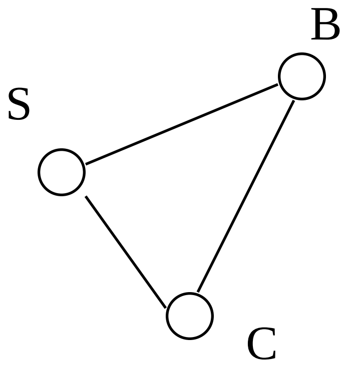
Repeated states

- Failure to detect repeated states can turn a linear problem into an exponential one!
- Test is often implemented as a hash table.

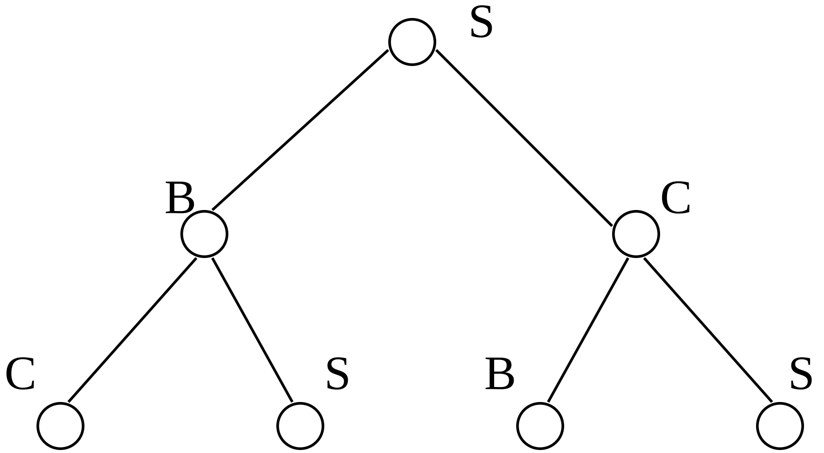




Solutions to Repeated States



State Space



Example of a Search Tree

- Graph search

optimal but memory inefficient

  - never generate a state generated before
    - must keep track of all possible states (uses a lot of memory)
    - e.g., 8-puzzle problem, we have 9! = 362,880 states
    - approximation for DFS/DLS: only avoid states in its (limited) memory: avoid looping paths.
    - Graph search optimal for BFS and UCS, not for DFS.

Search strategies

- A search strategy is defined by 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 ∞)
  - *l*: the depth limit (for Depth-limited complexity)
  - *C**: the cost of the optimal solution (for Uniform-cost complexity)
  - ε: minimum step cost, a positive constant (for Uniform-cost complexity)

Uninformed search strategies

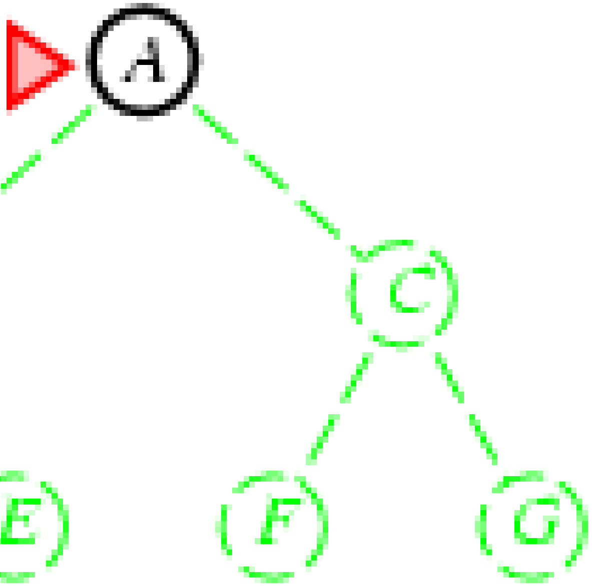
- Uninformed: You have no clue whether one non-goal state is better than any other. Your search is blind. You don't know if your current exploration is likely to be fruitful.
- Various blind strategies:
  - Breadth-first search
  - Uniform-cost search
  - Depth-first search
  - Iterative deepening search (generally preferred)
  - Bidirectional search (preferred if applicable)

Blind Search Strategies (3.4)

- Depth-first: Add successors to front of queue
- Breadth-first: Add successors to back of queue
- Uniform-cost: Sort queue by path cost g(n)
- Depth-limited: Depth-first, cut off at limit *l*
- Iterated-deepening: Depth-limited, increasing *l*
- Bidirectional: Breadth-first from goal, too.

Breadth-first search

- Expand shallowest unexpanded node
- *Frontier* (or fringe): nodes in queue to be explored
- *Frontier* is a first-in-first-out (FIFO) queue, i.e., new successors go at end of the queue.
- *Goal-Test* when inserted.



Is A a goal state?

Properties of breadth-first search

- Complete? Yes it always reaches goal (if b is finite)
- Time? $1+b+b^2+b^3+\ldots +b^d + (b^{d+1}-b)) = O(b^{d+1})$

  (this is the number of nodes we generate)
- Space? $O(b^{d+1})$ (keeps every node in memory,

  either in fringe or on a path to fringe).
- Optimal? Yes (if we guarantee that deeper solutions are less optimal, e.g. step-cost=1).

- Space is the bigger problem (more than time)

Uniform-cost search

Breadth-first is only optimal if path cost is a non-decreasing function of depth, i.e., f(d) ≥ f(d-1); e.g., constant step cost, as in the 8-puzzle.

Can we guarantee optimality for any positive step cost?

Uniform-cost Search:

Expand node with smallest path cost g(n).

- *Frontier* is a priority queue, i.e., new successors are merged into the queue sorted by g(n).
  - Remove successor states already on queue w/higher g(n).
- *Goal-Test* when node is popped off queue.

Uniform-cost search

Implementation: *Frontier* = queue ordered by path cost.
Equivalent to breadth-first if all step costs all equal.

Complete? Yes, if step cost ≥ ε
(otherwise it can get stuck in infinite loops)

Time? # of nodes with *path cost* ≤ cost of optimal solution.

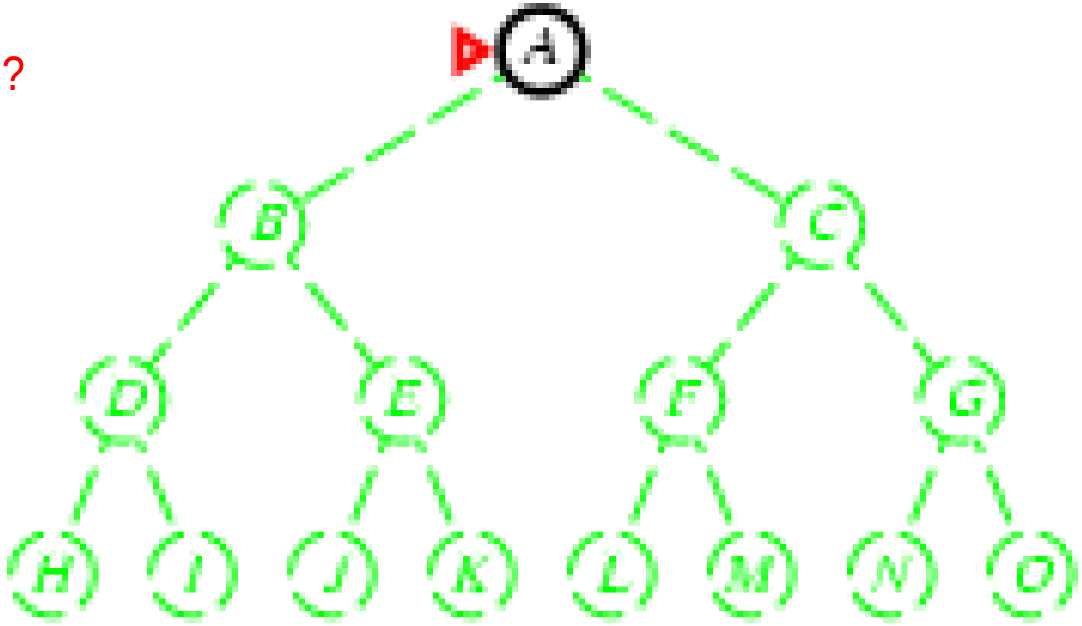
Space? # of nodes with path cost ≤ cost of optimal solution.

Optimal? Yes, for any step cost ≥ ε

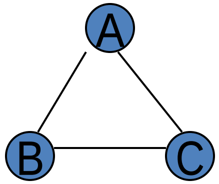
Depth-first search

- Expand *deepest* unexpanded node
- *Frontier* = Last In First Out (LIFO) queue, i.e., new successors go at the front of the queue.
- *Goal-Test* when inserted.

Is A a goal state?



Properties of depth-first search



- Complete? No: fails in infinite-depth spaces

  Can modify to avoid repeated states along path
- Time? $O(b^m)$ with m=maximum depth
- 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! (we only need to remember a single path + expanded unexplored nodes)
- Optimal? No (It may find a non-optimal goal first)

Iterative deepening search

- To avoid the infinite depth problem of DFS, we can decide to only search until depth L, i.e. we don’t expand beyond depth L.
  → Depth-Limited Search

- What if solution is deeper than L? → Increase L iteratively.
  → Iterative Deepening Search

- As we shall see: this inherits the memory advantage of Depth-First search, and is better in terms of time complexity than Breadth first search.

Properties of iterative deepening search

- Complete? Yes
- Time? $O(b^d)$
- Space? $O(bd)$
- Optimal? Yes, if step cost = 1 or increasing function of depth.

Bidirectional Search

- Idea
  - simultaneously search forward from S and backwards from G
  - stop when both “meet in the middle”
  - need to keep track of the intersection of 2 open sets of nodes
- What does searching backwards from G mean
  - need a way to specify the predecessors of G
    - this can be difficult,
    - e.g., predecessors of checkmate in chess?
  - which to take if there are multiple goal states?
  - where to start if there is only a goal test, no explicit list?

Summary of algorithms

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening DLS
Complete?	Yes	Yes	No	No	Yes
Time	$O(b^d)$	$O(b^{C^*/\varepsilon})$	$O(b^m)$	$O(b^l)$	$O(b^d)$
Space	$O(b^d)$	$O(b^{C^*/\varepsilon})$	$O(bm)$	$O(bl)$	$O(bd)$
Optimal?	Yes	Yes	No	No	Yes

Generally the preferred
uninformed search strategy

Best-first search

- Idea: use an evaluation function *f(n)* for each node
  - f(n) provides an estimate for the total cost.
  - → Expand the node n with smallest f(n).
  - g(n) = path cost so far to node n.
  - h(n) = estimate of (optimal) cost to goal from node n.
  - f(n) = g(n)+h(n).
- Implementation:
  Order the nodes in frontier by increasing order of cost.

- Evaluation function is an estimate of node quality

  ⇒More accurate name for "best first" search would be "seemingly best-first search"

⇒***Search efficiency depends on heuristic quality***

Heuristic function

- Heuristic:
  - Definition: a commonsense rule (or set of rules) intended to increase the probability of solving some problem
  - “using rules of thumb to find answers”

- Heuristic function h(n)
  - Estimate of (optimal) cost from n to goal
  - Defined using only the *<u>state</u>* of node *n*
  - h(n) = 0 if n is a goal node
  - Example: straight line distance from n to Bucharest
    - Note that this is not the true state-space distance
    - It is an estimate – actual state-space distance can be higher

  - Provides problem-specific knowledge to the search algorithm

Greedy best-first search

- $h(n)$ = estimate of cost from n to *goal*
  - e.g., $h(n)$ = straight-line distance from n to Bucharest
- Greedy best-first search expands the node that appears to be closest to goal.
  - $f(n) = h(n)$

Properties of greedy best-first search

- Complete?
  - Tree version can get stuck in loops.
  - Graph version is complete in finite spaces.
- Time? $O(b^m)$, but a good heuristic can give dramatic improvement
- Space? $O(b^m)$ - keeps all nodes in memory
- Optimal? No

e.g., Arad → Sibiu → Rimnicu Vilcea → Pitesti → Bucharest is shorter!

A^* search

- Idea: avoid expanding paths 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
- $f(n)$ = estimated total cost of path through n to goal
- Greedy Best First search has $f(n)=h(n)$
- Uniform Cost search has $f(n)=g(n)$

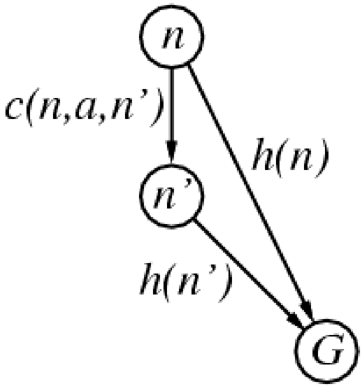
Admissible heuristics

- A heuristic $h(n)$ is admissible if for every node n, $h(n) \leq h^*(n)$, 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

Consistent heuristics (consistent => admissible)

- A heuristic is consistent if for every node *n*, every successor *n'* of *n* generated by any action *a*,

$$\boxed{h(n) \le c(n,a,n') + h(n')}$$



- If *h* is consistent, we have

$f(n') = g(n') + h(n')$ (by def.)
$= g(n) + c(n,a,n') + h(n')$ $(g(n')=g(n)+c(n.a.n'))$
$\ge g(n) + h(n) = f(n)$ (consistency)
$f(n') \ge f(n)$

It's the triangle inequality !

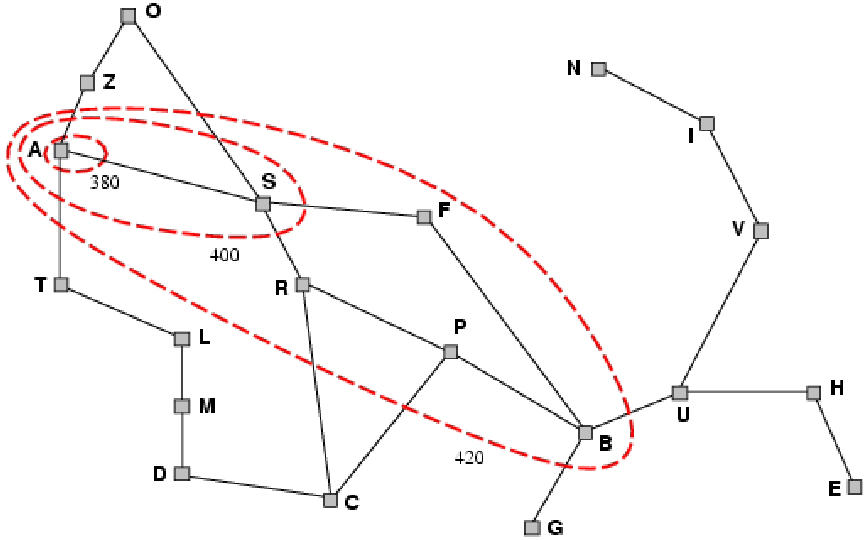
- i.e., *f(n)* is non-decreasing along any path.

- Theorem:
  If *h(n)* is consistent, A* using GRAPH-SEARCH is optimal

keeps all checked nodes in memory to avoid repeated states

Contours of A* Search

- A* expands nodes in order of increasing *f* value
- Gradually adds "*f*-contours" of nodes
- Contour *i* has all nodes with $f=f_i$, where $f_i < f_{i+1}$



Properties of A*

- Complete? Yes (unless there are infinitely many nodes with $f \leq f(G)$, i.e. step-cost > ε)
- Time/Space? Exponential b^d

  except if: $|h(n) - h^*(n)| \leq O(\log h^*(n))$
- Optimal? Yes (with: Tree-Search, admissible heuristic; Graph-Search, consistent heuristic)
- *Optimally Efficient*: Yes (no optimal algorithm with same heuristic is guaranteed to expand fewer nodes)

Simple Memory Bounded A*

- This is like A*, but when memory is full we delete the worst node (largest f-value).
- Like RBFS, we remember the best descendent in the branch we delete.
- If there is a tie (equal f-values) we delete the oldest nodes first.
- simple-MBA* finds the optimal *reachable* solution given the memory constraint.
- Time can still be exponential.

A Solution is not reachable
if a single path from root to goal
does not fit into memory

SMA* pseudocode (not in 2nd edition 2 of book)

```
function SMA*(problem) returns a solution sequence
  inputs: problem, a problem
  static: Queue, a queue of nodes ordered by f-cost

  Queue ← MAKE-QUEUE({MAKE-NODE(INITIAL-STATE[problem])})
  loop do
      if Queue is empty then return failure
      n ← deepest least-f-cost node in Queue
      if GOAL-TEST(n) then return success
      s ← NEXT-SUCCESSOR(n)
      if s is not a goal and is at maximum depth then
         f(s) ← ∞
      else
         f(s) ← MAX(f(n),g(s)+h(s))
      if all of n's successors have been generated then
         update n's f-cost and those of its ancestors if necessary
      if SUCCESSORS(n) all in memory then remove n from Queue
      if memory is full then
         delete shallowest, highest-f-cost node in Queue
         remove it from its parent's successor list
         insert its parent on Queue if necessary
      insert s in Queue
  end
```

Simple Memory-bounded A* (SMA*)

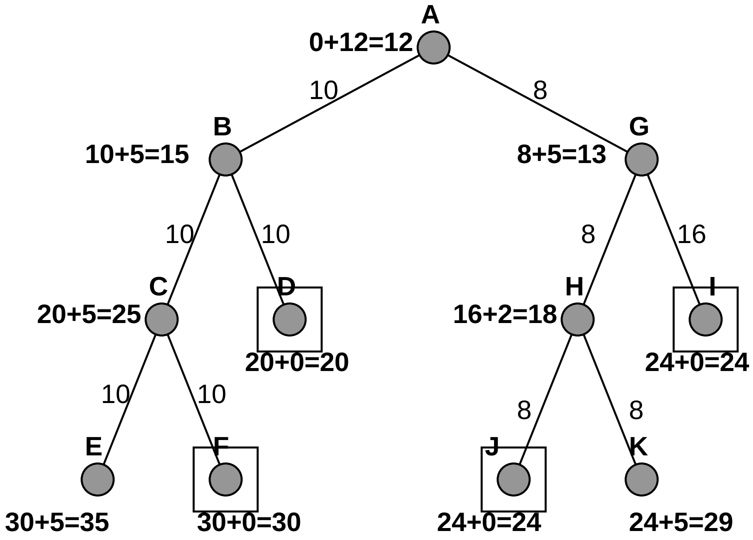
(Example with 3-node memory)

Progress of SMA*. Each node is labeled with its *current* *f*-cost.
Values in parentheses show the value of the best forgotten descendant.

Search space

g+h = f

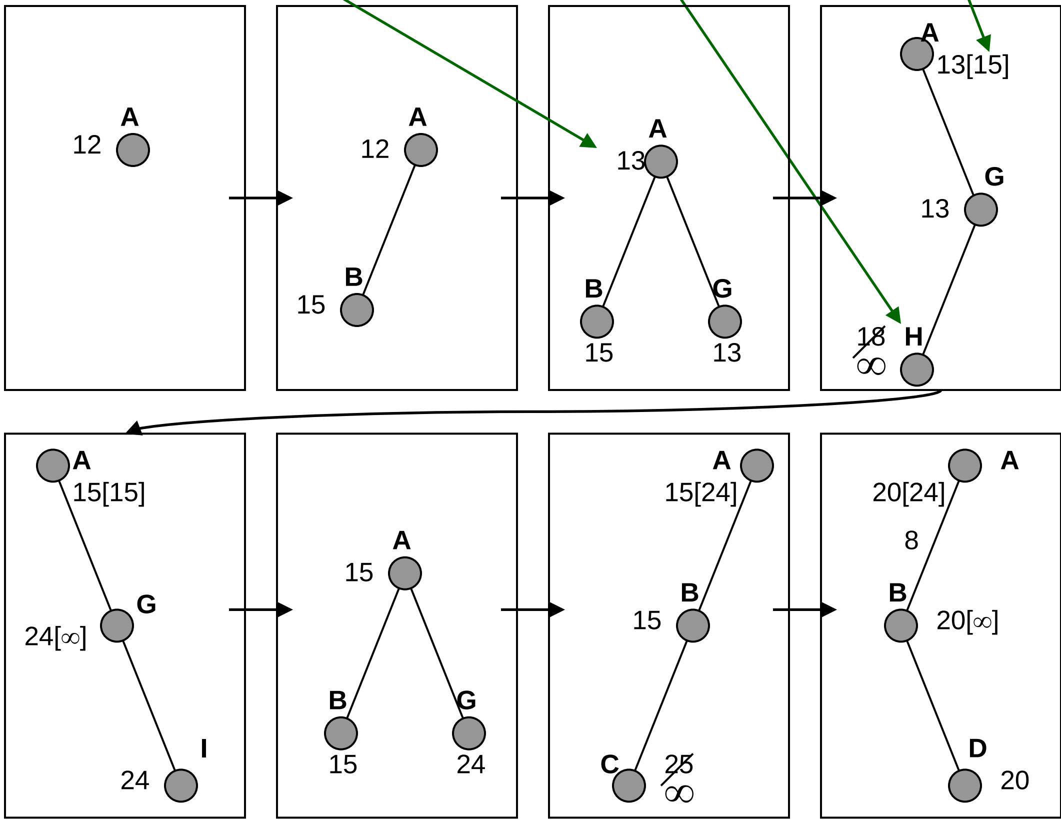
☐ = goal



maximal depth is 3, since memory limit is 3. This branch is now useless.

best forgotten node

best estimated solution so far for that node



Algorithm can tell you when best solution found within memory constraint is optimal or not.

Conclusions

- The Memory Bounded A* Search is the best of the search algorithms we have seen so far. It uses all its memory to avoid double work and uses smart heuristics to first descend into promising branches of the search-tree.

Dominance

- If $h_2(n) \geq h_1(n)$ for all n (both admissible)
- then h_2 dominates h_1
- h_2 is better for search: it is guaranteed to expand less or equal nr of nodes.

- Typical search costs (average number of nodes expanded):

- *d=12* IDS = 3,644,035 nodes
  $A^*(h_1)$ = 227 nodes
  $A^*(h_2)$ = 73 nodes
- *d=24* IDS = too many nodes
  $A^*(h_1)$ = 39,135 nodes
  $A^*(h_2)$ = 1,641 nodes

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

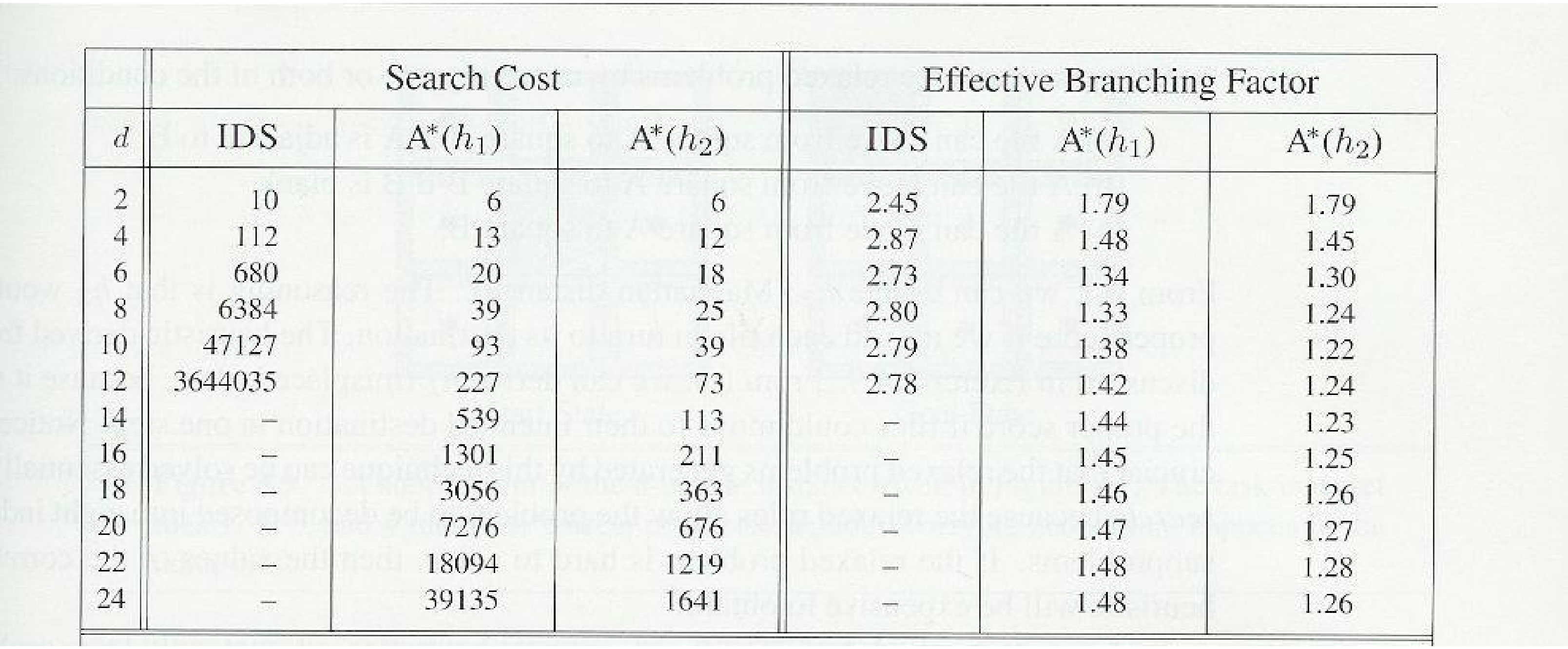
Effective branching factor

- Effective branching factor b*
  - Is the branching factor that a uniform tree of depth *d* would have in order to contain *N+1* nodes.

$$N + 1 = 1 + b^* + (b^*)^2 + \ldots + (b^*)^d$$

  - Measure is fairly constant for sufficiently hard problems.
    - Can thus provide a good guide to the heuristic’s overall usefulness.

Effectiveness of different heuristics



	Search Cost			Effective Branching Factor		
d	IDS	$A^*(h_1)$	$A^*(h_2)$	IDS	$A^*(h_1)$	$A^*(h_2)$
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	3644035	227	73	2.78	1.42	1.24
14	–	539	113	–	1.44	1.23
16	–	1301	211	–	1.45	1.25
18	–	3056	363	–	1.46	1.26
20	–	7276	676	–	1.47	1.27
22	–	18094	1219	–	1.48	1.28
24	–	39135	1641	–	1.48	1.26

- Results averaged over random instances of the 8-puzzle

Inventing heuristics via “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
- Can be a useful way to generate heuristics
  - E.g., ABSOLVER (Prieditis, 1993) discovered the first useful heuristic for the Rubik’s cube puzzle

More on heuristics

- h(n) = max{ h1(n), h2(n),……hk(n) }
  - Assume all h functions are admissible
  - Always choose the least optimistic heuristic (most accurate) at each node

  - Could also learn a convex combination of features
    - Weighted sum of h(n)'s, where weights sum to 1
    - Weights learned via repeated puzzle-solving

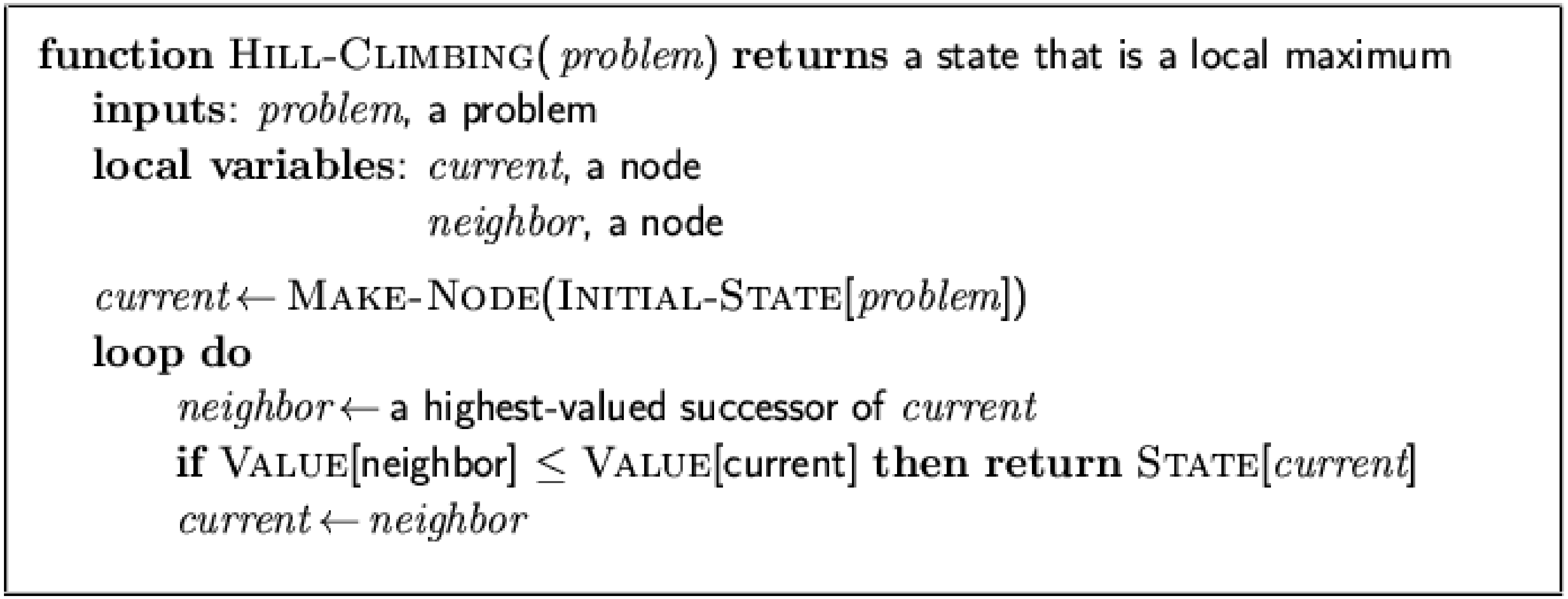
- Could try to learn a heuristic function based on "features"
  - E.g., x1(n) = number of misplaced tiles
  - E.g., x2(n) = number of goal-adjacent-pairs that are currently adjacent
  - h(n) = w_1 x1(n) + w_2 x2(n)
    - Weights could be learned again via repeated puzzle-solving
    - Try to identify which features are predictive of path cost

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
- In such cases, we can use local search algorithms
- keep a single "current" state, try to improve it.
- Very memory efficient (only remember current state)

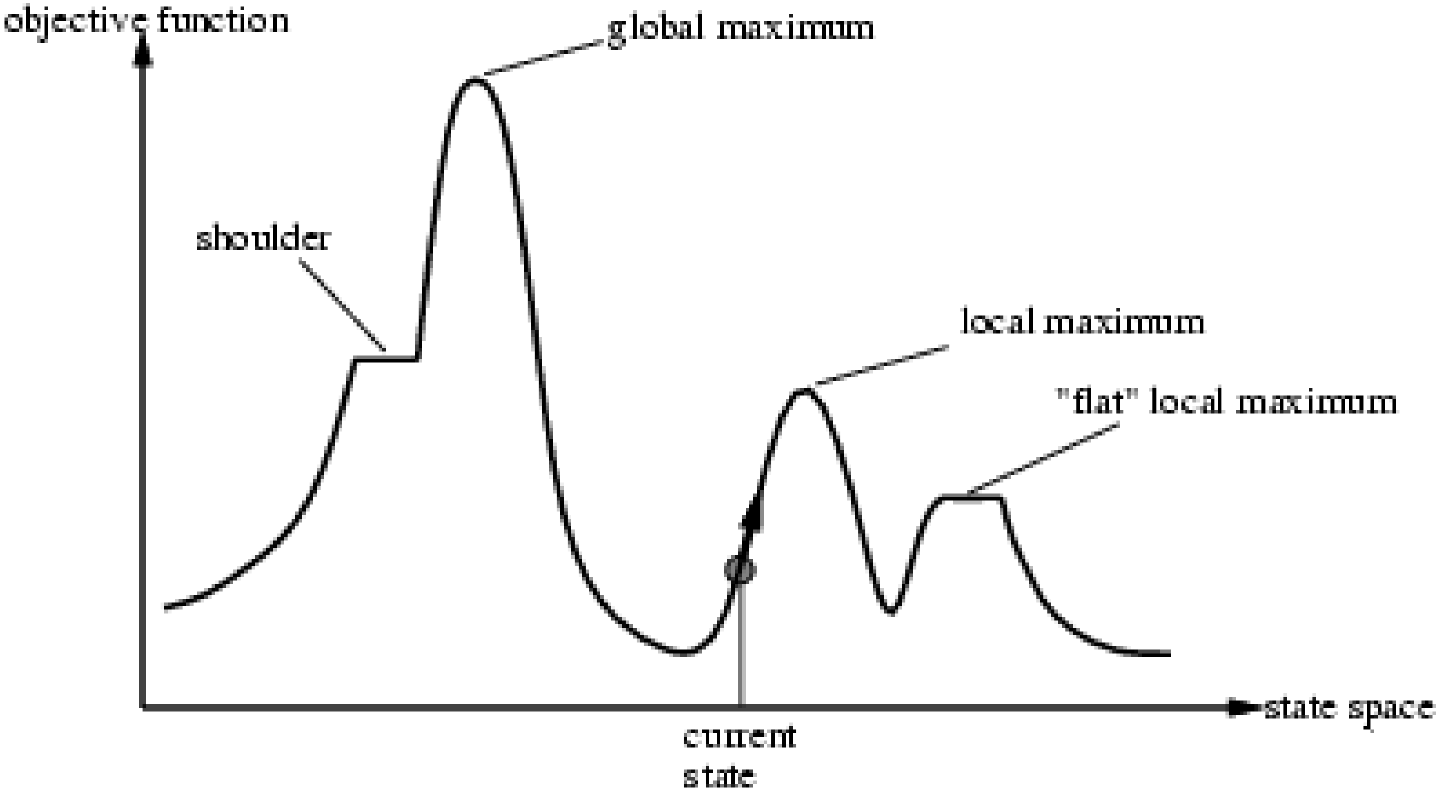
Hill-climbing search

- "Like climbing Everest in thick fog with amnesia"
- 

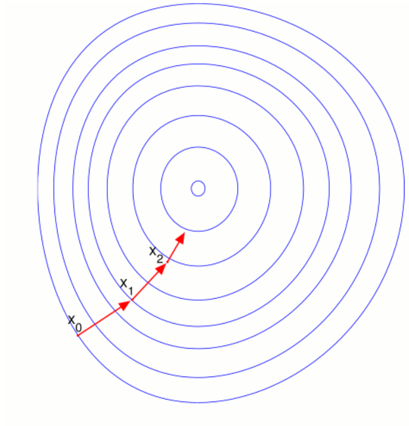
```
function HILL-CLIMBING( problem) returns a state that is a local maximum
    inputs: problem, a problem
    local variables: current, a node
                     neighbor, a node

    current ← MAKE-NODE(INITIAL-STATE[problem])
    loop do
        neighbor ← a highest-valued successor of current
        if VALUE[neighbor] $\leq$ VALUE[current] then return STATE[current]
        current ← neighbor
```

Hill-climbing Difficulties

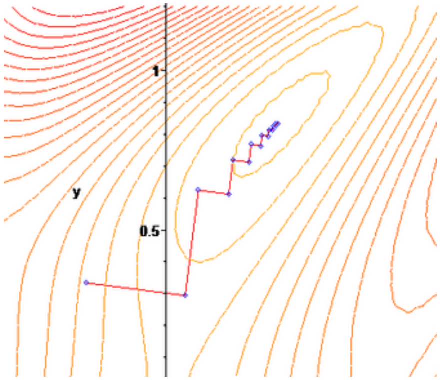
- Problem: depending on initial state, can get stuck in local maxima



Gradient Descent



- Assume we have some cost-function: $C(x_1,\ldots,x_n)$
and we want minimize over continuous variables X1,X2,..,Xn



1. Compute the *gradient :* $\frac{\partial}{\partial x_i} C(x_1,\ldots,x_n) \qquad \forall i$

2. Take a small step downhill in the direction of the gradient:

$$x_i \rightarrow x'_i = x_i - \lambda \frac{\partial}{\partial x_i} C(x_1,\ldots,x_n) \qquad \forall i$$

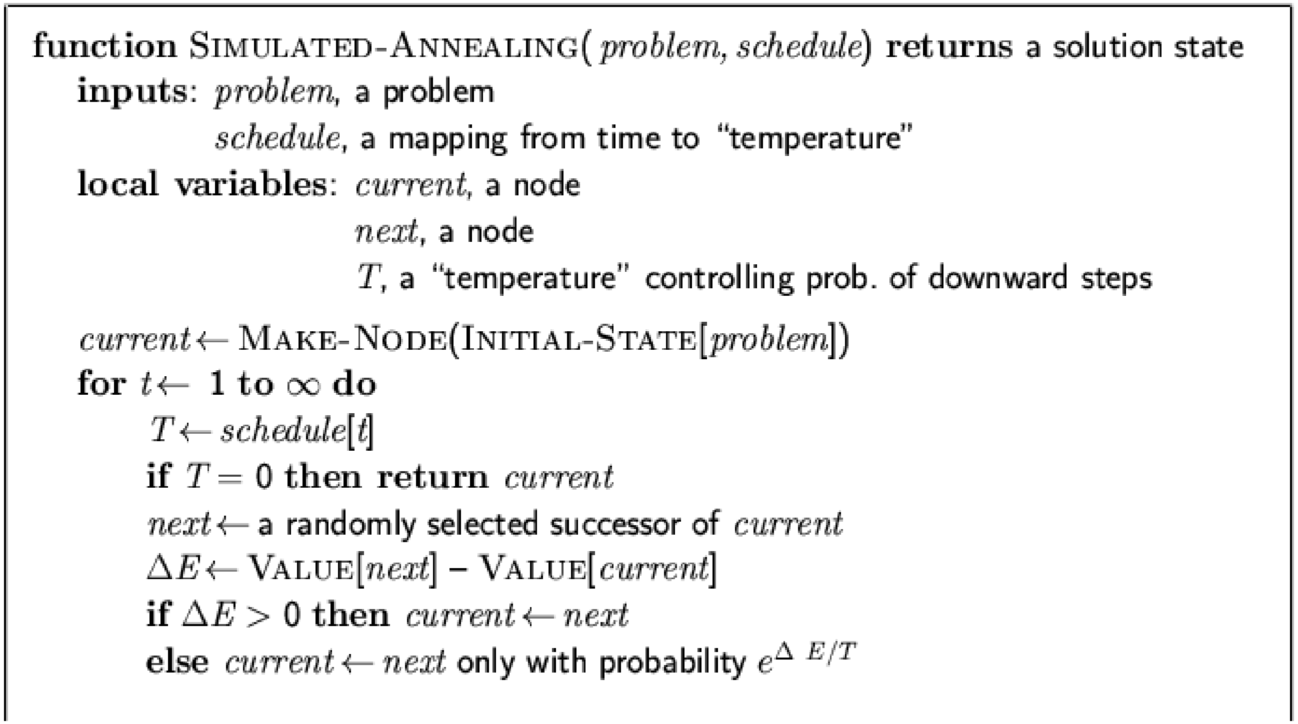
3. Check if $C(x_1,..,x'_i,..,x_n) < C(x_1,..,x_i,..,x_n)$

4. If true then accept move, if not reject.

5. Repeat.

Simulated annealing search

- Idea: escape local maxima by allowing some "bad" moves but gradually decrease their 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
        next ← a randomly selected successor of current
        $\Delta E$ ← VALUE[next] – VALUE[current]
        if $\Delta E > 0$ then current ← next
        else current ← next only with probability $e^{\Delta E/T}$
```

Properties of simulated annealing search

- One can prove: If *T* decreases slowly enough, then simulated annealing search will find a global optimum with probability approaching 1 (however, this may take VERY long)
  - However, in any finite search space RANDOM GUESSING also will find a global optimum with probability approaching 1 .

- Widely used in VLSI layout, airline scheduling, etc.

Tabu Search

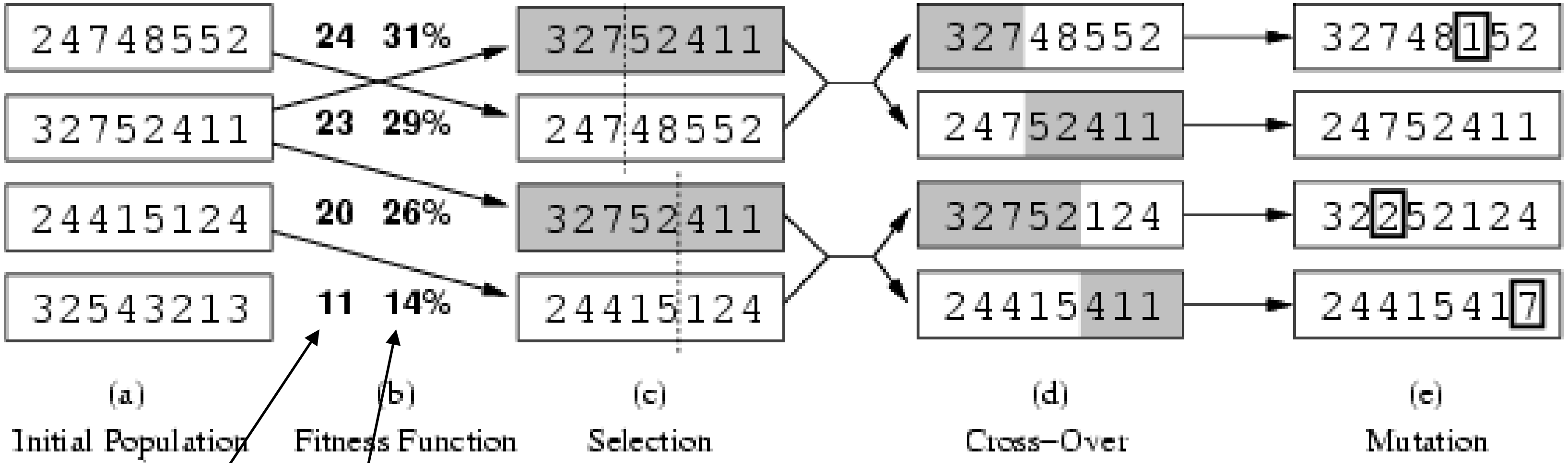
- A simple local search but with a memory.
- Recently visited states are added to a tabu-list and are temporarily excluded from being visited again.
- This way, the solver moves away from already explored regions and (in principle) avoids getting stuck in local minima.

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.
- Concentrates search effort in areas believed to be fruitful.
  - May lose diversity as search progresses, resulting in wasted effort.

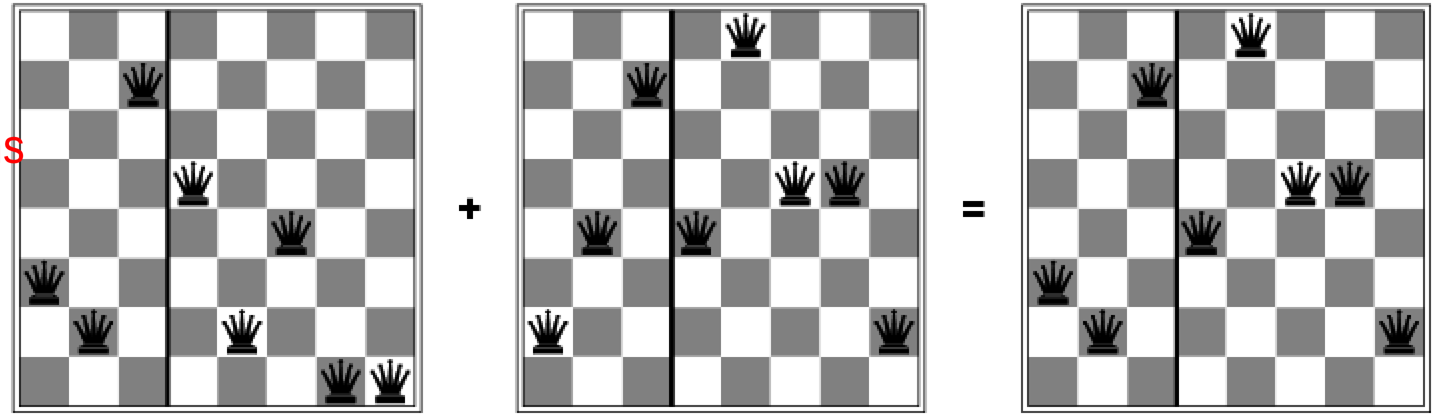
Genetic algorithms

- A successor state is generated by combining two parent states
- 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



fitness:
#non-attacking queens

probability of being
regenerated
in next generation



- Fitness function: number of non-attacking pairs of queens (min = 0, max = 8 7/2 = 28)
- P(child) = 24/(24+23+20+11) = 31%
- P(child) = 23/(24+23+20+11) = 29% etc

Linear Programming

Problems of the sort:

$$\begin{aligned}&\text{maximize} \quad c^T x\\ &\text{subject to}: \ \mathrm{Ax} \le \mathrm{b};\ \mathrm{Bx} = \mathrm{c}\end{aligned}$$

- Very efficient “off-the-shelves” solvers are available for LRs.
- They can solve large problems with thousands of variables.