Informed search algorithms

Chapter 4
Outline

• Best-first search
• Greedy best-first search
• A* search
• Heuristics
• Local search algorithms
• Hill-climbing search
• Simulated annealing search
• Local beam search
• Genetic algorithms
Informed (Heuristic) Search Strategies.

• Use problem–specific knowledge beyond the definition of the problem itself.
• Can fine solutions more efficiently than an uninformed strategy.
Best-first search (BFS)

- An instance of TREE-SEARCH or GRAPH-SEARCH
- Idea:
  - Use an evaluation function \( f(n) \) for each node; estimate of "desirability"
  - \( \Rightarrow \) expand most desirable unexpanded node.
  - \( \Rightarrow \) The node with the lowest evaluation is selected for expansion.
  - \( \Rightarrow \) Measure = distance to goal state.
- Implementation: priority queue.
  - QueueingFn = insert successors in decreasing order of desirability
- Special cases:
  - Greedy search, A* search,
BFS

• Best ≠ best path to goal.
• Best = Appears to be the best according to the evaluation function.
• If $f(n)$ is accurate, then OK. ($f(n) = ??$)
• True meaning: “seemingly-best-first search”
• Greedy method.
• Heuristic function $h(n)$:
  – estimated cost of the cheapest path from node $n$ to a goal node.
  – If $n$ is a goal node, then $h(n)=0$. 
Greedy best-first search

• Tried to expand the node that is closet to the goal.
• Let $f(n)=h(n)$.
• Greedy: at each step it tries to get as close to the goal as it can.
# Straight-line distance

<table>
<thead>
<tr>
<th>Location</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arad</td>
<td>366</td>
</tr>
<tr>
<td>Bucharest</td>
<td>0</td>
</tr>
<tr>
<td>Craiova</td>
<td>160</td>
</tr>
<tr>
<td>Dobreta</td>
<td>242</td>
</tr>
<tr>
<td>Eforie</td>
<td>161</td>
</tr>
<tr>
<td>Fagaras</td>
<td>176</td>
</tr>
<tr>
<td>Giurgiu</td>
<td>77</td>
</tr>
<tr>
<td>Hirsova</td>
<td>151</td>
</tr>
<tr>
<td>Iasi</td>
<td>226</td>
</tr>
<tr>
<td>Lugoj</td>
<td>244</td>
</tr>
<tr>
<td>Mehadia</td>
<td>241</td>
</tr>
<tr>
<td>Neamt</td>
<td>234</td>
</tr>
<tr>
<td>Oradea</td>
<td>380</td>
</tr>
<tr>
<td>Pitesti</td>
<td>100</td>
</tr>
<tr>
<td>Rimnicu Vilcea</td>
<td>193</td>
</tr>
<tr>
<td>Sibiu</td>
<td>253</td>
</tr>
<tr>
<td>Timisoara</td>
<td>329</td>
</tr>
<tr>
<td>Urziceni</td>
<td>80</td>
</tr>
<tr>
<td>Vaslui</td>
<td>199</td>
</tr>
<tr>
<td>Zerind</td>
<td>374</td>
</tr>
</tbody>
</table>

**Figure 4.1** Values of $h_{SLD}$—straight-line distances to Bucharest.
Romania with step costs in km

Straight-line distance to Bucharest

- Arad: 366
- Bucharest: 0
- Craiova: 160
- Dobreta: 242
- Eforie: 161
- Fagaras: 178
- Giurgiu: 77
- Hirsova: 151
- Iasi: 226
- Lugoj: 244
- Mehadia: 241
- Neamt: 234
- Oradea: 380
- Pitești: 98
- Rimnicu Vilcea: 193
- Sibiu: 253
- Timisoara: 329
- Urziceni: 80
- Vaslui: 199
- Zerind: 374
Greedy search

- Estimation function:
  \[ h(n) = \text{estimate of cost from } n \text{ to goal} \]
  (heuristic)

- For example:
  \[ h_{SLD}(n) = \text{straight-line distance from } n \text{ to Buchares} \]

- Greedy search expands first the node that \textbf{appears} to be closest to the goal, according to \( h(n) \).
Figure 4.2  Stages in a greedy best-first search for Bucharest using the straight-line distance heuristic $h_{SLD}$. Nodes are labeled with their $h$-values.
Greedy best-first search example
Greedy best-first search example
Greedy best-first search example
Greedy best-first search example
Properties of greedy best-first search

- **Complete?** No – can get stuck in loops, e.g., Iasi → Neamt → Iasi → Neamt →
  - Susceptible to false starts. (may be no solution)
  - May cause unnecessary nodes to expanded
  - Stuck in loop. (Incomplete)
- **Time?** $O(b^m)$, but a good heuristic can give dramatic improvement
- **Space?** $O(b^m)$ -- keeps all nodes in memory
- **Optimal?** No
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 the cheapest cost from $n$ to goal
• $f(n)$ = estimated total cost of path through $n$ to goal
• Both complete and optimal
A* search example
A* search example
A* search example
A* search example
A* search example
A* search example
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**.
Optimality of A* (TREE-search)

• Suppose some suboptimal goal $G_2$ has been generated and is in the fringe. Let $n$ be an unexpanded node in the fringe such that $n$ is on a shortest path to an optimal goal $G$.

$G_2$ and $n$ in fringe

• $f(G_2) = g(G_2) + h(G_2) = g(G_2) > C^*$ since $h(G_2) = 0$
• $g(G_2) > g(G)$ since $G_2$ is suboptimal
• If $h(n)$ does not overestimate the cost of completing the solution path ($h(n) \leq h^*(n)$)

$G_2$ and $n$ in fringe

• $f(n) = g(n) + h(n) \leq g(n) + h^*(n) \leq C^*$
• $f(n) \leq C^* < f(G_2)$
• So, $G_2$ will not be expanded and A* must return an optimal solution.
Optimality of A* (proof)

• Suppose some suboptimal goal $G_2$ has been generated and is in the fringe. Let $n$ be an unexpanded node in the fringe such that $n$ is on a shortest path to an optimal goal $G$.

• $f(G_2) > f(G)$ from above
• $h(n) \leq h^*(n)$ since $h$ is admissible
• $g(n) + h(n) \leq g(n) + h^*(n)$
• $f(n) \leq f(G)$

Hence $f(G_2) > f(n)$, and A* will never select $G_2$ for expansion
Consistency (monotonicity)

heuristics

• A heuristic $h(n)$ 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

\[
\begin{align*}
  f(n') &= g(n') + h(n') \\
  &= g(n) + c(n,a,n') + h(n') \\
  &\geq g(n) + h(n) \\
  &= f(n)
\end{align*}
\]

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

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

Triangle inequality
Optimality of A*

- 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*

- A* expands all nodes with $f(n) < C^*$
- A* might then expand some of the nodes right on the “goal contour” ($f(n) = C^*$) before selecting a goal state.
- The solution found must be an optimal one.
Properties of A*

- **Complete?** Yes (unless there are infinitely many nodes with \( f \leq f(G) \))
- **Time?** Exponential
- **Space?** Keeps all nodes in memory, before finding solution it may run out of the memory.
- **Optimal?** Yes
- **Optimal Efficient:** for any given heuristic function, on other optimal algorithm is guaranteed to expand fewer nodes than A*. Since A* expand no nodes with \( f(n) > C^* \).
Memory-Bounded heuristic search

• Reduced memory requirement of A*.
• Iterative-deepening + A* = IDA*.
• The cutoff value used is the f-cost (g+h) rather than the depth.
function IDA*(problem) returns a solution sequence

inputs: problem, a problem
static: f-limit, the current f- COST limit
root, a node

root ← MAKE-NODE(INITIAL-STATE[problem])
f-limit ← f- COST(root)
loop do
    solution, f-limit ← DFS-CONTOUR(root, f-limit)
    if solution is non-null then return solution
    if f-limit = ∞ then return failure; end

function DFS-CONTOUR(node, f-limit) returns a solution sequence and a new f- COST limit

inputs: node, a node
         f-limit, the current f- COST limit
static: next-f, the f- COST limit for the next contour, initially ∞

if f- COST[node] > f-limit then return null, f- COST[node]
if GOAL-TEST[problem](STATE[node]) then return node, f-limit
for each node s in SUCCESSORS(node) do
    solution, new-f ← DFS-CONTOUR(s, f-limit)
    if solution is non-null then return solution, f-limit
     next-f ← MIN(next-f, new-f); end
return null, next-f

Figure 4.10 The IDA* (Iterative Deepening A*) search algorithm.
RBFS

- Recursive best-first search.
- Mimic the operation of standard DFS using linear space.
- Like recursive DFS
  - It keeps track of the f-value of the **best alternate path** available from any ancestor of the current node.
  - If the current node exceeds this limit, the recursion unwinds back to the alternate path.
  - RBFS remembers the f-values of the best leaf in the forgotten subtree.
**RBFS**

function **Recursive-Best-First-Search**(problem) returns a solution, or failure

RBFS(problem, MAKE-NODE(INITIAL-STATE[problem]), ∞)

function RBFS(problem, node, f_limit) returns a solution, or failure and a new \( f \)-cost limit

if GOAL-TEST[problem](STATE[node]) then return node

successors ← EXPAND(node, problem)

if successors is empty then return failure, ∞

for each \( s \) in successors do

\[ f[s] ← \max(g(s) + h(s), f[node]) \]

repeat

best ← the lowest \( f \)-value node in successors

if \( f[best] > f_{limit} \) then return failure, \( f[best] \)

alternative ← the second-lowest \( f \)-value among successors

result, \( f[best] ← \text{RBFS}(\text{problem}, \text{best}, \min(f_{limit}, \text{alternative})) \)

if result ≠ failure then return result
RBFS

- Efficient than IDA*
- Suffer from excessive node regeneration.
- Optimal: if $h(n)$ is admissible.
- Space $O(bd)$
- Time complexity: hard to characterize.
- Can’t check the repeated states.
- If more memory were available, RBFS has no way to make use of it.
- MA* (memory-bounded A*) and SMA* (simplified MA*)
SMA*

- Simplified memory-bounded A*
- Proceeds just like A*, expanding the best leaf until memory is full.
- It cannot add a new node to the search tree without dropping and old one.
- SMA* always drops the worst leaf node (highest f-value).
- Like RBFS, SMA* then backs up the value of the forgotten node to its parent.
- If new node does not fit
  - free() stored node with worst f-value
  - propagate f-value of freed node to parent
- SMA* will regenerate a subtree only when it is needed
  - the path through deleted subtree is unknown, but cost is known
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 on Queue
end
Memory = 3

Figure 4.11 Progress of an SMA* search with a memory size of three nodes, on the state space shown at the top. Each node is labelled with its current $f$-cost. Values in parentheses show the value of the best forgotten descendant.
SMA*

- Complete
- Optimal.
- SMA* might well be the best general-purpose algorithm for finding optimal solution, particular for
  - Graph search
  - Nonuniform cost
  - Node generation is expansive.
- Time: memory limitations can make a problem intractable from the point of view of computation time.
4.2 Admissible heuristics

E.g., for the 8-puzzle:
Average solution cost  \(~22\) steps
Branching factor =3
Space: \(3^{22} = 3.1 \times 10^{10}\)

- \(h_1(n)\) = number of misplaced tiles
- \(h_2(n)\) = total Manhattan distance
(i.e., no. of squares from desired location of each tile)

- \(h_1(S) = ?\)
- \(h_2(S) = ?\)
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)

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

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

**Figure 4.8** Comparison of the search costs and effective branching factors for the \textsc{Iterative-Deepening-Search} and \(A^*\) algorithms with \(h_1, h_2\). Data are averaged over 100 instances of the 8-puzzle, for various solution lengths.
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 proble.
• 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.
ABSOLVER

• Generate heuristic automatic from problem definition.
• Generate a new heuristic for 8-puzzle problem better than any-existing heuristic.
• Found the first useful heuristic for the famous Rubik’s cube puzzle (魔術方塊).
Combination of heuristics

One problem with generating new heuristic functions is that one often fails to get one "clearly best" heuristic. If a collection of admissible heuristics $h_1 \ldots h_m$ is available for a problem, and none of them dominates any of the others, which should we choose? As it turns out, we need not make a choice. We can have the best of all worlds, by defining

$$h(n) = \max\{h_1(n), \ldots, h_m(n)\}.$$
Drive from subproblem

![Start State and Goal State of the 8-puzzle](image)

**Figure 4.9** A subproblem of the 8-puzzle instance given in Figure 4.7. The task is to get tiles 1, 2, 3, and 4 into their correct positions, without worrying about what happens to the other tiles.

The optimal solution of the subproblem is a lower bound on the cost of the complete problem.
Learning heuristic from experience

- Pattern database
- Inducting learning
- feature
4.3 Local search algorithm and optimization problem.
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.,
  (1) find optimal configuration (e.g., TSP), or,
  (2) find configuration satisfying constraints (n-queens).

- In such cases, we can use local search algorithms.
- Keep a single "current" state, try to improve it.
Iterative improvement

- Optimization problem.
- Objective function.
- In such cases, can use iterative improvement algorithms: keep a single “current” state, and try to improve it.
Example: \( n \)-queens

- Put \( n \) queens on an \( n \times n \) board with no two queens on the same row, column, or diagonal.
- Complete configuration (states)
Hill-climbing search

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

• "Like climbing Everest in thick fog with amnesia"

```plaintext
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
```
H = number of pairs of queens that are attacking each other

Figure 4.12  (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.
Local Minima Problem

• Question: How do you avoid this local minima?
Consequences of the Occasional Ascents

desired effect

Help escaping the local optima.

adverse effect

Might pass global optima after reaching it

(easy to avoid by keeping track of best-ever state)
Hill-climbing search: 8-queens problem

- $h =$ number of pairs of queens that are attacking each other, either directly or indirectly
- $h = 17$ for the above state
Hill-climbing search: 8-queens problem

- A local minimum with $h = 1$
Problem??

◊ **Local maxima**: a local maximum is a peak that is higher than each of its neighboring states, but lower than the global maximum. Hill-climbing algorithms that reach the vicinity of a local maximum will be drawn upwards towards the peak, but will then be stuck with nowhere else to go. Figure 4.10 illustrates the problem schematically. More concretely, the state in Figure 4.12(b) is in fact a local maximum (i.e., a local minimum for the cost $h$); every move of a single queen makes the situation worse.

◊ **Ridges**: a ridge is shown in Figure 4.13. Ridges result in a sequence of local maxima that is very difficult for greedy algorithms to navigate.

◊ **Plateaux**: a plateau is an area of the state space landscape where the evaluation function is flat. It can be a flat local maximum, from which no uphill exit exists, or a **shoulder**, from which it is possible to make progress. (See Figure 4.10.) A hill-climbing search might be unable to find its way off the plateau.
Figure 4.13  Illustration of why ridges cause difficulties for hill-climbing. The grid of states (dark circles) is superimposed on a ridge rising from left to right, creating a sequence of local maxima that are not directly connected to each other. From each local maximum, all the available actions point downhill.
Modify Hill-Climbing

- Sideway move
- Stochastic hill climbing
- First-choice hill climbing.
- Random-restart hill climbing
Boltzmann machines


To motivate their solution, consider how one might get a ball-bearing traveling along the curve to "probably end up" in the deepest minimum. The idea is to shake the box "about $h$ hard" — then the ball is more likely to go from $D$ to $C$ than from $C$ to $D$. So, on average, the ball should end up in $C$'s valley.
Simulated annealing: basic idea

• From current state, pick a random successor state;

• If it has better value than current state, then “accept the transition,” that is, use successor state as current state;

• Otherwise, do not give up, but instead flip a coin and accept the transition with a given probability (that is lower as the successor is worse).

• So we accept to sometimes “un-optimize” the value function a little with a non-zero probability.
Boltzmann’s statistical theory of gases

• In the statistical theory of gases, the gas is described not by a deterministic dynamics, but rather by the probability that it will be in different states.

• The 19th century physicist Ludwig Boltzmann developed a theory that included a probability distribution of temperature (i.e., every small region of the gas had the same kinetic energy).

• Hinton, Sejnowski and Ackley’s idea was that this distribution might also be used to describe neural interactions, where low temperature $T$ is replaced by a small noise term $T$ (the neural analog of random thermal motion of molecules). While their results primarily concern optimization using neural networks, the idea is more general.
Boltzmann distribution

- At thermal equilibrium at temperature T, the Boltzmann distribution gives the relative probability that the system will occupy state A vs. state B as:

\[
\frac{P(A)}{P(B)} = \exp\left( - \frac{E(A) - E(B)}{T} \right) = \frac{\exp\left( \frac{E(B)}{T} \right)}{\exp\left( \frac{E(A)}{T} \right)}
\]

- where E(A) and E(B) are the energies associated with states A and B.
Simulated annealing

Kirkpatrick et al. 1983:

- **Simulated annealing** is a general method for making likely the escape from local minima by allowing jumps to higher energy states.

- The analogy here is with the process of annealing used by a craftsman in forging a sword from an alloy.

- He heats the metal, then slowly cools it as he hammers the blade into shape.
  - If he cools the blade too quickly the metal will form patches of different composition;
  - If the metal is cooled slowly while it is shaped, the constituent metals will form a uniform alloy.
Real annealing: Sword

- He heats the metal, then slowly cools it as he hammers the blade into shape.
  - If he cools the blade too quickly the metal will form patches of different composition;
  - If the metal is cooled slowly while it is shaped, the constituent metals will form a uniform alloy.
Simulated annealing in practice
- set $T$
- optimize for given $T$
- lower $T$
  - (see Geman & Geman, 1984)
- repeat
Simulated annealing in practice

- set T
- optimize for given T
- lower T
- repeat
Simulated annealing in practice

- set T
- optimize for given T
- lower T (see Geman & Geman, 1984)
- repeat

• Geman & Geman (1984): if T is lowered sufficiently slowly (with respect to the number of iterations used to optimize at a given T), simulated annealing is guaranteed to find the global minimum.

• Caveat: this algorithm has no end (Geman & Geman’s T decrease schedule is in the 1/log of the number of iterations, so, T will never reach zero), so it may take an infinite amount of time for it to find the global minimum.
Simulated annealing algorithm

• Idea: Escape local extrema by allowing "bad moves," but gradually decrease their size and frequency.

Note: goal here is to maximize $E$. 

```plaintext
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 the probability of downward steps

    current ← Make-Node(Initial-State[problem])
    for $t$ ← 1 to $\infty$ 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 $e^{\Delta E/T}$
```
Note on simulated annealing: limit cases

- **Boltzmann distribution**: accept “bad move” with $\Delta E < 0$ (goal is to maximize $E$) with probability $P(\Delta E) = \exp(\Delta E / T)$

  - If $T$ is large: $\Delta E < 0$
    $\Delta E / T < 0$ and very small
    $\exp(\Delta E / T)$ close to 1
    accept bad move with **high** probability

  - If $T$ is near 0: $\Delta E < 0$
    $\Delta E / T < 0$ and very large
    $\exp(\Delta E / T)$ close to 0
    accept bad move with **low** probability
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.

• Widely used in VLSI layout, airline scheduling, etc.
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.
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
Genetic algorithms

- Fitness function: number of non-attacking pairs of queens (min = 0, max = 8 ⌊7/2 = 28)
- $24/(24+23+20+11) = 31\%$
- $23/(24+23+20+11) = 29\%$ etc
Genetic algorithms
function GENETIC-ALGORITHM(population, FITNESS-FN) returns an individual

inputs: population, a set of individuals
        FITNESS-FN, a function that measures the fitness of an individual

repeat
    new_population ← empty set
    loop for i from 1 to SIZE(population) do
        x ← RANDOM-SELECTION(population, FITNESS-FN)
        y ← RANDOM-SELECTION(population, FITNESS-FN)
        child ← REPRODUCE(x, y)
        if (small random probability) then child ← MUTATE(child)
        add child to new_population
    population ← new_population
    until some individual is fit enough, or enough time has elapsed
return the best individual in population, according to FITNESS-FN

function REPRODUCE(x, y) returns an individual

inputs: x, y, parent individuals

n ← LENGTH(x)

\[ c \leftarrow \text{random number from 1 to n} \]

return APPEND(SUBSTRING(x, 1, c), SUBSTRING(y, c + 1, n))

Figure 4.17  A genetic algorithm. The algorithm is the same as the one diagrammed in Figure 4.15, with one variation: in this more popular version, each mating of two parents produces only one offspring, not two.