Backtracking = exhaustive search + pruning

Example: SAT via Backtracking

Let \( \phi = (w \lor x \lor y \lor z) \land (w \lor \overline{x}) \land (x \lor \overline{y}) \land (y \lor \overline{z}) \land (z \lor \overline{w}) \land (\overline{w} \lor \overline{z}) \).

So the problem you want to solve is NP-Complete. Now what?

- ✗ Give up.
- ✗ Burn cycles and try to solve it exactly.
- ✗ Try the first thing that comes into your head and hope it produces correct answers and is fast enough to get by.
- ✓ Open a different tool box. (Chapter 9 of DPV.)
Backtracking: The general scheme

First we need a fast test for subproblems such that

\[
\text{test}(P) = \begin{cases} 
\text{failure,} & \text{if subproblem } P \text{ has no solution;} \\
\text{success,} & \text{if a solution to } P \text{ is found;} \\
\text{uncertainty,} & \text{otherwise.}
\end{cases}
\]

Then:

Start with some problem \( P_0 \)

\( S \leftarrow \{ P_0 \} \) // the set of active subproblems

while \( S \neq \emptyset \) do

Choose a \( P \in S \); \( S \leftarrow S - \{ P \} \)

Expand \( P \) into subproblems \( P_1, \ldots, P_k \)

for \( i \leftarrow 1 \) to \( k \) do

\[
\text{case test}(P_i) \text{ of}
\]

success: announce solution and halt

failure: discard \( P_i \)

uncertainty: add \( P_i \) to \( S \)

Announce that there is no solution.

Branch-and-Bound

- \( \text{B&B} \) = the backtracking idea for optimization problems
- We consider minimization problems.
- First we need a fast way to compute lower bounds for the cost.
- Then:

Start with some problem \( P_0 \)

\( S \leftarrow \{ P_0 \} \) // the set of active subproblems

\( \text{bestSoFar} \leftarrow \infty \)

while \( S \neq \emptyset \) do

Choose a \( P \in S \); \( S \leftarrow S - \{ P \} \)

Expand \( P \) into subproblems \( P_1, \ldots, P_k \)

for \( i \leftarrow 1 \) to \( k \) do

\[
\text{if } (P_i \text{ is a complete solution) then update } \text{bestSoFar}
\]

\[
\text{else if } (\text{lowerbound}(P_i) < \text{bestSoFar}) \text{ then add } P_i \text{ to } S
\]

return \( \text{bestSoFar} \)

Branch-and-Bound Applied to TSP, 1

- \( G = (V, E) \) each \( e \in E \) with length \( d_e > 0 \).
- Fix an \( a \in V \).
- Partial solution: \( [a, S, b] \) = a path from \( a \) to \( b \), \( S \) = the verts in this path
- Extension: \( [a, S \cup \{ x \}, x] \) where \( x \in (V - S) \) and \( (b, x) \in E \).
- \( \text{lowerbound}([a, S, b]) \) = the sum of:
  - the cheapest edge from \( a \) to \( V - S \).
  - the cheapest edge from \( b \) to \( V - S \).
  - the cost of a minimum spanning tree of \( V - S \).

?? Why is this a lower bound on the cost?

Branch-and-Bound Applied to TSP, 2

- 28 partial solutions examined.
- \( 7! = 5,040 \) partial solutions in a brute-force search.
Approximation Algorithms

- Instead of seeking an optimum solution, try “close to optimum”
- The question is how close is good enough.
- $\text{opt}(I)$ = the value of an optimum solution for instance $I$.
- **Convention**: Assume $\text{opt}(I)$ is always a positive integer.
- **Convention**: Let us focus on minimization problems.

The approximation ratio for $A$ is

$$\alpha_A = \max_I \frac{A(I)}{\text{opt}(I)}$$

For maximization problems, take:

$$\alpha_A = \max_I \frac{\text{opt}(I)}{A(I)}$$

Recall from Chapter 5: Set Cover, 1

Suppose $B$ is a set and $S_1, \ldots, S_m \subseteq B$.

**Definition**

(a) A set cover of $B$ is a $\{S'_1, \ldots, S'_k\} \subseteq \{S_1, \ldots, S_m\}$ with $B \subseteq \bigcup_{i=1}^k S'_i$.

(b) A minimal set cover of $B$ is a set cover of $B$ using as few of the $S_i$-sets as possible.

**The Set Cover Problem (SCP)**

**Given**: $B$ and $S_1, \ldots, S_m$ as above.

**Find**: A minimal set cover of $B$.

**Example**

For $B = \{1, \ldots, 14\}$ and

- $S_1 = \{1, 2\}$
- $S_2 = \{3, 4, 5, 6\}$
- $S_3 = \{7, 8, 9, 10, 11, 12, 13, 14\}$
- $S_4 = \{1, 3, 5, 7, 9, 11\}$
- $S_5 = \{2, 4, 6, 8, 10, 12, 14\}$

the solution to SCP is $\{S_4, S_5\}$.

Recall from Chapter 5: Set Cover, 2

**A Greedy Approximation to the Set Cover Problem**

```plaintext
// Input: B and S_1, \ldots, S_m \subseteq B as above.
// Output: A set cover of B which is close to minimal.
C ← ∅
while (some element of B is not yet covered) do
    Pick the S_i with the largest number of uncovered B-elements
    C ← C \cup \{S_i\}
return C
```

**Example**

For $B = \{1, \ldots, 14\}$ and

- $S_1 = \{1, 2\}$
- $S_2 = \{3, 4, 5, 6\}$
- $S_3 = \{7, 8, 9, 10, 11, 12, 13, 14\}$
- $S_4 = \{1, 3, 5, 7, 9, 11\}$
- $S_5 = \{2, 4, 6, 8, 10, 12, 14\}$

The algorithm returns $\{S_1, S_2, S_3\}$. 
A Greedy Approx. to SCP

// Input: B and S₁, . . . , Sₘ ⊆ B
// Output: A near min. set cover
C ← ∅

while (all of B is not covered) do
    Pick the Sᵢ with the largest number of uncovered B-elms
    C ← C ∪ {Sᵢ}
return C

Claim
Suppose B contains n elements and the min. cover has k sets.
Then the greedy algorithm will use at most k ln n sets.

Proof: Let
n₀ = the number of uncovered elms after t-many while loop iterations of the

So n₀ = n.

after iteration t:

Claim
Suppose B contains n elements and the min. cover has k sets.
Then the greedy algorithm will use at most k ln n sets.

Proof: Let
nₜ = the number of uncovered elms after t-many while loop iterations of the
A Greedy Approx. to SCP

// Input: B and $S_1, \ldots, S_m \subseteq B$
// Output: A near min. set cover
$C \leftarrow \emptyset$

while (all of B is not covered) do
    Pick the $S_i$ with the largest number of uncovered B-elms
    $C \leftarrow C \cup \{ S_i \}$
return $C$

Claim
Suppose B contains $n$ elements and the min. cover has $k$ sets.
Then the greedy algorithm will use at most $k \ln n$ sets.

Proof: Let

$n_t = \text{the number of uncovered elms after } t \text{-many while loop iterations of the}$

So $n_0 = n$. After iteration $t$:
- there are $n_t$ elms left.
- $k$ many sets cover them
- So there must be some set with at least $n_t/k$ many elements.

\[
0 \leq n_t - \frac{n_t}{k} = n_t \left(1 - \frac{1}{k}\right) = n_0 \left(1 - \frac{1}{k}\right)^t.
\]
Recall from Chapter 5: Set Cover, 4

**A Greedy Approx. to SCP**

// Input: \( B \) and \( S_1, \ldots, S_m \subseteq B \)
// Output: A near min. set cover
\( C \leftarrow \emptyset \)

while (all of \( B \) is not covered) do
  Pick the \( S_i \) with the largest number of uncovered \( B \)-elms
  \( C \leftarrow C \cup \{ S_i \} \)

return \( C \)

**Claim**

Suppose \( B \) contains \( n \) elements and the min. cover has \( k \) sets.

Then the greedy algorithm will use at most \( k \ln n \) sets.

**Proof:** Let \( n_t = \) the number of uncovered elms after \( t \)-many while loop iterations of the

\[ n_t = n \left( 1 - e^{-t} \right) \]  

We know: \( n_{t+1} \leq n \left( 1 - \frac{1}{e} \right)^t \).

Fact: \( 1 - x \leq e^{-x} \) for all \( x \), with equality iff \( x = 0 \).

Recall from Chapter 5: Set Cover, 5

**A Greedy Approx. to SCP**

// Input: \( B \) and \( S_1, \ldots, S_m \subseteq B \)
// Output: A near min. set cover
\( C \leftarrow \emptyset \)

while (all of \( B \) is not covered) do
  Pick the \( S_i \) with the largest number of uncovered \( B \)-elms
  \( C \leftarrow C \cup \{ S_i \} \)

return \( C \)

**Claim**

Suppose \( B \) contains \( n \) elements and the min. cover has \( k \) sets.

Then the greedy algorithm will use at most \( k \ln n \) sets.

**Proof:** Let \( n_t = \) the number of uncovered elms after \( t \)-many while loop iterations of the

\[ n_t = \frac{n}{e^t} \]  

We know: \( n_{t+1} \leq n \left( 1 - \frac{1}{e} \right)^t \).

Fact: \( 1 - x \leq e^{-x} \) for all \( x \), with equality iff \( x = 0 \).

\[ a_A = \max \frac{A(I)}{\text{Opt}(I)} = \log n. \]

Approximating Vertex Cover, 1

**Vertex Cover (as an optimization problem)**

**Given:** \( G = (V, E) \) an undirected graph
**Find:** \( S \subseteq V \) such that \( S \) touches every edge.
**Goal:** Minimize \( |S| \).

- Vertex Cover is a special case of Set Cover.
- Therefore, it can be approximated within a \( O(\log n) \) factor.
- It turns out we can do much better.

Approximating Vertex Cover, 2

**Definition**

Suppose \( G = (V, E) \) an undirected graph.

(a) A matching is an \( M \subseteq E \) such that any two edges in \( M \) have no endpoints in common.

(b) \( M \) is a maximum matching when for each \( e \in (E - M) \), \( M \cup \{ e \} \) fails to be a matching.

**Observations**

- Maximal matchings are easy to construct.  
  (How?)
- If \( C \) is a vertex cover of \( G \) and \( M \) is a maximum matching, then each \( (u, v) \in M \) must have \( u \in C \) or \( v \in C \).
  (the size of a min. vertex cover for \( G \)) \( \geq \) (the size of a max. matching for \( G \))
- If \( M \) is a maximal matching, then \( S = \{ u; u \text{ is an endpoint of an } e \in M \} \) is a vertex cover.  
  (Why?)
- \( |S| = 2|M| \geq \) (the size of a min. vertex cover for \( G \)) \( \geq |M| \).
Approximating Vertex Cover, 3

An approximation algorithm for Vertex Cover

\[ \text{An approximation algorithm for Vertex Cover} \]

\[ \text{input} \ G = (V, E) \]
\[ \text{return} \ S = \{ u : u \text{ is an endpoint of an edge in } M \} \]

- By the Observations, the approximation ratio of this algorithm is \( a_A \leq 2 \).
- In fact, you can find examples where the ratio is exactly 2.
- The approximation ratio of this algorithm is \( a_A = 2 \).

Amazing Fact (Dinur and Safra, 2005)

Minimum vertex cover cannot be approximated within a factor of 1.3606 for any sufficiently large vertex degree unless P=NP.

Clustering, 1

Definition

A metric on a space \( X \) is a function
\[ d : X \times X \to \mathbb{R}^{\geq 0} \]
such that, for all \( x, y, z \in X \):
1. \( d(x, y) \geq 0 \)
2. \( d(x, y) = 0 \iff x = y \)
3. \( d(x, y) = d(y, x) \)
4. \( d(x, y) \leq d(x, z) + d(z, y) \)

\( k \)-Clustering

Input: Points \( X = \{ x_1, \ldots, x_n \} \), metric \( d \), integer \( k > 0 \).
Output: A partition of \( X \) into \( k \) clusters \( C_1, \ldots, C_k \).
Goal: Minimize the diameter of the clusters: \( \max \max_{j} d(x, x') \).

- \( k \)-Clustering is NP-complete.
- \( k \)-Clustering is important in lots of areas, see http://en.wikipedia.org/wiki/K-clustering#Applications

Clustering, 2

Approximation Algorithm for \( k \)-Clustering

\[ \text{Approximation Algorithm for } k \text{-Clustering} \]

Pick any point \( p_1 \in X \) to start
for \( i \leftarrow 2 \) to \( k \) do
\[ p_i \leftarrow \text{a point in } X \text{ that is farthest away from } p_1, \ldots, p_{i-1} \]
\[ // \text{i.e., } p_i \text{ maximizes: } \min \{ d(x, p_i) : j = 1, \ldots, i - 1 \} \]
Create \( k \) clusters: \( C_i = \{ x \in X : p_i \text{ is the closest center} \} \)

Claim

For the above algorithm, \( a_A \leq 2 \).

- Let \( x \) be the point farthest from \( p_1, \ldots, p_k \).
- Let \( r \) be the distance of \( x \) to the nearest \( p_i \).
- Every point must be within \( r \) from its cluster center.
- The diameter of the clusters is \( \leq 2r \).
- The points \( p_1, \ldots, p_k \) and \( x \) are all \( \geq r \) distant from one another.
- Any partition of \( X \) into \( k \) cluster must put two of \( p_1, \ldots, p_k, x \) into the same cluster. (By the PHP)
- These clusters must have diameter \( \geq r \). QED

Traveling Salesman with metric distances, 1

Traveling Salesman Problem

Given: \( n \) vertices and all \( n \cdot (n - 1) / 2 \)-many distances between them.
Find: An ordering of \( 1, \ldots, n: \pi(1), \pi(2), \ldots, \pi(n) \) so that the tour's cost
\[ d(\pi(1), \pi(2)) + d(\pi(2), \pi(3)) + \cdots + d(\pi(n), \pi(1)) \]
is minimal.

Question: Suppose we require the distances to come from a metric. Does this help make the problem easier? \textbf{Answer: Yes!}

Definition

A metric on a space \( X \) is a function \( d : X \times X \to \mathbb{R}^{\geq 0} \) such that, for all \( x, y, z \in X \):
1. \( d(x, y) \geq 0 \)
2. \( d(x, y) = 0 \iff x = y \)
3. \( d(x, y) = d(y, x) \)
4. \( d(x, y) \leq d(x, z) + d(z, y) \).
Traveling Salesman with metric distances, 2

- Take a TSP path and delete an edge. The result is a spanning tree.
- \( \cdot \cdot \cdot \) (cost of a MST for \( G \))
  \(< \) (cost of a solutions to TSP for \( G \))
- Now take \( T \), a MST for \( G \). Turn \( T \) into a tour that uses each edge twice.
- Let \( c_1, \ldots, c_n \) be the cities on the tour — in the order they are first visited.
- Edit the tour so that from city \( c_i \) the tour shortcuts to city \( c_{i+1} \) and from city \( c_n \) it shortcuts to city \( c_1 \).
- By the triangle inequality, the shortcuts can keep the cost the same or improve it.
- \( \cdot \cdot \cdot \) (cost of a solutions to TSP for \( G \))
  \(< 2 \times \) (cost of a MST for \( G \))
- \( \cdot \cdot \cdot \). We can approximate the metric version of TSP within a factor of 2.

RECALL: Rudrata/Hamiltonian Cycle \( \leq \) TSP

Rudrata/Hamiltonian Cycle Problem
Given: \( G = (V, E) \), an undirected graph.
Find: A simple cycle that visits each vertex of \( G \).

Traveling Salesman Problem (TSP)
Given: \( V', n \) vertices; all \( \frac{n(n-1)}{2} \)-many distances between them; and \( b \), a budget
Find: \( \pi \), an ordering of \( 1, \ldots, n \), such that \( \sum_{i=1}^{n} d_{\pi(i), \pi(1+i \mod n)} \leq b \)

Construction
Given \( (V, E) \) and \( \text{C} \geq 1 \), define
\[
V' = V
\]
\[
d_{ij} = \begin{cases} 1, & \text{if } (i, j) \in E; \\ 1 + \text{C}, & \text{otherwise.} \end{cases}
\]
\[
b = |V|.
\]
\( \cdot \cdot \cdot \). An approximate solution to TSP would let us solve Rudrata Path in polytime!

Approximating General TSP

- Suppose we had \( \mathcal{A} \), a polytime approximation algorithm for TSP with approximation factor \( \alpha_{\mathcal{A}} \).
- Suppose \( G \) is any instance of Rudrata Path.
- Construct \( I(G, C) \) where \( C = \alpha_{\mathcal{A}} \cdot \text{n} \) and run \( \mathcal{A} \) on it.
- If \( G \) has a Rudrata path, then \( \mathcal{A} \) finds a TSP tour of cost \( \alpha_{\mathcal{A}} \cdot \text{OPT}(I(G, C)) = \alpha_{\mathcal{A}} \cdot \text{n} \).
- If \( G \) has no Rudrata path, then \( \mathcal{A} \) must return a tour of cost \( > \alpha_{\mathcal{A}} \cdot \text{n} \).
- Since \( \mathcal{A} \) is supposed to run in polytime, this means we can decide Rudrata path in polytime!!!!
- \( \cdot \cdot \cdot \). If TSP has a polytime approximation algorithm, \( \text{then } \text{P=NP} \).
- \( \cdot \cdot \cdot \). If \( \text{P} \neq \text{NP} \), \( \text{then } \text{TSP has no polytime approximation algorithm} \).

Approximating Knapsack, 1

Knapsack without repetition
Given:
- A knapsack with capacity \( W \).
- Items \( 1, \ldots, n \)
- Item \( i \) has weight \( w_i \) & value \( v_i \).
Find: a set \( M \subseteq \{ 1, \ldots, n \} \)
- \( \sum_{i \in M} w_i \leq W \) and
- \( \sum_{i \in M} v_i \) is maximized.

- By Chapter 6, there is a dynamic programming solution to Knapsack that runs in \( O(n \cdot W) = O(n \cdot 2^{|W|}) \) time.
- There is a similar dynamic programming solution to Knapsack that runs in \( O(n \cdot V) = O(n \cdot 2^{|V|}) \) time, where \( V = \sum_{i=1}^{n} v_i \).
- We use the \( O(n \cdot V) \) version as the basis for an approximation algorithm.
Approximating Knapsack, 2

function ksApprox(⃗v, ⃗w, W, ϵ) // ϵ = an approximation factor
    // Assume each wi ≤ W.
    v_max ← max{ vi : i = 1, ..., n }. // Rescale the values
    for i = 1, ..., n do
        ˜v_i ← ⌊ v_i · n / v_max · ϵ ⌋. // Rescale the values
    Run the dynamic programming algorithm using the ˜v_i values.
    return the resulting choices of items

Runtime Analysis
- Since each ˜v_i ≤ n / ϵ, we have ˜v_1 + ... + ˜v_n ≤ n^2 / ϵ.
- So the DP algorithm runs in O(n^3 / ϵ) time.

Approximation Analysis
Suppose:
- S is an optimal solution to the original problem with total value K^*.
- ˜S is the solution produces for the scaled problem.
Then:
\[ \sum_{i \in S} ˜v_i = \sum_{i \in S} \left[ \frac{v_i \cdot n}{v_{\text{max}} \cdot \epsilon} \right] \geq \sum_{i \in S} \left( \frac{v_i \cdot n}{v_{\text{max}} \cdot \epsilon} - 1 \right) = K^* \cdot \frac{n}{v_{\text{max}} \cdot \epsilon} - n. \]
So:
\[ \sum_{i \in S} v_i \geq \frac{v_{\text{max}} \cdot \epsilon}{n} \sum_{i \in S} ˜v_i \geq \frac{v_{\text{max}} \cdot \epsilon}{n} \left( K^* \cdot \frac{n}{v_{\text{max}} \cdot \epsilon} - n \right) = K^* - v_{\text{max}} \cdot \epsilon \geq K^* (1 - \epsilon). \]

The approximability hierarchy
- No finite approximation ratio is possible.
  E.g., TSP.
- An approximation ratio of about log n is possible.
  E.g., Set Cover.
- A constant approximation ratio is possible, but there are limits to how small this can be.
  E.g., Vertex Cover, k-Clustering, and metric TSP.
  The proofs of these lower limit results are really hard!!!
- A constant approximation ratio is possible, and in fact you can get arbitrarily close to 0.
  E.g., Knapsack.

NOTE: All of the above assumes P ≠ NP. If P=NP, all the problems can be solved exactly in polytime.

Local search heuristics: The general scheme

s ← any initial solution
while there is a solution s' in the neighborhood of s with cost(s') < cost(s) do
    s ← s'
return s

For any application of this scheme to a particular problem, the key question what is a good notion of neighborhood?
Local search heuristics: Traveling Salesman, 1

- Assume we have a complete graph on \( n \) vertices (with a cost assigned to each edge).
- So there are \( (n - 1)! \) many tours.
- Two tours differ by at least two edges. \((Why?)\)
- So let us try:
  - Tours \( T_1 \) and \( T_2 \) are neighbors when they differ by two edges.

- With this choice of “neighbor”:
  1. What is the overall running time?
  2. Does this always return an optimal answer?

- Assume we have a complete graph on \( n \) vertices (with a cost assigned to each edge).
- So there are \( (n - 1)! \) many tours.
- Two tours differ by at least two edges. \((Why?)\)
- So let us try:
  - Tours \( T_1 \) and \( T_2 \) are neighbors when they differ by two edges.

- With this choice of “neighbor”:
  1. What is the overall running time?
  2. Does this always return an optimal answer?
- Answers:
  1. Hard to say.
  2. Of course not.

Local search heuristics: Traveling Salesman, 2

- Tours \( T_1 \) and \( T_2 \) are neighbors when they differ by two edges.

- With this choice of “neighbor”:
  - What is the overall running time?
    - Each tour has \( O(n^2) \) neighbors, so making the choice is not too expensive.
    - But, the algorithm may well go through exponentially many iterations.

Local search heuristics: Traveling Salesman, 3

- Tours \( T_1 \) and \( T_2 \) are neighbors when they differ by two edges.

- With this choice of “neighbor”:
  - What is the overall running time?
  - Does this always return an optimal answer?
    - The final answer will be locally optimal, but not necessarily optimal.
    - The problem is that this notion of neighbor is too myopic. E.g.,

- If we allow three-edge changes, then:

but then a tour has \( O(n^3) \) neighbors and the choice part of the algorithm slows down.
Local search heuristics: Optima, Local vs. global

Figure 9.8 Local search.

Figure 9.7 shows a specific example of a partitioning problem. Given an undirected graph \( G = (V, E) \), with nonnegative edge weights, and \( \alpha \in (0, 1/2] \).

**Goal:** Minimize the capacity of the cut \((A, B)\).

Local search heuristics: Optima, Local vs. global

Start with a partition with \(|A| = |B|\).

Neighbors of \((A, B)\) is

\[
\{ (A - \{a\} + \{b\}, B - \{b\} + \{a\}) : a \in A, b \in B \}.
\]

Local search: Graph partitioning, 1

**Graph partitioning**

**Given:** An undirected graph \( G = (V, E) \) with nonnegative edge weights, and \( \alpha \in (0, 1/2] \).

**Return:** A partition of \( V \) into \( A \) and \( B \) with

\[|A|, |B| \geq \alpha |V|\].

**Goal:** Minimize the capacity of the \((A, B)\)-cut.

**Note:** The general problem is reducible to the special case of \( \alpha = 1/2 \).

**Strategy:**
- Start with a partition with \(|A| = |B|\).
- Neighbors of \((A, B)\) are

\[
\{ (A - \{a\} + \{b\}, B - \{b\} + \{a\}) : a \in A, b \in B \}.
\]

Local search: Graph partitioning, 2

- Start with a partition with \(|A| = |B|\).
- Neighbors of \((A, B)\) is

\[
\{ (A - \{a\} + \{b\}, B - \{b\} + \{a\}) : a \in A, b \in B \}.
\]

Local search: Graph partitioning, 3

- The problem with this notion of neighbor is that there are stubborn local minima.
Dealing with local optima: Randomized Restarts

\[ L \leftarrow \text{an empty list} \]
\[ \text{repeat } k \text{ times} \]
\[ s \leftarrow \text{a randomly chosen initial solution} \]
\[ \text{while (there is a solution } s' \text{ in the neighborhood of } s \text{ with } cost(s') < cost(s) ) \text{ do} \]
\[ s \leftarrow s' \]
\[ \text{add } s \text{ to } L \]
\[ \text{end-repeat} \]
\[ \text{return the best solution in } L \]

This can shake free of bad local optima.

Dealing with local optima: Simulated Annealing

\[ s \leftarrow \text{a randomly chosen initial solution} \]
\[ \text{repeat} \]
\[ s' \leftarrow \text{a randomly chosen solution in } s\text{'s neighborhood} \]
\[ \Delta \leftarrow cost(s') - cost(s) \]
\[ \text{if (} \Delta < 0 \text{) then } s \leftarrow s' \]
\[ \text{else with probability } e^{-\Delta/T} \text{ do } s \leftarrow s' \]
\[ \text{until we decide we are done} \]

- \( T \equiv \text{temperature} \)
- If \( T \approx 0 \) this is roughly the previous scheme.
- If \( T \) is big, then \( s \) jumps around a lot.
- We vary \( T \), initially large (hot), and gradually small (cooler).