Agenda

1. Introduction
2. Uninformed Search versus Informed Search (Best First Search, A* Search, Heuristics)
3. Local Search, Stochastic Hill Climbing, Simulated Annealing
4. Tabu Search
5. Answer-set Programming (ASP)
6. Constraint Satisfaction (CSP)
7. Structural Decomposition Techniques (Tree/Hypertree Decompositions)
8. Evolutionary Algorithms/ Genetic Algorithms
Hill-climbing Methods

- Hill climbing methods use an iterative improvement technique.
- Technique is applied to a single point - the current point - in the search space.
- During each iteration, a new point is selected from the neighborhood of the current point.
- If new point provides better value (in light of evaluation function) the new point becomes the current point.
- Otherwise, some other neighbor is selected and tested against the current point.
- The method terminates if no further improvement is possible, or we run out of time.
Iterated Hill-Climber

**Algorithm** iterated hill-climber

\[ t \leftarrow 0 \]

initialize best

repeat

\[ local \leftarrow \text{FALSE} \]

select a current point \( v_c \) at random

evaluate \( v_c \)

repeat

select all new points in the neighborhood of \( v_c \)

select the point \( v_n \) from the set of new points with the best value of evaluation function \( \text{eval} \)

if \( \text{eval}(v_n) \) is better than \( \text{eval}(v_c) \) then

\[ v_c \leftarrow v_n \]

else

\[ local \leftarrow \text{TRUE} \]

end if

until \( local \)

\[ t \leftarrow t + 1 \]

if \( v_c \) is better than best then

\[ best \leftarrow v_c \]

end if

until \( t = \text{MAX} \)
Weaknesses of Hill-climbing Algorithms

1. They usually terminate at solutions that are only locally optimal.
2. No information about how much the local optimum deviates from the global optimum, or from other local optima.
3. The obtained optimum depends on the initial configuration.
4. In general, it is not possible to provide an upper bound for the computation time.

But, they are easy to apply. All that is needed is:

- the representation,
- the evaluation function, and
- a measure that defines the neighborhood around a given solution.
Balance Between Exploration and Exploitation

Effective search techniques provide a mechanism for balancing two conflicting objectives:

- exploiting the best solutions found so far, and
- at the same time exploring the search space.
Balance Between Exploration and Exploitation

Effective search techniques provide a mechanism for balancing two conflicting objectives:

- exploiting the best solutions found so far, and
- at the same time exploring the search space.

Hill-climbing Techniques

Exploit the best available solution for possible improvement but neglect exploring a large portion of the search space.
Balance Between Exploration and Exploitation

Effective search techniques provide a mechanism for balancing two conflicting objectives:

- exploiting the best solutions found so far, and
- at the same time exploring the search space.

Hill-climbing Techniques

Exploit the best available solution for possible improvement but neglect exploring a large portion of the search space.

Random Search

Explores the search space thoroughly (points are sampled from the search space with equal probabilities) but foregoes exploiting promising regions.
Balance Between Exploration and Exploitation

Effective search techniques provide a mechanism for balancing two conflicting objectives:

• exploiting the best solutions found so far, and
• at the same time exploring the search space.

There is no way to choose a single search method that can serve well in every case!
Local Search

1. Pick a solution from the search space and evaluate its merit. Define this as the current solution.
2. Apply a transformation to the current solution to generate a new solution and evaluate its merit.
3. If the new solution is better than the current solution then exchange it with the current solution; otherwise discard the new solution.
4. Repeat sets 2 and 3 until no transformation in the given set improves the current solution.
Local Search

1. Pick a solution from the search space and evaluate its merit. Define this as the current solution.

2. Apply a transformation to the current solution to generate a new solution and evaluate its merit.

3. If the new solution is better than the current solution then exchange it with the current solution; otherwise discard the new solution.

4. Repeat sets 2 and 3 until no transformation in the given set improves the current solution.

The key lies in the type of the transformation applied to the current solution.
Local Search

1. Pick a solution from the search space and evaluate its merit. Define this as the **current** solution.
2. Apply a **transformation** to the current solution to generate a new solution and evaluate its merit.
3. If the **new solution is better** than the current solution then **exchange** it with the current solution; otherwise discard the new solution.
4. **Repeat** sets 2 and 3 until no **transformation** in the given set **improves** the current solution.

The key lies in the type of the **transformation** applied to the current solution.

- One extreme could be to return a potential solution from the search space **selected uniformly at random**.
Local Search

1. Pick a solution from the search space and evaluate its merit. Define this as the current solution.

2. Apply a transformation to the current solution to generate a new solution and evaluate its merit.

3. If the new solution is better than the current solution then exchange it with the current solution; otherwise discard the new solution.

4. Repeat sets 2 and 3 until no transformation in the given set improves the current solution.

The key lies in the type of the transformation applied to the current solution.

- One extreme could be to return a potential solution from the search space selected uniformly at random.
  - Then, current solution has no effect on the probabilities of selecting any new solution.
  - The search becomes essentially enumerative.
  - Could be even worse: one might resample points that have already been tried.
Local Search

1. Pick a solution from the search space and evaluate its merit. Define this as the current solution.

2. Apply a transformation to the current solution to generate a new solution and evaluate its merit.

3. If the new solution is better than the current solution then exchange it with the current solution; otherwise discard the new solution.

4. Repeat sets 2 and 3 until no transformation in the given set improves the current solution.

The key lies in the type of the transformation applied to the current solution.

- One extreme could be to return a potential solution from the search space selected uniformly at random.
  - Then, current solution has no effect on the probabilities of selecting any new solution.
  - The search becomes essentially enumerative.
  - Could be even worse: one might resample points that have already been tried.

- Another extreme would be to always return the current solution - this gets you nowhere!
Local Search ctd.

- Searching within some local neighborhood of current solution is a useful compromise.
- Then, current solution imposes a bias on where we can search next.
- If we find something better, we can update the current point to new solution and retain what we have learned.
- If the size of the neighborhood is small, the search might be very quick, but we might get trapped at local optimum.
- If the size of neighborhood is very large, there is less chance to get stuck, but the efficiency may suffer.
- The type of transformation we apply determines the size of neighborhood.
Local Search and the SAT

Local search algorithms are surprisingly good at finding satisfying assignments for certain classes of SAT formulas. GSAT is one of the best-known (randomized) local search algorithms for SAT.

Algorithm GSAT

```
for i ← 1 step 1 to MAX-TRIES do
    T ← a randomly generated truth assignment
    for j ← 1 step 1 to MAX-FLIPS do
        if T satisfies the formula then
            return(T)
        else
            make a flip
        end if
    end for
    return("no satisfying assignment found")
end for
```
Local Search and the SAT

Algorithm GSAT

for \( i \leftarrow 1 \) step 1 to MAX-TRIES do
  \( T \leftarrow \) a randomly generated truth assignment
  for \( j \leftarrow 1 \) step 1 to MAX-FLIPS do
    if \( T \) satisfies the formula then
      return(\( T \))
    else
      make a flip
    end if
  end for
return("no satisfying assignment found")

• "make a flip" flips the variable in \( T \) that results in the largest decrease in the number of unsatisfied clauses.
• MAX-TRIES, determines the number of new search sequences.
• MAX-FLIPS, determines the maximum number of moves per try.
Local Search and the SAT ctd.

- GSAT begins with randomly generated truth assignment.
- If assignment satisfies the problem, the algorithm terminates.
- Else, it flips each of the variables from TRUE to FALSE or FALSE to TRUE and records the decrease in the number of unsatisfied clauses.
- After trying all possible flips, it updates current solution to solution with largest decrease in unsatisfied clauses.
- If this new solution satisfies the problem, we are done.
- Otherwise, the algorithm starts flipping again.
Local Search and the SAT ctd.

- GSAT begins with randomly generated truth assignment.
- If assignment satisfies the problem, the algorithm terminates.
- Else, it flips each of the variables from TRUE to FALSE or FALSE to TRUE and records the decrease in the number of unsatisfied clauses.
- After trying all possible flips, it updates current solution to solution with largest decrease in unsatisfied clauses.
- If this new solution satisfies the problem, we are done.
- Otherwise, the algorithm starts flipping again.

**Interesting feature of the algorithm:**

- Best available flip might **decrease** the number of unsatisfied clauses.
- Selection is only made from neighborhood of current solution. If every neighbor (defined as being one flip away) is worse than current solution, then GSAT takes the one that is the least bad.

- Has the chance to escape local optimum!
  - But, it might **oscillate** between points and never escape from some plateaus.
  - One can assign a **weight** to each clause, and **increase** the weight for those who remain unsatisfied.
Local Search and the TSP

- There are many local search algorithms for TSP.
- The simplest is called $2-opt$.
- Starts with random permutation of cities (call this tour $T$) and tries to improve it.
- Neighborhood of $T$ is defined as the set of all tours that can be reached by changing two nonadjacent edges in $T$.
- This move is called a $2-interchange$. 
Local Search and the TSP

- There are many local search algorithms for TSP.
- The simplest is called $2-opt$.
- Starts with random permutation of cities (call this tour $T$) and tries to improve it.
- Neighborhood of $T$ is defined as the set of all tours that can be reached by changing two nonadjacent edges in $T$.
- This move is called a $2-interchange$. 
2\textemdash opt Algorithm

- A new tour $T'$ replaces $T$ if it is better.
  - Note: we replace the tour every time we find an improvement.
  - Thus, we terminate the search in the neighborhood of $T$ when the first improvement is found.

- If none of the tours in neighborhood of $T$ is better, then $T$ is called 2\textemdash optimal and algorithm terminates.

- As GSAT, algorithm should be restarted from several random permutations.
2−opt Algorithm

- A new tour $T'$ replaces $T$ if it is better.
  - Note: we replace the tour every time we find an improvement.
  - Thus, we terminate the search in the neighborhood of $T$ when the first improvement is found.
- If none of the tours in neighborhood of $T$ is better, then $T$ is called 2−optimal and algorithm terminates.
- As GSAT, algorithm should be restarted from several random permutations.
- Can be generalized to $k$−opt, where either $k$ or upto $k$ edges are selected.
- Trade-off between size of neighborhood and efficiency of the search:
  - If $k$ is small the entire neighborhood can be searched quickly, but increases likelihood of suboptimal answer.
  - For larger values of $k$, the number of solutions in neighborhood become enormous (grows exponentially with $k$). Seldomly used for $k > 3$. 
Escaping Local Optima

- Traditional problem-solving strategies either
  - guarantee discovering global solution, but are too expensive, or
  - have a tendency of "getting stuck" in local optima.

- There is almost no chance to speed up algorithms that guarantee finding global solution.
  - Problem of finding polynomial-time algorithms for real problems (as they are NP-hard).

- Remaining option is to design algorithms capable of escaping local optima.
Escaping Local Optima

- Traditional problem-solving strategies either
  - guarantee discovering global solution, but are too expensive, or
  - have a tendency of "getting stuck" in local optima.

- There is almost no chance to speed up algorithms that guarantee finding global solution.
  - Problem of finding polynomial-time algorithms for real problems (as they are NP-hard).

- Remaining option is to design algorithms capable of escaping local optima.

Simulated Annealing

Additional parameter (called temperature) that change the probability of moving from one point of the search space to another.
Escaping Local Optima

- Traditional problem-solving strategies either
  - guarantee discovering global solution, but are too expensive, or
  - have a tendency of "getting stuck" in local optima.
- There is almost no chance to speed up algorithms that guarantee finding global solution.
  - Problem of finding polynomial-time algorithms for real problems (as they are NP-hard).
- Remaining option is to design algorithms capable of escaping local optima.

**Simulated Annealing**

Additional parameter (called temperature) that change the probability of moving from one point of the search space to another.

**Tabu Search**

Memory, which forces the algorithm to explore new areas of the search space.
Local Search Revisited

**Algorithm** local search

\[
x = \text{some initial starting point in } S \\
\textbf{while} \ \text{improve}(x) \neq "\text{no}" \ \textbf{do} \\
\quad x = \text{improve}(x) \\
\textbf{end while} \\
\text{return}(x)
\]
Local Search Revisited

**Algorithm** local search

\[
x = \text{some initial starting point in } S
\]

**while** improve\((x)\) \(\neq \) "no" **do**

\[
x = \text{improve}\((x)\)
\]

**end while**

**return**\((x)\)

- **improve**\((x)\) returns new point \(y\) from neighborhood of \(x\), i.e., \(y \in N(x)\), if \(y\) is better than \(x\),
- otherwise, returns a string "no". In that case, \(x\) is a local optimum in \(S\).
Simulated Annealing

Algorithm simulated annealing

\[ x = \text{some initial starting point in } S \]
\[ \text{while not termination-condition do} \]
\[ x = \text{improve?}(x, T) \]
\[ \text{update}(T) \]
\[ \text{end while} \]
\[ \text{return}(x) \]
Simulated Annealing vs. Local Search

There are three important differences:

1. How the procedure halts.
   - Simulated annealing is executed until some external termination condition is satisfied.
   - Local search is performed until no improvement is found.

2. `improve?(x, T)` doesn’t have to return a better point from the neighborhood of `x`. It returns an accepted solution `y ∈ N(x)`, where acceptance is based on the current temperature `T`.

3. Parameter `T` is updated periodically, and the value of `T` influences the outcome of the procedure "improve?".
Algorithm iterated hill-climber

\[ t \leftarrow 0 \]
initialize \(\text{best}\)
repeat
    \[t \leftarrow \text{MAX}\]
    if \(v_c\) is better than \(\text{best}\) then
        \(\text{best} \leftarrow v_c\)
    end if
    repeat
        \(\text{local} \leftarrow \text{FALSE}\)
        select a current point \(v_c\) at random
        evaluate \(v_c\)
        repeat
            select all new points in the neighborhood of \(v_c\)
            select the point \(v_n\) from the set of new points with the best value of evaluation function \(\text{eval}\)
            if \(\text{eval}(v_n)\) is better than \(\text{eval}(v_c)\) then
                \(v_c \leftarrow v_n\)
            else
                \(\text{local} \leftarrow \text{TRUE}\)
            end if
        until \(\text{local}\)
        \(t \leftarrow t + 1\)
    until \(t = \text{MAX}\)
Modification of Iterated Hill-Climber

- **Instead** of checking all strings in the neighborhood of $v_c$ and selecting the best one, select only one point, $v_n$, from this neighborhood.

- Accept this new point, i.e., $v_c \leftarrow v_n$ with some probability that depends on the relative merit of these two points, i.e., the difference between the values returned by the evaluation function for these two points.
Modification of Iterated Hill-Climber

- **Instead** of checking all strings in the neighborhood of $v_c$ and selecting the best one, select only one point, $v_n$, from this neighborhood.

- Accept this new point, i.e., $v_c \leftarrow v_n$ with some probability that depends on the relative merit of these two points, i.e., the difference between the values returned by the evaluation function for these two points.

$\Rightarrow$ **Stochastic hill-climber**
**Stochastic Hill-Climber**

**Algorithm** stochastic hill-climber

\[
t \leftarrow 0
\]

select a current point \( v_c \) at random

evaluate \( v_c \)

repeat

select the string \( v_n \) from the neighborhood of \( v_c \)

select \( v_n \) with probability

\[
\frac{1}{1 + e^{\frac{\text{eval}(v_c) - \text{eval}(v_n)}{T}}}
\]

\[
t \leftarrow t + 1
\]

until \( t = \text{MAX} \)
Analyzing Stochastic Hill-Climber

- Probabilistic formula for accepting a new solution is based on maximizing the evaluation function.
- It has only one loop. No repeated calls from different random points.
- Newly selected point is accepted with probability $p$. Thus, the rule of moving from current point $v_c$ to new neighbor, $v_n$, is probabilistic.
- New accepted point can be worse than current point.
- $p = \frac{1}{1 + e^{\frac{\text{eval}(v_c) - \text{eval}(v_n)}{T}}}$
- Probability of acceptance depends on the difference in merit between these two competitors, i.e., $\text{eval}(v_c) - \text{eval}(v_n)$, and on the value of an additional parameter $T$.
- $T$ remains constant during the execution of the algorithm.
Role of Parameter $T$

Example:

- $eval(v_c) = 107$ and $eval(v_n) = 120$
- $eval(v_c) - eval(v_n) = -13$, new point $v_n$ is better than $v_c$
Role of Parameter $T$

Example:

- $eval(v_c) = 107$ and $eval(v_n) = 120$
- $eval(v_c) - eval(v_n) = -13$, new point $v_n$ is better than $v_c$
- What is probability of accepting new point based on different values of $T$?
Role of Parameter $T$

Example:

- $eval(v_c) = 107$ and $eval(v_n) = 120$
- $eval(v_c) - eval(v_n) = -13$, new point $v_n$ is better than $v_c$
- What is probability of accepting new point based on different values of $T$?

<table>
<thead>
<tr>
<th>$T$</th>
<th>$e^{-\frac{13}{T}}$</th>
<th>$p$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.000002</td>
<td>1.00</td>
</tr>
<tr>
<td>5</td>
<td>0.0743</td>
<td>0.93</td>
</tr>
<tr>
<td>10</td>
<td>0.2725</td>
<td>0.78</td>
</tr>
<tr>
<td>20</td>
<td>0.52</td>
<td>0.66</td>
</tr>
<tr>
<td>50</td>
<td>0.77</td>
<td>0.56</td>
</tr>
<tr>
<td>$10^{10}$</td>
<td>0.9999...</td>
<td>0.5...</td>
</tr>
</tbody>
</table>
Role of Parameter $T$

Example:

- $eval(v_c) = 107$ and $eval(v_n) = 120$
- $eval(v_c) - eval(v_n) = -13$, new point $v_n$ is better than $v_c$
- What is probability of accepting new point based on different values of $T$?

<table>
<thead>
<tr>
<th>$T$</th>
<th>$e^{-\frac{13}{T}}$</th>
<th>$p$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.000002</td>
<td>1.00</td>
</tr>
<tr>
<td>5</td>
<td>0.0743</td>
<td>0.93</td>
</tr>
<tr>
<td>10</td>
<td>0.2725</td>
<td>0.78</td>
</tr>
<tr>
<td>20</td>
<td>0.52</td>
<td>0.66</td>
</tr>
<tr>
<td>50</td>
<td>0.77</td>
<td>0.56</td>
</tr>
<tr>
<td>$10^{10}$</td>
<td>0.9999...</td>
<td>0.5...</td>
</tr>
</tbody>
</table>

- The greater $T$, the smaller the importance of the relative merit of the competing points!
- If $T$ is huge (e.g., $T = 10^{10}$), the probability of acceptance approaches 0.5. The search becomes random!
- If $T$ is very small (e.g., $T = 1$), we have an ordinary hill-climber!
Role of new String

Suppose $T = 10$ and $eval(v_c) = 107$. Then, probability of acceptance depends only on the value of the new string.

<table>
<thead>
<tr>
<th>$eval(v_n)$</th>
<th>$eval(v_c) - eval(v_n)$</th>
<th>$\frac{eval(v_c) - eval(v_n)}{T}$</th>
<th>$p$</th>
</tr>
</thead>
<tbody>
<tr>
<td>80</td>
<td>27</td>
<td>14.88</td>
<td>0.06</td>
</tr>
<tr>
<td>100</td>
<td>7</td>
<td>2.01</td>
<td>0.33</td>
</tr>
<tr>
<td>107</td>
<td>0</td>
<td>1.00</td>
<td>0.50</td>
</tr>
<tr>
<td>120</td>
<td>−13</td>
<td>0.27</td>
<td>0.78</td>
</tr>
<tr>
<td>150</td>
<td>−43</td>
<td>0.01</td>
<td>0.99</td>
</tr>
</tbody>
</table>
Role of new String

Suppose $T = 10$ and $eval(v_c) = 107$. Then, probability of acceptance depends only on the value of the new string.

<table>
<thead>
<tr>
<th>$eval(v_n)$</th>
<th>$eval(v_c) - eval(v_n)$</th>
<th>$\frac{eval(v_c) - eval(v_n)}{T}$</th>
<th>$p$</th>
</tr>
</thead>
<tbody>
<tr>
<td>80</td>
<td>27</td>
<td>14.88</td>
<td>0.06</td>
</tr>
<tr>
<td>100</td>
<td>7</td>
<td>2.01</td>
<td>0.33</td>
</tr>
<tr>
<td>107</td>
<td>0</td>
<td>1.00</td>
<td>0.50</td>
</tr>
<tr>
<td>120</td>
<td>−13</td>
<td>0.27</td>
<td>0.78</td>
</tr>
<tr>
<td>150</td>
<td>−43</td>
<td>0.01</td>
<td>0.99</td>
</tr>
</tbody>
</table>

- If new point has same merit as current point, i.e., $eval(v_c) = eval(v_n)$, the probability of acceptance is 0.5.
- If new point is better, the probability of acceptance is greater than 0.5.
- The probability of acceptance grows together with the (negative) difference between these evaluations.
Simulated Annealing

- **Main difference** to stochastic hill-climber is that simulated annealing changes the parameter $T$ during the run.
- **Starts with high values of** $T$ making procedure more similar to random search, and then **gradually decreases** value of $T$.
- Towards end of the run, values of $T$ are quite small, like an ordinary hill-climber.
- In addition, **new points are always accepted** if they are **better** than current point.
Algorithm simulated annealing

\[
t \leftarrow 0 \\
\text{initialize } T \\
\text{select a current point } v_c \text{ at random} \\
\text{evaluate } v_c \\
\text{repeat} \\
\quad \text{repeat} \\
\quad \quad \text{select new point } v_n \text{ in the neighborhood of } v_c \\
\quad \quad \text{if } \text{eval}(v_c) < \text{eval}(v_n) \text{ then} \\
\quad \quad \quad v_c \leftarrow v_n \\
\quad \quad \text{else if } \text{random}[0, 1] < e^{\frac{\text{eval}(v_c) - \text{eval}(v_n)}{T}} \text{ then} \\
\quad \quad \quad v_c \leftarrow v_n \\
\quad \text{end if} \\
\text{until (termination-condition)} \\
T \leftarrow g(T, t) \\
t \leftarrow t + 1 \\
\text{until (halting-criterion)}
\]
Simulated Annealing ctd.

- Is also known as Monte Carlo annealing, statistical cooling, probabilistic hill-climbing, stochastic relaxation, and probabilistic exchange algorithm.
- Based on an analogy taken from thermodynamics.
  - To grow a crystal, the raw material is heated to a molten state.
  - The temperature of the crystal melt is reduced until the crystal structure is frozen in.
  - Cooling should not be done too quickly, otherwise some irregularities are locked in the crystal structure.
Analogies Between Physical System and Optimization Problem

<table>
<thead>
<tr>
<th>Physical System</th>
<th>Optimization Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>state</td>
<td>feasible solution</td>
</tr>
<tr>
<td>energy</td>
<td>evaluation function</td>
</tr>
<tr>
<td>ground state</td>
<td>optimal solution</td>
</tr>
<tr>
<td>rapid quenching</td>
<td>local search</td>
</tr>
<tr>
<td>temperature</td>
<td>control parameter $T$</td>
</tr>
<tr>
<td>careful annealing</td>
<td>simulated annealing</td>
</tr>
</tbody>
</table>
Problem-Specific Questions

As with any search algorithm, simulated annealing requires answers for the following problem-specific questions.

- What is a solution?
- What are the neighbors of a solution?
- What is the cost of a solution?
- How do we determine the initial solution?

Answers yield the structure of the search space together with the definition of a neighborhood, the evaluation function, and the initial starting point.
Problem-Specific Questions

As with any search algorithm, simulated annealing requires answers for the following problem-specific questions.

- What is a solution?
- What are the neighbors of a solution?
- What is the cost of a solution?
- How do we determine the initial solution?

Answers yield the structure of the search space together with the definition of a neighborhood, the evaluation function, and the initial starting point.

Further Questions

- How do we determine the initial "temperature" $T$?
- How do we determine the cooling ration $g(T, t)$?
- How do we determine the termination condition?
- How do we determine the halting criterion?
STEP 1:
\[ T \leftarrow T_{\text{max}} \]
select \( v_c \) at random

STEP 2:
pick a point \( v_n \) from the neighborhood of \( v_c \)
if \( \text{eval}(v_n) \) is better than \( \text{eval}(v_c) \) then
select if \( (v_c \leftarrow v_n) \)
else
select it with probability \( e^{-\frac{\Delta \text{eval}}{T}} \)
end if
repeat
this step
until \( k_T \) times

STEP 3:
set \( T \leftarrow rT \)
if \( T \geq T_{\min} \) then
goto STEP 2
else
goto STEP 1
end if

Where, \( T_{\text{max}} \) initial temperature, \( k_T \) number of iterations, \( r \) cooling ratio, and \( T_{\min} \) frozen temperature.
Algorithm SA-SAT

```plaintext
tries ← 0
repeat
    v ← random truth assignment
    j ← 0
    repeat
        if v satisfies the clauses then
            return v
            T = T_{max} \cdot e^{-j \cdot r}
        for k = 1 to the number of variables do
            compute the increase (decrease) δ in the number of clauses made true if v_k was flipped
            flip variable v_k with probability \( (1 + e^{-\frac{\delta}{T}})^{-1} \)
            v ← new assignment if the flip is made
        end for
        j ← j + 1
    end if
until T < T_{min}
tries ← tries + 1
until tries = MAX-TRIES
```

TU Dresden, 5th May 2015

PSSAI
SA for SAT ctd.

- Outermost loop variable called "tries" keeps track of the number of independent attempts to solve the problem.
- $T$ is set to $T_{max}$ at the beginning of each attempt ($j \leftarrow 0$) and a new random truth assignment is made.
- Inner repeat loop tries different assignments by probabilistically flipping each of the Boolean variables.
- Probability of a flip depends on the improvement $\delta$ of the flip and the current temperature.
- If the improvement is negative, the flip is unlikely to be accepted and vice versa.
- $r$ represents a decay rate for the temperature, the rate where it drops from $T_{max}$ to $T_{min}$.
- The drop is caused by incrementing $j$, as $T = T_{max} \cdot e^{-j \cdot r}$. 
SA-SAT vs. GSAT

- **Major difference**: GSAT can make a **backward move** (decrease in number of unsatisfied clauses) if other moves are not available.
- GSAT cannot make two backward moves in a row, as one backward move implies existence of next improvement move!
- SA-SAT can make an **arbitrary sequence of backward moves**, thus **escape local optima**!
- SA-SAT appeared to satisfy at least as many formulas as GSAT, with less work.
- **Applications** of SA: traveling salesman problem, production scheduling, Timetabling problems and image processing
Summary

- Hill-climbing methods face a danger of getting trapped in local optima.
- Local search can make one backward move.
- Simulated annealing is designed to escape local optima and can make uphill moves at any time.
- Hill-climbing, local search and SA work on complete solutions.
- SA has many parameters to worry about (temperature, rate of reductions, ...).
- The more sophisticated the method, the more you have to use your judgment as to how it should be utilized.