CSE 460
Stochastic Optimization

In this section we will look at general nonlinear optimization. The emphasis in this section will be on combinatorial problems, but some methods (SA, GA, ...) can also be used for numeric problems.

- Local Search
- Heuristics
- Stochastic Constraint Solving & Optimization
  - Simulated Annealing
  - Constraint Simulated Annealing
- Tabu Search
A general search framework that we will modify in the following is local search.

```
procedure local_search

generate initial solution G;
current solution S := G;
improved := true;

repeat
    S := some neighbour(S) such that quality(S') > quality(S);
    if no such neighbour exists exit loop; end;
end;

return S;
```

The key concept is “neighbourhood” relation on candidate solutions that allows to take a candidates to a slightly changed candidate. The idea is to find a balance between moving blindly to any other solution and not moving at all.

The definition of neighbourhood is problem dependent.

Like any other greedy search method, local search can get stuck in local extrema.
procedure GSAT(boolean formula f in conjunctive normal form)
begin
  for i from 1 to MAX_TRIES do
  begin
    T := random valuation of variables in f;
    for j from 1 to MAX-FLIPS do
      if f/T is true then return(T);
      else flip();
    end;
  end;
return “no solution found”;
end.

flip toggles the value of some variable x in T between true/false. x is chosen among all variables such that the number of dissatisfied clauses in f/T is minimal among all possible choices for x.
Local Search for TSP

For TSP a commonly used neighbourhood function is 2-interchange:

The 2-interchange neighbourhood is the set of all tours that can be obtained by changing two non-adjacent edges (without disconnecting).

This can easily be generalized to k-neighbourhood. Usually k=3 is used.

Trade off: Efficient search of neighbourhood vs. probability to find a good solution.

Remark: Any improving neighbour is accepted as a new candidate, the neighbourhood is not searched exhaustively for the best improvement.
The simplest idea to avoid finding only local minima is to perform local search from multiple starting points.

procedure multi_start

    current solution S := nil;
    improved := true;

    while (improved)
        generate new start point set G*;
        for each g in G*
            perform local search from g;
            call the end point of this search g';
            replace g in G* by g';
        end
        S := minimum({S} union G*);
        if (S remains unchanged) then improved := false;
    end

    return S;
Sampling and Clustering tries to avoid the danger of repeatedly arriving at the same local minima with a simple multi start procedure:

We use three sets:

- \( Y \) is the set of candidates, i.e. potential starting points for the next local search
- \( Y^* \) is the set of local optima already found
- \( Y' \) is the set of “cluster points” (start points that lead only to optima in \( Y^* \))

\( Y^* \cup Y' \) are called the seed points

S&C tries to avoid duplicating solutions by clustering possible starting points for multi start.

Obviously it can only be beneficial when the cost of clustering is less than the cost of local optimization.

S&C are generally based on a distance function in the neighbourhood space.
procedure sampling_and_clustering

Y := Y* := Y' := nil;
improved := true;

while (improved)
    generate $k$ new starting points and add them to $Y$;
    eliminate from $Y$ a fraction $w$ of point with the worst objective values;
    cluster new points of $Y$ around seed points ($Y^* \cup Y'$)
    for each $g$ in remaining un-clustered points in $Y$
        perform local search from $g$;
        call the end point of this search $g'$;
        if $g'$ not in $Y^*$
            then add $g'$ to $Y^*$
            else add $g$ to $Y'$
        end
    end
    if ($Y'$ remains unchanged) then improved := false;
end;

return $S$;
Heuristics

If a complete and blind search is too costly, we need to direct the search.

Often this can be done by problem-dependent “guessing”.

A search that is directed by such guesses is called “heuristic search”.

A heuristic can either

• only direct the order in which solutions are (exhaustively) explored
• restrict search to explore only a subset of possible solutions.
To solve an 8-puzzle we can build a tree of all possible move sequences.

down(2) → right(6)

down(4) → right(3) → down(3) → right(8)

down(2) → up(2) → right(3) → down(3) → left(6)
8-Puzzle Heuristics

Find the minimum number of moves needed to solve the puzzle! Computing the entire tree is too costly!

- Use a heuristic (lower bound) and expand only the cheapest node

\[ h(\text{state}) = \text{number of moves from start to state} + \text{number of misplaced tiles in state} \]

Keep all nodes on all levels open and always expand the one with the smallest lower bound.

This guarantees that the cheapest solution is found first.
Example A: Bin-Packing / Cutting Stock Problem  
(can be handled by a linear programming approach)

A number of items \{i[1], ..., i[n]\} with given sizes \(s(i[j])\)

\[0 < s(i[j]) \leq 1\]

have to be packed into bins of standard size 1.

Use the minimum number of bins!

Example: \{1/3, 3/4, 1/4, 2/3, 3/8, 1/4\}
Algorithm first-fit (heuristic bin-packing)

\[ n := \text{number of items}; \]
\[ \text{number the bins from } b[1]...b[n] \]
\[ \text{for } j \text{ from } 1 \text{ to } n \]
\[ k := 1; \text{ stored := false} \]
\[ \text{while } (k<=n \text{ and not stored}) \]
\[ \quad \text{if } s[i[j]] <= \text{free}[b[k]] \]
\[ \quad \quad \text{then begin} \]
\[ \quad \quad \quad \text{put } i[j] \text{ in } b[k]; \]
\[ \quad \quad \quad \text{stored=true;} \]
\[ \quad \quad \text{end;} \]
\[ \quad \text{else } k := k+1; \]
\[ \quad \text{end;} \]
\[ \text{end;} \]
\[ \text{end.} \]
Bounds of first-fit:

Let $opt$ be the optimum number of bins.
Let $ff$ be the number of bins required by first-fit.

(1) first fit can not use more than $n$ bins.

(2) no more than a single bin can be less than or exactly half full

(The move for the second half-full bin would have passed over the first half-full bin.)

(3) we need to use at least as many bins as if there was no space wasted

$$\sum_{j=1}^{n} s[i[j]] \leq opt \land ff \leq 2 \sum_{j=1}^{n} s[i[j]]$$

$$ff \leq 2 \cdot opt$$

a more involved analysis can in fact show that $ff \leq \frac{17}{10} \cdot opt$. 
Example B: Travelling salesman

Algorithm tour (heuristic TSP)

- construct a minimum spanning tree $T$;
- duplicate all edges in $T$ generating $T'$;
- extract an Euler tour from $T'$;
- modify the Euler tour such that vertices visited twice are skipped.
Bounds on Heuristics (B2)

Bounds of tour:

Let $opt$ be the length of the optimal route. Let $t$ be the length of the route constructed by tour.

(1) the optimal tour can not be shorter than the minimum spanning tree.
(2) the Euler tour has twice the length of the spanning tree size.
(3) the final tour is shorter than the Euler tour.

$$opt \leq t \leq 2 \cdot opt$$

A more involved analysis can show that $opt \leq t \leq 3/2 \cdot opt$. 
Meta-Heuristics

• Heuristics only modify a search procedure for a particular problem.
• Meta-heuristics are universal search frameworks that must still be tailored for a particular problem.
• Most stochastic meta-heuristics are based on natural metaphors.

• Stochastic/Randomized Methods
  - Simulated Annealing etc.
    Constraint Simulated Annealing
  - Evolutionary Methods
    Genetic Algorithms
    Genetic Programming
  - Ant Colonies
  - Connectionist Approaches
    eg Boltzmann Machines
• Deterministic Methods
  - Tabu Search
Stochastic Search

In order to avoid getting trapped in local minima, many meta-heuristics use stochastic (randomized) search methods. The general idea is to combine gradient descent/ascent with random sampling.

- Gradient descent can be used
- Use random jumps to escape from local minima
Metropolis Algorithm

This basic idea is embodied in the Metropolis algorithm which simulates the behaviour of molecules at constant temperature (statistical mechanics).

\[ t := 1; \]
\[ \text{start from a random configuration } K[0]. \]
\[ \text{repeat} \]
\[ \text{make a random transition in state space from } K[t-1] \text{ to } K' \]
\[ \text{if } \text{energy}(K') \leq \text{energy}(K[t-1]) \text{ then accept } K' \text{ as the new } K[t] \]
\[ \text{else accept with probability } \exp\left( -\frac{\Delta \text{energy}}{\text{Temperature}} \right) \]
\[ \text{else continue with } K[t]=K[t-1] \]
\[ \text{end.} \]

Simulated Annealing

Probably the most important application of the Metropolis algorithm in combinatorial optimization is simulated annealing.

The original motivation is to simulate the behaviour of molecules when a material is annealed into optimal crystal structure. Such an optimal structure represents the minimal energy state of this material.

The basic idea is to combine two components:
• an algorithm that finds the equilibrium at a given temperature (Metropolis)
• a cooling schedule that lowers the temperature and to repeatedly apply this algorithm while the cooling proceeds.

The cooling schedule is typically negative exponential

Simulated Annealing

We want to minimize a given function $f(x_1, ..., x_n) = f(x^*)$.

procedure simulated annealing

select a random point $x^*$;
$t := 0$; $T$=initial_temperature;

repeat

    select a random point $y^*$ in $N(x^*)$;  “the neighborhood of $x$”
    if $f(y^*) \leq f(x^*)$ then $x^* := y^*$;
    else if random[0,1) < exp( $-c/T \times (f(y^*)-f(x^*))$ );
    $t := t+1$;
    $T :=$ schedule($T$, $t$);

until ($t<final\_temperature$ or $f(x^*)<limit$)
end.

SA can never get trapped in a local minimum, because it can always (with some probability) jump to a better configuration.
Simulated Annealing

Important Parameters

- Neighbourhood function
- Initial temperature
- Cooling schedule \text{sched}(T,t)
  \begin{align*}
  \text{It is very important to cool slowly!}
  \end{align*}
  \text{(compare quenching in metallurgy)}
- Final temperature
- Limit

<table>
<thead>
<tr>
<th>Physics</th>
<th>Optimization</th>
</tr>
</thead>
<tbody>
<tr>
<td>state</td>
<td>candidate solution</td>
</tr>
<tr>
<td>state space transition</td>
<td>neighbourhood</td>
</tr>
<tr>
<td>energy</td>
<td>objective value</td>
</tr>
<tr>
<td>ground state</td>
<td>optimum</td>
</tr>
<tr>
<td>temperature</td>
<td>c/T</td>
</tr>
<tr>
<td>quenching</td>
<td>local search</td>
</tr>
</tbody>
</table>
Stochastic Hillclimber

An alternative is a general stochastic hill climber. Instead of accepting every downhill move and uphill moves with a probability depending on the energy difference, every move is accepted probabilistically.

```
procedure stochastic hill climber

select a random point x*;
t := 0; T=initial_temperature;
repeat

    select a random point y* in N(x*); "the neighbourhood of x"
    accept y* with probability 1/(1+exp( c*(f(y*)-f(x))/T )
    t := t+1;
    T := schedule(T, t);

until (t<final_temperature or f(x*)<limit)
end.
```
Stochastic Hill Climber vs. Simulated Annealing

Simulated Annealing

![Simulated Annealing graph](image)

Stochastic Hill Climbing

![Stochastic Hill Climbing graph](image)
A closely related meta-heuristics is Threshold Acceptance. The idea is to accept any modification of the current configuration unless it worsens the objective by more than a threshold that decreases over time.

```
procedure Threshold Acceptance Maximization

select a random point x*;
T := initial_threshold;
repeat
    select a random point y* in N(x*);
    if { f(y*) - f(x*) > - T } then accept x* := y*;
    if (no quality increase for more than max_try jumps)
        decrease threshold T ;
until (no change or maximum iterations reached)
end.
```

Though the convergence of TA towards an optimum is theoretically not guaranteed, in practice it sometimes performs even better than SA.
A third related meta-heuristics is Great Deluge. The idea is to accept any modification of the current configuration unless the new objective value is worse than an absolute threshold that is raised with time.

procedure Great Deluge Maximization

select a random point $x^*$;
$F := \text{initial\_flood\_level}$;
repeat
    select a random point $y^*$ in $N(x^*)$;
    if \{ $f(y^*) > F$ \} then accept $x^* := y^*$;
    increase flood level $F$;
until (no change or maximum iterations reached)
end.

Again, the convergence of GD is not guaranteed, but it can perform well in practice.
Constraints in Simulated Annealing

Constraint can be integrated in SA in two ways

- Modify the neighbourhood function to only include feasible points
- Penalize constraint violations

\[
\begin{align*}
\max / \min & \quad z = f(\bar{x}) = f(x_1, \ldots, x_n) \\
& \quad g_1(\bar{x}) = b_1 \\
\text{subject to} & \quad : \quad : \quad : \\
& \quad g_n(\bar{x}) = b_n
\end{align*}
\]

is transformed using penalty functions \(p_1, \ldots, p_n\)

\[
\begin{align*}
\max / \min & \quad f(\bar{x}) = f(x_1, \ldots, x_n) = \\
& \quad f(\bar{x}) + p_1(b_1 - g_1(\bar{x})) + \ldots + p_n(b_n - g_n(\bar{x}))
\end{align*}
\]
Penalty Methods

Typical choices for penalty functions are

• weighted boolean functions
  \( p_i(x) = 0 \) if \((x=0)\), \( c_i \) otherwise
• static multipliers (“degree of violation”)
  \( p_i(x) = c_i x \)

many other functions are possible.

The disadvantage of weighted boolean functions is that they make the objective landscape very rugged. This can also be true for static multipliers (depending on the types of constraints).
Adaptive / Dynamic Penalties

- Static penalties produce a very rugged landscape making SA inefficient.
- It is difficult to choose the penalties correctly.

A remedy is to adapt the penalties during the run time of the annealing. Instead of penalty function $p_i(x)$ we now have time dependent penalties

$$p_i(x, k) = c_i(k) \times (x^\alpha_i)$$

where for all $i$: $c_i(k) \leq c_i(k+1)$ and for at least one $i$: $c_i(k) < c_i(k+1)$.

This transforms the original problem into a series of progressively stricter optimization problems for $k=1...n$.

Again, it is very difficult to choose the penalty functions correctly.
Review: Lagrange Multipliers

To solve a constraint NLP

$$\text{max/min } z = f(\vec{x}) = f(x_1, \ldots, x_n)$$

$$g_i(\vec{x}) = b_i$$

subject to

$$g_n(\vec{x}) = b_n$$

Associate multipliers $\lambda$ with the constraint and solve instead:

$$\text{max/min } f(\vec{x}, \vec{\lambda}) = f(x_1, \ldots, x_n, \lambda_1, \ldots, \lambda_n) =$$

$$f(\vec{x}) + \lambda_1 (b_1 - g_1(\vec{x})) + \ldots + \lambda_n (b_n - g_n(\vec{x}))$$
Interpreting the Lagrangian

\[
\max/min \quad f(\bar{x}, \bar{\lambda}) = f(x_1, \ldots, x_n, \lambda_1, \ldots, \lambda_n) = \\
\quad f(\bar{x}) + \lambda_1 (b_1 - g_1(\bar{x})) + \ldots + \lambda_n (b_n - g_n(\bar{x}))
\]

- At an extremum \((\bar{x}_e, \bar{\lambda}_e)\) of \(f(\bar{x}, \bar{\lambda})\) we must have \(\frac{\partial f(\bar{x}, \bar{\lambda})}{\partial \lambda_i} = 0\) for all \(i\) because all partial derivatives must be zero.

- Therefore the extremum is a feasible point because it fulfills for all \(i\)

\[
\frac{\partial f(\bar{x}, \bar{\lambda})}{\partial \lambda_i} = b_i - g_i(\bar{x}) = 0
\]

- It also follows that since for all \(i\) : \(b_i - g_i(\bar{x}_e) = 0\) at the extremum we have \(f(\bar{x}, \bar{\lambda}) = f(\bar{x})\) for any \(\lambda\).

- Therefore \(f(\bar{x}_e)\) is a solution of the original constraint problem.
Finding the Extremum

<table>
<thead>
<tr>
<th>max/min</th>
<th>$f(\bar{x}, \bar{\lambda}) = f(x_1, \ldots, x_n, \lambda_1, \ldots, \lambda_n) =$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$f(\bar{x}) + \lambda_1 (b_1 - g_1(\bar{x})) + \ldots + \lambda_n (b_n - g_n(\bar{x}))$</td>
</tr>
</tbody>
</table>

We know that at the extremum of $f(\bar{x}, \bar{\lambda})$ we have

$$\frac{\partial f(\bar{x}, \bar{\lambda})}{\partial x_1} = \ldots = \frac{\partial f(\bar{x}, \bar{\lambda})}{\partial x_k} = \frac{\partial f(\bar{x}, \bar{\lambda})}{\partial \lambda_1} = \ldots = \frac{\partial f(\bar{x}, \bar{\lambda})}{\partial \lambda_n} = 0$$

This tells us how to find the extremum of the Lagrangian, but it is only a necessary, not a sufficient condition.

Remember saddle points!!!
**Constraint Simulated Annealing**

CSA is a multiplier-based method for solving constraint NLPs

\[
\text{max/min} \quad f(\bar{x}) = f(x_1, \ldots, x_n) \\
\quad h_i(\bar{x}) \; = \; 0 \\
\text{subject to} \quad : \quad : \\
\quad h_n(\bar{x}) \; = \; 0
\]

by instead optimizing \( L(x^*, \lambda^*) \) for some continuous non-negative transformation \( g \) with \( g(x) = 0 \) iff \( x = 0 \):

\[
L(\bar{x}, \tilde{\lambda}) = L(x_1, \ldots, x_n, \lambda_1, \ldots, \lambda_n) = f(\bar{x}) + \lambda_1 \cdot g(h_1(\bar{x})) + \ldots + \lambda_n \cdot g(h_n(\bar{x}))
\]

Theory is based on discrete space

- differentiation interpretation of real Lagrange multipliers cannot be applied

CSA Theorem

**Def:** A saddle point is a point \((x^*, \lambda^*)\) if for all \(x\) in the discrete neighbourhood \(N(x^*)\) and for all \(\lambda'\): \(L(x^*, \lambda') \leq L(x^*, \lambda^*) \leq L(x, \lambda^*)\)

“The first inequality is true if all constraints are satisfied, the second part guarantees that \(x^*\) is a local minimum, i.e. for all neighbours \(x\) of \(x^*\) either \(f(x)\) is worse or constraints are violated.”

**Lemma:** If \(h_i(x) \geq 0\) for all \(i\) then for any constrained local minimum \(x^*\) in discrete space there exists a finite \(\lambda^*\) such that for all \(\lambda' \geq \lambda^*\): \((x^*, \lambda')\) is a saddle point.

“In discrete space we can always find a multiplier vector so that for every minimum we have a saddle point.

Idea: For every minimum \(x^*\) we can enumerate its neighbours and choose a large enough \(\lambda\) so that no matter how small the violation in the neighbourhood it, it compensates any Improvement in \(f(x)\).”

**Theorem:** In discrete space, if \(g(x)\) is a non-negative continuous transformation with \(g(x)=0\) if and only if \(x=0\) then the set of saddle points and the set of constrained local minima are the same.

This is a strong theorem, because it gives necessary and sufficient conditions.
procedure CSA
    set starting point $X^*=(x^*, \lambda^*)$;
    set starting temperature $T=T_0$;
    while (not stopped) do
        for $k$ from 1 to $num_{\text{trials}}$ do
            generate $X'$ from $N(X^*)$;
            accept $X'$ as new $X^*$ with probability $P(X^*, X', T)$;
        end;
        reduce temperature $T$;
    end;
end.

The neighbourhood function and acceptance probability must be chosen differently from conventional SA.
Neighbourhood and Acceptance

For minimization perform a probabilistic descent in x space and a probabilistic ascent in λ space:

\[
P(X, X', T) = \begin{cases} 
1 & \text{if } L(X') < L(X) \text{ and } x \text{ has changed} \\
1 & \text{if } L(X') > L(X) \text{ and } \lambda \text{ has changed} \\
\exp(-[L(X') - L(X)]/T) & \text{if } L(X') > L(X) \text{ and } x \text{ has changed} \\
\exp(-[L(X) - L(X')]/T) & \text{if } L(X') < L(X) \text{ and } \lambda \text{ has changed}
\end{cases}
\]

As long as a constraint is satisfied, do not change its multiplier:

\[
N((x^*, \lambda^*)) = \{(x', \lambda^*) \mid x' \in N_1(x^*) \} \cup \{(x^*, \lambda') \mid \lambda' \in N_2(\lambda^*) \}
\]

where \(N_1\) is a usual SA neighbourhood and

\[
N_2(\lambda^*) = \{\mu^* \mid \mu^* < \lambda^* \land \mu_i = \lambda_i \text{ if } h_i(x^*) = 0\} \\
\cup \{\mu^* \mid \mu^* > \lambda^* \land \mu_i = \lambda_i \text{ if } h_i(x^*) = 0\}
\]
Simulated Annealing in Graph Layout

Graph Layout has successfully applied to graph layout problems for general undirected graphs: this is a hard optimization problem.

Neighbourhood function
Let a single node jump randomly on a circle around its current position
The maximum jump distance decreases with the temperature

Objective function

\[ f(g) = p_1 \times \#\text{crossings}(g) + \]
\[ p_2 \times 1/\text{std-deviation-edge-length}(g) + \]
\[ p_3 \times \sum(n_1, n_2 \text{ in Nodes}) 1/\text{dist}(n_1, n_2) + \]
\[ p_4 \times \sum(n \text{ in Nodes}) 1/\text{border-distance}(n) + \]
\[ p_5 \times \sum(e \text{ in Edges}) \text{length}(e) + \]
\[ p_6 \times \sum(n \text{ in Nodes, e in Edges}) 1/\text{dist}(n, e) \]
SA Example Animation
Penalty Dependencies

The choice of penalties is crucial for
• quality of final layout.
• efficiency of the method
because it determines the ruggedness of the objective function.

Consider, for example, the edge crossing component which is not a smooth (or continuous) function. For smooth objective functions, SA is relatively more efficient.

Variations Depending on Penalties
More SA-GD Examples
Tabu Search

Deterministic search
Adaptive memory search

“learn about useful/useless spots and areas in the search space”

Diversification
*discourage moves similar to search history*

Intensification
*encourage moves in known promising areas*

Simplest Variant
perform local search but do not revisit any candidate for $n$ steps
Tabu Search Algorithm

procedure Tabu Search

select initial (random) solution \( x \);
set current best solution \( B := x \);
set tabu memory := empty;

repeat

select the best solution \( y \) in \( N(x)-N_1(x)+N_2(x) \);
if \{ cost(y) better than cost(x) \} then \( B := y \);
update tabu memory;
update other memories ;
\( x := y \);
until (maximum iterations reached)
end.

- \( N(x) \) standard neighbourhood of candidate
- \( N_1(x) \) excluded neighbourhood dependent on tabu memory
- \( N_2(x) \) extended neighbourhood dependent on aspiration criteria
Tabu Memory

- Explicit memory records complete candidate solutions
- Attribute memory records particular properties of candidate solutions

Example G-SAT: \((x \lor y \lor z) \land (x \lor z) \land (y \lor z) \land \neg x\)

\((x, y, z) \mapsto (0, 1, 1)\)

- explicit memory contains complete valuations

\((x, y, z) \mapsto (0, *, *)\)

- attribute memory contains valuation of particular variable
Recency-based Memory / Tabu List

• records the $n$ most recently visited candidates and/or actions
• “short term” memory
• these are typically tabu (i.e. excluded from the search)

Example G-SAT \[(x \lor y \lor z) \land (x \lor z) \land (y \lor z) \land \neg x \land y\]

Neighbourhood: obtained by flipping a single value

Recency memory ($n=2$)

how long ago did last flip for an element happen?

\[
\begin{align*}
(1,0,0) & \Rightarrow (0,0,0) \\
(0,0,0) & \Rightarrow (1,0,0) \\
(0,0,1) & \Rightarrow (2,0,1) \\
(0,1,1) & \Rightarrow (0,1,2)
\end{align*}
\]
Aspiration Criteria

• used to override tabu criteria for particularly good candidates
• if at step $n$ a candidate in $N(x)$ is tabu but better than the best candidate found so far, accept it anyway
• extends the search neighbourhood $N_2(x)$
• many different forms of aspiration criteria used
Frequency-based Memory

- records how often a candidates and/or action was encountered during the last $h$ iterations (event horizon)
- “long term” memory
- typically used for encouraging “neglected” possibilities.

Example G-SAT

$$(x \lor y \lor z) \land (x \lor z) \land (y \lor z) \land \neg x \land y$$

$$
\begin{align*}
(1,0,0) & \Rightarrow (0,0,0) \\
(0,0,0) & \Rightarrow (1,0,0) \\
(0,0,1) & \Rightarrow (1,0,1) \\
(0,1,1) & \Rightarrow (1,1,1) \\
(1,1,1) & \Rightarrow (2,1,1)
\end{align*}
$$

usage example:
- If all non-tabu moves lead to inferior solutions (stuck in local maximum) but some of the tabu moves lead to better solutions (which still do not aspire)
- then evaluate the quality of the improving tabu solutions as

$$
\text{revised-quality}(x) := \text{quality}(x) - c \times \text{frequency(move to x)}
$$

and choose best move according to this.
Quality-based Memory
- Elitist Memory -

• records the $n$ best candidates found so far
• typically explicit memory
• "long term" memory
• the immediate neighbourhood of elements in the elite memory can be used to extend the search neighbourhood $N_2(x)$
In a given edge-weighted graph find the tree with k edges of minimum cost ( = sum of edge weights )

Initial Greedy Solution

![Graph with nodes and edges, and a table showing steps, candidates, selections, and total weights.]

<table>
<thead>
<tr>
<th>Step</th>
<th>Candidates</th>
<th>Selection</th>
<th>Total Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(1,2)</td>
<td>(1,2)</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>(1,4), (2,3)</td>
<td>(1,4)</td>
<td>26</td>
</tr>
<tr>
<td>3</td>
<td>(2,3), (3,4), (4,6), (4,7)</td>
<td>(4,7)</td>
<td>34</td>
</tr>
<tr>
<td>4</td>
<td>(2,3), (3,4), (4,6), (6,7), (7,8)</td>
<td>(6,7)</td>
<td>40</td>
</tr>
</tbody>
</table>
Neighbourhood Generation

Exchange a tree edge against a non-tree edge maintaining tree

- static swap: the set of tree nodes remains unchanged
- dynamic swap: the set of tree nodes changes
Tabu Criteria

- Tabu Criteria:
  - added edges must not be dropped again within the next $k_1$ iterations
  - dropped edges must not be added again within the next $k_2$ iterations
  - A move is tabu if either its or its actions is tabu

- Asymmetric Tabu Tenure: $k_2=2$, $k_1=1$
  emphasizes active exploration,
  because there are more non-tree than tree edges

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Tabu-active net tenure</th>
<th>Add</th>
<th>Drop</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>(4,6)</td>
<td>(4,7)</td>
</tr>
<tr>
<td>2</td>
<td>(4,6)</td>
<td>(4,7)</td>
<td>(6,8)</td>
<td>(6,7)</td>
</tr>
<tr>
<td>3</td>
<td>(6,8), (4,7)</td>
<td>(6,7)</td>
<td>(8,9)</td>
<td>(1,2)</td>
</tr>
</tbody>
</table>

First three iterations from initial greedy solution.

*Note that the total cost of the candidates increases.*
Exclusion of Tabu Moves

Note that in iteration 2 a sub-optimal move must be taken because edge (4,7) is tabu.
Further Iterations

The numbers $i^*$ indicate the remaining tabu tenure for the labelled edges.
Further Iterations

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Tabu-active net tenure</th>
<th>Add</th>
<th>Drop</th>
<th>Move Value</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>(6,8), (4,7)</td>
<td>(6,7)</td>
<td>(8,9)</td>
<td>(1,2)</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>(6,7), (8,9)</td>
<td>(1,2)</td>
<td>(4,7)</td>
<td>(1,4)</td>
<td>-17</td>
</tr>
<tr>
<td>5</td>
<td>(1,2), (4,7)</td>
<td>(1,4)</td>
<td>(6,7)</td>
<td>(4,6)</td>
<td>-9</td>
</tr>
<tr>
<td>6</td>
<td>(1,4), (6,7)</td>
<td>(4,6)</td>
<td>(6,9)</td>
<td>(6,8)</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>(4,6), (6,9)</td>
<td>(6,8)</td>
<td>(8,10)</td>
<td>(4,7)</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>(6,8), (8,10)</td>
<td>(4,7)</td>
<td>(9,12)</td>
<td>(6,7)</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>(4,7), (9,12)</td>
<td>(6,7)</td>
<td>(10,11)</td>
<td>(6,9)</td>
<td>-7</td>
</tr>
<tr>
<td>10</td>
<td>(6,7), (10,11)</td>
<td>(6,9)</td>
<td>(5,9)</td>
<td>(9,12)</td>
<td>7</td>
</tr>
</tbody>
</table>

Note that local optima are achieved in step 5 and 9, but TS escapes again.
Development of Best Solution

If the development of the current weight starts to stagnate or falls below a minimal improvement rate during a number of iterations, it is recommendable to restart TS from a new initial solution.
Simulated Annealing vs. Tabu Search

<table>
<thead>
<tr>
<th>SA</th>
<th>Tabu</th>
</tr>
</thead>
<tbody>
<tr>
<td>anytime</td>
<td>anytime</td>
</tr>
<tr>
<td>stochastic</td>
<td>deterministic</td>
</tr>
<tr>
<td>downhill from anywhere</td>
<td>downhill from local optimum only</td>
</tr>
<tr>
<td>(in maximization)</td>
<td>(in maximization)</td>
</tr>
<tr>
<td>no memory</td>
<td>adaptive memory</td>
</tr>
<tr>
<td>static neighbourhood</td>
<td>dynamic neighbourhood</td>
</tr>
<tr>
<td>random sampling</td>
<td>responsive exploration</td>
</tr>
<tr>
<td>parameterized</td>
<td>class of methods</td>
</tr>
<tr>
<td>some tailoring required</td>
<td>much tailoring required</td>
</tr>
</tbody>
</table>

“bad strategic choice can yield more information than a good random choice”

Summary

In this section we have looked at

• Local Search
• Heuristics
• Stochastic Constraint Solving & Optimization
  • Simulated Annealing
  • Constraint Simulated Annealing
• Tabu Search