Metaheuristics in Scheduling
Local Search and Genetic Algorithms

Tim Nieberg
Metaheuristics in Scheduling

- *NP*-hard scheduling problem $\rightarrow$ Brand-and-Bound?
  - ... usually only valid for small instances!
- *NP*-hard scheduling problem $\rightarrow$ Approximation?
  - ... does not work in general!
- *NP*-hard scheduling problem $\rightarrow$ Heuristic?
  - ... is there a general approach to design a non-trivial heuristic?

We discuss two general techniques for solving optimization problems heuristically.
Recap: \textit{Discrete Optimization Problem} (Minimization)

A \textit{(Discrete) Optimization Problem} is given by its problem description $\Pi = (\mathcal{I}, \mathcal{S})$, where

- $\mathcal{I}$ is the set of instances, and
- $\mathcal{S}(x)$ is the (discrete) set of feasible solutions for an instance $x \in \mathcal{I}$,

together with an objective function $f : \mathcal{S}(x) \rightarrow \mathbb{R}$ that evaluates each feasible solution.

We then seek—given an instance $x$—a feasible solution $y \in \mathcal{S}(x)$ with minimum objective function value.
Suppose we are given an instance $x \in \mathcal{I}$.

- $S = S(x)$ is a discrete set

Local Search is an iterative procedure that moves from one solution in $S$ to the next (until some stopping criterion is satisfied).

- ... think of it as the discrete analogy of hill-climbing
In order to move systematically through the solution set, the possible moves from one solution to another is restricted by

- *neighborhood structures* $\mathcal{N} : S \rightarrow 2^S$.

For each solution $s \in S$, the set $\mathcal{N}(s)$ describes the subset of solutions which can be reached from $s$ in the next step.

- $\mathcal{N}(s)$ is called *neighborhood of $s$*
A neighborhood structure $\mathcal{N}$ may be represented by a directed graph $G = (V, A)$ where

- $V = S$
- $(u, v) \in A \iff v \in \mathcal{N}(u)$.

This graph is called the *neighborhood graph*.

Note: in general, it is not possible to store this graph completely!

- $S$ usually has exponential size w.r.t. the instance.
In order to avoid problems with size of the neighborhood graph, a neighborhood is usually described by operators:

- Let $F : S \rightarrow S$ be a function,
- for each feasible $s \in S$, $F(s)$ is a subset consisting only of feasible solutions, we call $F$ thus an allowed modification.

For every $s \in S$, we can define a neighborhood structure for a set $AM$ of allowed modifications as follows

$$\mathcal{N}(s) := \{F(s) \mid F \in AM\}.$$
Suppose the neighborhood graph $G = (V, A)$ is connected

- for every (starting) solution $s \in \mathcal{S}$, there exists a directed path to every other solution in $\mathcal{S}$.
- In particular, we can provide a sequence of operations to $s$ that result in an optimal solution $s^* \in \mathcal{S}$.
Suppose the neighborhood graph $G = (V, A)$ is connected

- for every (starting) solution $s \in S$, there exists a directed path to every other solution in $S$.

- In particular, we can provide a sequence of operations to $s$ that result in an optimal solution $s^* \in S$.

Well, this is usually overkill!
Suppose the neighborhood graph $G = (V, A)$ is connected

- $\Rightarrow$ for every (starting) solution $s \in S$, there exists a directed path to every other solution in $S$.

- In particular, we can provide a sequence of operations to $s$ that result in an optimal solution $s^* \in S$.

Well, this is usually overkill!

We only need the latter condition:

- A neighborhood $\mathcal{N}$ is called \textit{OPT-connected} if, from each solution $s \in S$, an optimal solution can be reached by a finite sequence $s, s_1, \ldots, s_k, s_{k+1}$ of solutions $s_i \in S$ s.t. $s_{i+1} \in \mathcal{N}(s_i)$ for $i = 1, \ldots, k$, and $s_{k+1}$ optimal.
Example: Exchange Neighborhood

Recall: many (active) schedule are completely described by permutations.

**Swap Neighborhood** for a permutation $\pi$:

- Swap two adjacent elements in $\pi$.

**Exchange Neighborhood** for a permutation $\pi$:

- New permutation $\pi'$ with

$$
\pi'(a) := \pi(b) \land \pi'(b) := \pi(a)
$$

for two indexes $a$ and $b$.

(In order to obtain an allowed modification, we need to restrict all permutations to feasibility, this is problem specific!)
Given a solution $s \in S$,

- in each iteration, we choose a solution $s' \in \mathcal{N}(s)$ (or the allowed modification that yields $s'$), and
- based on the objective function values $f(s)$ and $f(s')$, we choose a starting solution for the next iteration.

According to different criteria for the choice of the next solution, different types of local search methods emerge.

$\mathcal{N}$ OPT-connected $\Rightarrow$ independent of starting solution, we are able to reach optimality.

$\mathcal{N}$ not OPT-connected $\Rightarrow$ may happen that we are unable to even reach an optimal solution at all.
For a local search approach, the simplest choice is to always take a neighboring solution with smallest objective value.

Algo.: Iterative Improvement

1. Generate initial solution $s \in S$
2. WHILE $\exists s' \in \mathcal{N}(s) \mid f(s') < f(s)$ DO
   1. Choose best solution $s' \in \mathcal{N}(s)$;
   2. $s := s'$;
For a local search approach, the simplest choice is to always take a neighboring solution with smallest objective value.

**Algo.: Iterative Improvement**

1. Generate initial solution $s \in S$
2. WHILE $\exists s'' \in N(s) \mid f(s') < f(s)$ DO
   1. Choose best solution $s' \in N(s)$;
   2. $s := s'$;

... terminates with *local minimum* $s^*$ w.r.t. neighborhood $N$

- ... we could start the algorithm several times with different solutions
- ... we could also accept solutions with increasing objective value
  - need strategies to avoid cycling!
Simulated Annealing

**Idea:** avoid cycling by randomization, i.e. simulate the annealing process from physics

- choose a solution \( s' \in S \) randomly
- accept solution only with a certain probability

In the \( i \)-th iteration, \( s' \) is accepted with probability

\[
\min \{1, e^{\frac{f(s') - f(s)}{t_i}} \}
\]

where \( (t_i) \) is a sequence of positive control values with

\[
\lim_{i \to \infty} t_i = 0.
\]
Simulated Annealing

**Alg.: Simulated Annealing**

1. \( i := 0 \)
2. Generate initial solution \( s \in S \)
3. \( \text{best} := f(s) \)
4. **REPEAT**
   1. Generate randomly a solution \( s' \in \mathcal{N}(s) \)
   2. IF \( \text{Rand}(0,1) < \min\{1, e^{\frac{f(s') - f(s)}{t_i}}\} \) THEN
      1. \( s := s' \)
      2. IF \( f(s') < \text{best} \) THEN
         3. \( s^* := s \)
         4. \( \text{best} := f(s') \)
   5. \( i := i + 1 \)
5. **UNTIL** some stopping condition is satisfied
often, \((t_i)\) is defined (in analogy to physics) as

\[ t_{i+1} := \alpha t_i, \quad 0 < \alpha < 1 \]

search may be stopped after number of iterations, certain number of non-improving solutions, time limit, ...

Other variant of S.A. is given by \textit{Threshold Acceptance}, where

- acceptance rule for \(s' \in \mathcal{N}\) is accepted if difference \(f(s) - f(s')\) is within some limit \(l_i\)
- \(l_i\) also decreases with number of iterations
Another *deterministic* strategy to avoid cycling is to store all visited solutions in a so-called *tabu-list* $T$.

$\Rightarrow$ a neighbor is only accepted if it is not contained in $T$. 
Algo.: Tabu Search

1. Generate initial solution $s \in S$
2. $best := f(s); s^* := s$
3. $T := \emptyset$
4. **REPEAT**
   1. $Cand(s) := \{s' \in N(s) \mid \text{move from } s \text{ to } s' \text{ is not tabu }\}$
   2. Choose a solution $s' \in Cand(s)$
   3. Update $T$
   4. $s := s'$
   5. IF $f(s') < best$ THEN
   6. $s^* := s$
   7. $best := f(s')$
5. **UNTIL** some stopping condition is satisfied
Another *deterministic* strategy to avoid cycling is to store all visited solutions in a so-called *tabu-list* $T$.

- $\Rightarrow$ a neighbor is only accepted if it is not contained in $T$
- ... due to memory constraints, this may not be possible!
- $T$ may contain only the $|T| \leq B$ visited solutions
  - only cycles of length greater than $B$ may occur
  - if $B$ sufficiently large, the probability of cycling becomes small
- $T$ may not contain complete solution descriptions, but only *attributes* of already visited solutions
  - all solutions having one of the stored attributes are tabu
  - solution will not be re-visited as long as its attributes are stored in $T$
- Disadvantage: also new solutions may be declared tabu!
  - *aspiration criteria*: accept solution even if they are tabu
  - e.g. based on objective function value
Algo.: Tabu Search

1. Generate initial solution $s \in S$
2. $best := f(s); s^* := s$
3. $T := \emptyset$
4. REPEAT
   1. $Cand(s) := \{s' \in \mathcal{N}(s) \mid \text{move from } s \text{ to } s' \text{ is not tabu OR } s' \text{ satisfies the aspiration criterion } \}$
   2. Choose a solution $s' \in Cand(s)$
   3. Update $T$
   4. $s := s'$
   5. IF $f(s') < best$ THEN
      6. $s^* := s$
      7. $best := f(s')$
5. UNTIL some stopping condition is satisfied
Neighbor Selection

Depending on the size of the neighborhood, several selection strategies for \(Cand(s)\) emerge:

- **best-fit**: explore entire neighborhood and take best neighbor
- **first-fit**: explore neighborhood and take first neighbor that improves current solution
  - if no such neighbor exists, take the best one from \(Cand(s)\)
- ...


For tabu-list management, two types are distinguished. 

*static* tabu-lists

- constant size

*dynamic* tabu-lists

- variable length
  - if a solution is found that improves the current leader, the list is emptied as we have never visited this part of the solution space before
  - improving phase of T.S.: *decrease* length of list
  - non-improving phase: *increase* length of list

Generally speaking, a tabu-list serves a *short-term* memory of the local search procedure.
Besides short-term memory (T.L.), also long-term memory may be kept that is used for diversification. Here, properties of promising solutions not explored further are stored which are then used in a restarting phase:

- If within a certain number of iterations, the current leader is not improved (intensification), then
- the search process is stopped and restarted with a new solution (diversification).

Note: a restart from a randomly generated solution would neglect all information of the previous search process.
Arriving at a local search algorithm for a specific problem:
- Define problem specific ingredients of local search:
  - most importantly: the neighborhood
  - Tune the chosen local search approach.

Claim:
The *problem specific ingredients* are far more important than the *tuning*. 
Efficiency of Local Search

Local efficiency (one iteration):

- quality of $s'$ or $\mathcal{N}(s)$
- computational time to calculate and evaluate $s'$
- size of $\mathcal{N}(s)$

Note: large size of neighborhood needs not result in large computational time (see c.f. research on VLSN: efficient search for optimal solution w.r.t. neighborhood)

Global efficiency:

- number of iterations, computational time
- quality of final solution
  - related to price of anarchy (game theory)
Recap: Jop-Shop Problem $J||C_{\text{max}}$

- $n$ jobs $j = 1, \ldots, n$ consisting of $n_j$ operations
- $m$ machines
- each operation $O_{ij}$ has machine $\mu_{ij}$ and processing time $p_{ij}$

Recap: Disjunctive Graph Model and Complete Selections

- complete selection $\rightarrow$ all arcs in model fixed
  - cycle-free $\iff$ feasible solution
- ... can use permutation of operations per machine to describe complete selection uniquely
- ... can use longest path calculation to determine starting time of each operation (critical path)

Use $\pi = (\pi_1, \ldots, \pi_m)$ to describe the set $S$ of solutions.
Apply the Swap-Neighborhood approach based on the following lemma.

**Lemma:** Let \( s \) be a complete selection, and let \( P \) be a longest path in \( G(s) \). Let \((v, w)\) be an arc of \( P \) such that \( v \) and \( w \) are processed on the same machine. Then, \( s' \) obtained by reversing \( v \) and \( w \) is again a complete selection. (Proof on the board)

We call the resulting neighborhood \( \mathcal{N}_1 \).
Apply the Swap-Neighborhood approach based on the following lemma.

**Lemma:** Let \( s \) be a complete selection, and let \( P \) be a longest path in \( G(s) \).
Let \((v, w)\) be an arc of \( P \) such that \( v \) and \( w \) are processed on the same machine. Then, \( s' \) obtained by reversing \( v \) and \( w \) is again a complete selection. \((\Rightarrow \mathcal{N}_1)\)

**Theorem:** \( \mathcal{N}_1 \) is OPT-connected.
(Proof on the board)
Consider a feasible solution, i.e. complete selection, $s = \pi$.

**Definition (Block):** Let $G(s) = (V, C \cup D)$ bet the graph induced by the complete selection $s$, and let $P$ be a critical path in $G(s)$. A sequence $u_1, \ldots, u_k$ of successive nodes in $P$ is called *block* if the following two properties hold:

1. (i) The sequence contains at least two nodes.
2. (ii) The sequence represents a maximal number of operations to be processed on the same machine.

We denote the $j$-th block on a given critical path $P$ by $B_j$. 
Lemma: Let $s$ be a complete selection corresponding to a feasible solution for the job-shop problem. If there exists another selection $s'$ such that $L(s') < L(s)$ holds, then in $s'$ at least one operation from some block $B$ of $G(s)$ has to be processed before the first or after the last operation of $B$.

(Proof on the board.)
Lemma: Let $s$ be a complete selection corresponding to a feasible solution for the job-shop problem. If there exists another selection $s'$ such that $L(s') < L(s)$ holds, then in $s'$ at least one operation from some block $B$ of $G(s)$ has to be processed before the first or after the last operation of $B$.

Consequence: $s, s'$ with $L(s') < L(s)$, then one of the following holds

- at least one operation of one block $B$ in $G(s)$, different from the first operation in $B$, has to be processed before all other operations of $B$ in the schedule given by $G(s')$
- at least one operation of one block $B$ in $G(s)$, different from the last operation in $B$, has to be processed after all other operations of $B$ in the schedule given by $G(s')$
Consider \((u, v)\) on critical path w.r.t. \(s\).

Disadvantage of \(N_1\):

- If \((u, v)\) belong to a block, but do not contain first or last operation of this block, no improvement occurs.
- \(\Rightarrow\) generally, several moves are needed to improve solution

Let \((v, w)\) be processed on the same machine, and denote by \(PM(v)(SM(w))\) the immediate predecessor (successor) of \(v(w)\) (if exists).

Consider as moves all permutations of \(\{PM(v), v, w\}\) and \(\{v, w, SM(w)\}\) where \((v, w)\) is reversed and that are feasible \((N_2)\).

Clearly, \(N_1 \subseteq N_2 \Rightarrow N_2\) is OPT-connected
Directly using the block-lemma, we obtain: $\mathcal{N}_3$ is defined as the neighborhood, where operations of a block are shifted at the beginning or the end of the respective block. 

**open question:** Is $\mathcal{N}_3$ OPT-connected?
\( \mathcal{N}_3 \) can be extended to a neighborhood \( \mathcal{N}_4 \) which is OPT-connected in the following way:

- Let \( P \) be a critical path in \( G(s) \).
- \( s' \) is derived from \( s \) by moving one operation \( j \) of a block \( B \) of \( P \) different from the first (last) operation in \( B \) before (after) all operations of \( B \) (if feasible).
- Otherwise (i.e. above not feasible), \( j \) is moved to the position inside \( B \) closest to first (last) operation, that is still feasible.

Note: \( \mathcal{N}_3 \subseteq \mathcal{N}_4 \)

**Lemma:** \( \mathcal{N}_4 \) is OPT-connected
(proof on the board)
Organization of the Tabu-List

$\mathcal{N}_1$ up to $\mathcal{N}_4$ work by reversing an arc $(v, w)$ in $G(s)$:
- attribute = arc reversed by recent moves
- a solution is defined to be tabu if an arc belonging to the attribute set is contained in it

As supporting data-structure, we use a matrix $A = (a_{ij})$:
- $a_{ij} =$ count of the iteration in which arc $(i, j)$ was last reversed
- we forbid a swap of $(i, j)$ if the count + length of the tabu-list is greater than the current iteration

$\Rightarrow$ the tabu-list length can be arbitrarily chosen (memory does not increase with length)

On $A$, we use a dynamic tabu-list management:
- improving phase: decrease length of list
- non-improving phase: increase length of list

also include an aspiration criterion, e.g. based on a lower bound
Genetic Algorithms

*Genetic Algorithm*
- general search technique inspired by *biological evolution*
  - ‘survival of the fittest’
- work on a set $POP$ of solutions (*population*)
  - instead of single solution as in local search
- single solution $s \in POP$ is called *chromosome*
  - usually encoded by a sequence of symbols (DNA)
- for each feasible solution $s$, $fit(s)$ is a measure of adaption (*fitness value*)
  - $fit(s)$ is often related to the objective function $f(s)$
Starting from an initial population, 'parent' solutions are selected and new 'child' solutions are created by genetic operators:

- **crossover**
  - mix subsequences of parent chromosomes

- **mutation**
  - perturbate a chromosome

Size of the population is controlled by fitness value.
Algo.: *Genetic Algorithm*

1. Generate initial population $POP$
2. Compute fitness of each individual $s \in POP$
3. REPEAT
   1. Choose two parent solutions $s^M, s^F \in POP$
   2. Create a child solution $s^C$ from $s^M, s^F$ by crossover
   3. Mutate $s^C$ with certain probability
   4. Compute fitness of $s^C$
   5. Add $s^C$ to $POP$ and reduce $POP$ by selection
4. UNTIL some stopping criterion is satisfied
Different variations are also possible

- several children may be generated simultaneously
- population may be divided into two sets
  - matching and creation of two new children $\rightarrow$ new population
- mutations may be realized by local search

In this approach, also infeasible solutions (chromosomes) may be present, this can be encoded in the fitness value.