Metaheuristics in Scheduling
Local Search and Genetic Algorithms

Tim Nieberg

Metaheuristics in Scheduling

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

We discuss two general techniques for solving optimization problems heuristically.

Recap: *Discrete Optimization Problem* (Minimization)
A *(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 $f : \mathcal{S}(x) \to \mathbb{R}$ 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.

Basic Structure of Local Search

Suppose we are given an instance $x \in \mathcal{I}$.

- $\mathcal{S} = \mathcal{S}(x)$ is a discrete set

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

- ... think of it as the discrete analogy of *hill-climbing*

Neighborhoods on the Solution Space

In order to move systematically through the solution set, the possible moves from one solution to another is restricted by

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

For each solution $s \in \mathcal{S}$, 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

Neighborhood Graph

A neighborhood structure $\mathcal{N}$ may be represented by a directed graph $G = (V, A)$ where

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

This graph is called the *neighborhood graph*.

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

- $\mathcal{S}$ usually has exponential size w.r.t. the instance.

Allowed Modifications

In order to avoid problems with size of the neighborhood graph, a neighorhood is usually described by *operators*:

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

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

$$\mathcal{N}(s) := \{F(s) \mid F \in AM\}.$$

Connectivity of the Neighborhood Graph

Suppose the neighborhood graph $G = (V, A)$ is connected

- $\Rightarrow$ 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 $s^* \in \mathcal{S}$.

Connectivity of the Neighborhood Graph

Suppose the neighborhood graph $G = (V, A)$ is connected

- $\Rightarrow$ 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 $s^* \in \mathcal{S}$.

Well, this is usually overkill!

Connectivity of the Neighborhood Graph

Suppose the neighborhood graph $G = (V, A)$ is connected

- $\Rightarrow$ 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 $s^* \in \mathcal{S}$.

Well, this is usually overkill!
We only need the latter condition:

- A neighborhood $\mathcal{N}$ is called *OPT-connected* if, from each solution $s \in \mathcal{S}$, an optimal solution can be reached by a finite sequence $s, s_1, \ldots, s_k, s_{k+1}$ of solutions $s_i \in \mathcal{S}$ 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 π:

- Swap two adjacent elements in π.

Exchange Neighborhood for a permutation π:

- New permutation π' with

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

  for two indexes a and b.

(In order to obtain an allowed modification, we need to restrict all permutations to feasiblity, this is problem specific!)

Local Search Method

Given a solution $s \in \mathcal{S}$,

- 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.

Iterative Improvement

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

Iterative Improvement

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

... terminates with *local minimum* s^* w.r.t. neighborhood $\mathcal{N}$

- ... 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 $s' \in \mathcal{S}$ randomly
- accept solution only with a certain probability

In the i-th iteration, s' is accepted with probablity

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

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

$$\lim_{i \to \infty} t_i = 0.$$

Simulated Annealing

Algo.: *Simulated Annealing*

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

Threshold Acceptance

- often, (t_i) is defined (in analogy to physics) as

$$t_{i+1} := \alpha t_i, 0 < \alpha < 1$$

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

Other variant of S.A. is given by *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

Tabu Search

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

Tabu Search

Algo.: *Tabu Search*

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

Tabu Search

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

Tabu Search

Algo.: *Tabu Search*

1. Generate initial solution $s \in \mathcal{S}$
2. $best := f(s); s^* := s$
3. $T := \varnothing$
4. REPEAT
    1. $Cand(s) := \{s' \in \mathcal{N}(s) \mid$ move from s to s' is not tabu OR s' 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. $\quad s^* := s$
    7. $\quad best := f(s')$
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 $Cand(s)$
- ...

Tabu List Management

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.

Diversification

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.

Application of Local Search

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)

Applying Tabu Search to the Job Shop Problem

Recap: Jop-Shop Problem $J||C_{\max}$

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

Recap: Disjunctive Graph Model and Complete Selections

- complete selection $\longrightarrow$ all arcs in model fixed
  - cycle-free $\Longleftrightarrow$ 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 $\mathcal{S}$ of solutions.

TS-JS: Neighborhood Structures

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 $\mathcal{N}_1$.

TS-JS: Neighborhood Structures

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)

Machine-Blocks

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:

- (i) The sequence contains at least two nodes.
- (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.

Machine-Blocks

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.)

Machine-Blocks

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 $G(s')$

$\mathcal{N}_2$

Consider (u, v) on critical path w.r.t. s.
Disadvantage of $\mathcal{N}_1$:

- 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 ($\mathcal{N}_2$).
Clearly, $\mathcal{N}_1 \subseteq \mathcal{N}_2 \Rightarrow \mathcal{N}_2$ is OPT-connected

$\mathcal{N}_3$

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?

$\mathcal{N}_4$

$\mathcal{N}_3$ can be extended to a neighborhood $\mathcal{N}_4$ wich 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 $f(s)$

Genetic Algorithm

Starting from an initial population, 'parent' solutions are selected and new 'child' solutions are created by genetic operators:

- *corssover*
  - mix subsequences of parent chromosomes
- *mutation*
  - pertubate a chromosome

Size of the population is controlled by fitness value.

Genetic Algorithm

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

Genetic Algorithm

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 $\longrightarrow$ 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.