Two Historical Local Search Methods

Leandro Montero

UCO Angers

2025-2026

based on lectures of Fabien Tricoire

Outline

Outline

Notation and Terminology

In the following we assume:

- That we are working on a minimisation problem,
- for which we already have a working construction heuristic,
- and good, appropriate neighbourhoods.

Notation and Terminology

In the following we assume:

- That we are working on a minimisation problem,
- for which we already have a working construction heuristic,
- and good, appropriate neighbourhoods.

Terminology

- We do not consider, unless specified, unfeasible solutions; a solution always satisfies all constraints in the problem.
- The incumbent solution is the current working solution.
- The best found solution is the best of all incumbent solutions since the beginning of the execution.
- A move allows, from a given solution, to obtain one of its neighbours

Notation and Terminology

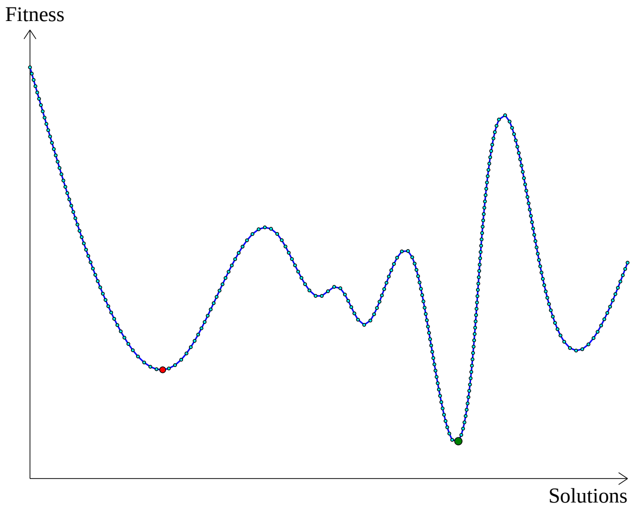
Notation

If a and b are 2 solutions to a given problem, then:

- $z(a)$ and $z(b)$ are their respective objective values (fitness)
- $\Delta_{a,b} = z(b) - z(a)$ is the difference of these two
- $Gap(a, b) = \frac{\Delta_{a,b}}{z(a)}$ is the percentage gap

Outline

Never forget this!



Stochastic Local Search: The main idea

Definitions

Descent: a local search performing only improvements.

Steepest descent: a local search always performing the best improvement available in the neighbourhood.

(This is only true for minimisation problems; for maximisation problems, we have ascent and steepest ascent. . .)

Stochastic Local Search: The main idea

Definitions

Descent: a local search performing only improvements.

Steepest descent: a local search always performing the best improvement available in the neighbourhood.

(This is only true for minimisation problems; for maximisation problems, we have ascent and steepest ascent. . .)

- These descent/ascent methods always lead very quickly to a local optimum
- Sometimes it might be useful to temporarily degrade the quality of the incumbent solution
- The main idea of these old-school stochastic local searches is to allow degradations under certain circumstances

Outline

Before we start

In this section, we will use the TSP as a toy problem

Quick reminder

- n cities (numbered 1 to n) plus a depot (0)
- Cost c_{ij} between cities i and j
- Objective: minimise total travelling cost required to start from the depot, visit all cities and go back to the depot

Before we start

In this section, we will use the TSP as a toy problem

Quick reminder

- n cities (numbered 1 to n) plus a depot (0)
- Cost c_{ij} between cities i and j
- Objective: minimise total travelling cost required to start from the depot, visit all cities and go back to the depot

With the help of this toy problem, we will:

- Learn about Simulated Annealing
- Develop a Simulated Annealing algorithm for the TSP
- Not go into every detail
- Experiment this algorithm on a few instances for quantitative analysis

Simulated Annealing (SA)

The annealing analogy

Annealing is a physical process consisting in melting a material, and then cooling it very slowly, in order to reach the most stable structure. If cooled too fast, it might reach an unwanted state. The aim is to reach a perfect state of minimum energy (ground state), thus obtaining specific physical properties.
(freely adapted from the book by Hoos and Stützle)

Simulated Annealing (SA)

The annealing analogy

Annealing is a physical process consisting in melting a material, and then cooling it very slowly, in order to reach the most stable structure. If cooled too fast, it might reach an unwanted state. The aim is to reach a perfect state of minimum energy (ground state), thus obtaining specific physical properties.
(freely adapted from the book by Hoos and Stützle)

Exercise: find an analogy with combinatorial optimisation

Simulated Annealing: key ideas

Analogy: temperature

- Physics: the higher the temperature, the easier it is to reach less stable states
- Optimisation: we want to reproduce this mechanism
  - Store the current "temperature" in a variable T
  - Reduce its value slowly through time
  - Use T to influence the probability to move to worse solutions

Simulated Annealing: key ideas

Analogy: temperature

- Physics: the higher the temperature, the easier it is to reach less stable states
- Optimisation: we want to reproduce this mechanism
  - Store the current "temperature" in a variable T
  - Reduce its value slowly through time
  - Use T to influence the probability to move to worse solutions

Temperature: the main ideas

- A high temperature allows more degradations
- Temperature should decrease along the whole solution process
  - Focus on diversification at the beginning
  - Focus on intensification towards the end
  - But still allow both mechanisms all along

Simulated Annealing: key ideas

Metaphor: moving to less stable states

- Higher temperatures allow more degradation, but this doesn't capture all the useful properties of annealing
- It should also be less common to perform big degradations than to perform small ones

Moving from solution x to solution x' should depend on both T and $\Delta_{x,x'}$

Simulated Annealing: the abstract algorithm

```
1: $T \leftarrow T_0$
2: $x \leftarrow buildSolution()$
3: $x^* \leftarrow x$
4: while stopping criterion not reached do
5:     $x' \leftarrow randomNeighbour(x)$
6:     $x \leftarrow accept(x, x', T)$
7:     $update(T)$
8:     if $z(x) < z(x^*)$ then
9:         $x^* \leftarrow x$
10:    end if
11: end while
```

Simulated Annealing: the abstract algorithm

```
1: $T \leftarrow T_0$
2: $x \leftarrow buildSolution()$
3: $x^* \leftarrow x$
4: **while** stopping criterion not reached **do**
5:     $x' \leftarrow randomNeighbour(x)$
6:     $x \leftarrow accept(x, x', T)$
7:     $update(T)$
8:     **if** $z(x) < z(x^*)$ **then**
9:         $x^* \leftarrow x$
10:    **end if**
11: **end while**
```

- Things we do not investigate today

Simulated Annealing: the abstract algorithm

```
1: $T \leftarrow T_0$
2: $x \leftarrow buildSolution()$
3: $x^* \leftarrow x$
4: while stopping criterion not reached do
5:     $x' \leftarrow randomNeighbour(x)$
6:     $x \leftarrow accept(x, x', T)$
7:     $update(T)$
8:     if $z(x) < z(x^*)$ then
9:         $x^* \leftarrow x$
10:    end if
11: end while
```

- Things we do not investigate today
- Things we need to specify

Simulated Annealing: the abstract algorithm

```
1: $T \leftarrow T_0$
2: $x \leftarrow buildSolution()$
3: $x^* \leftarrow x$
4: while stopping criterion not reached do
5:     $x' \leftarrow randomNeighbour(x)$
6:     $x \leftarrow accept(x, x', T)$
7:     $update(T)$
8:     if $z(x) < z(x^*)$ then
9:         $x^* \leftarrow x$
10:    end if
11: end while
```

- Things we do not investigate today
- Things we need to specify

Exercise: how is x^* called?

Acceptance decision

In practice

- The acceptance function $accept(x, x', T)$ is a probabilistic function, with parameters:
    - x: the current (or incumbent) solution
    - x': the new candidate solution
    - T: the temperature
- It returns the new incumbent solution, which is either x or x'
- It relies on the probability function $p(x, x', T)$

Acceptance decision

In practice

- The acceptance function $accept(x, x', T)$ is a probabilistic function, with parameters:
  - x: the current (or <u>incumbent</u>) solution
  - x': the new candidate solution
  - T: the temperature
- It returns the new incumbent solution, which is either x or x'
- It relies on the probability function $p(x, x', T)$

Acceptance probability

Let $p(x, x', T)$ be the probability to choose x' as the new incumbent in *accept*:

$$p(x, x', T) = \begin{cases} 1 & \text{if } z(x') \leqslant z(x) \\ f(\Delta_{x,x'}, T) & \text{otherwise } (f \text{ to be defined}) \end{cases}$$

Probabilistic acceptance: abstract algorithm (1)

Functions $z(x)$ and $f(\Delta_{x,x'}, T)$ are given...

Exercise: write the abstract algorithm for $p(x, x', T)$

Probabilistic acceptance: abstract algorithm (2)

Assume:

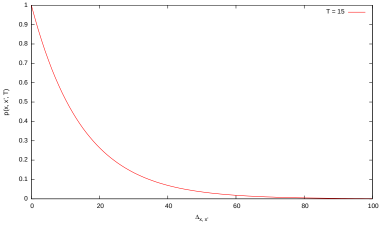
- Function *random*() returns a random number between 0 and 1
- Function $p(x, x', T)$ is given

Exercise: write the abstract algorithm for $accept(x, x', T)$

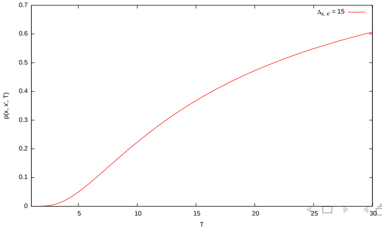
Finding a good acceptance function

Exercise: define $f(\Delta_{x,x'}, T)$, knowing that:

- For a fixed T, $f(\Delta_{x,x'}, T)$ should behave like this:



- For fixed x and x', $f(\Delta_{x,x'}, T)$ should behave like this:

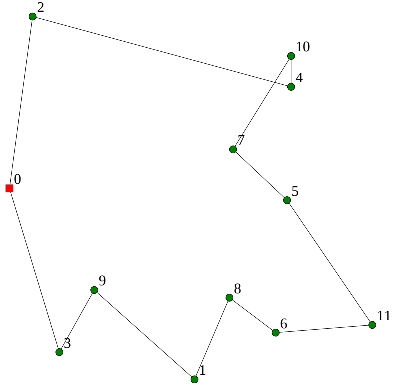


The first Simulated Annealing

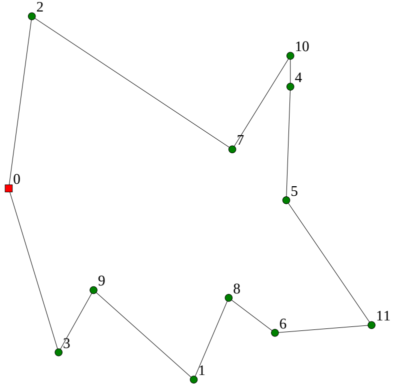
- Kirkpatrick et al., 1983: "Optimization by Simulated Annealing", Science vol. 220, pp 671-680
- Introduced the metaphor
- Applied SA to a variety of problems...
- ...including the TSP (with N cities, N up to 6000)
- Neighbourhood: reverse a sequence of cities

Acceptance: an example

Exercise: what is the probability to move from x to x'?



$z(x) = 367.99$



$z(x') = 370.65$

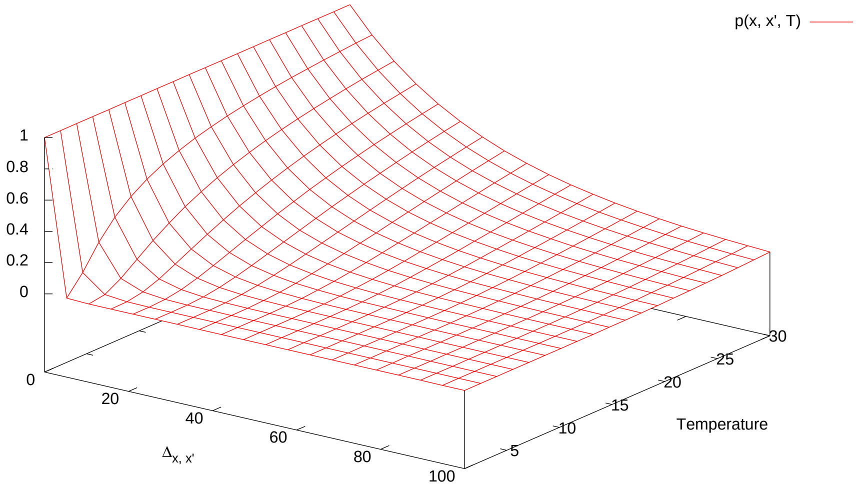
The first Simulated Annealing

900 cities: degradation acceptance probability in 2 dimensions

Exercise: draw the curve of $f(\Delta_{x,x'}, T)$ for T taking values 1, 5, 10, 20, 30

The first Simulated Annealing

900 cities: degradation acceptance probability in 3 dimensions



Simulated Annealing: the abstract algorithm

```
1: $T \leftarrow T_0$
2: $x \leftarrow buildSolution()$
3: $x^* \leftarrow x$
4: while stopping criterion not reached do
5:     $x' \leftarrow randomNeighbour(x)$
6:     $x \leftarrow accept(x, x', T)$
7:     $update(T)$
8:     if $z(x) < z(x^*)$ then
9:         $x^* \leftarrow x$
10:    end if
11: end while
```

- Things we do not investigate today
- Things we already specified
- Things we still need to specify

Stopping criteria

Possible stopping criteria

- Number of iterations
- Final temperature reached
- Number of iterations without improvement
- Too low acceptance ratio

Temperature updating

Cooling schedule

The cooling schedule, or annealing schedule, determines how the temperature is adjusted through the whole search process. The most common case is to decrease it gradually:

- Every k steps: $T \leftarrow \alpha \cdot T$

Temperature updating

Cooling schedule

The cooling schedule, or annealing schedule, determines how the temperature is adjusted through the whole search process. The most common case is to decrease it gradually:

- Every k steps: $T \leftarrow \alpha \cdot T$
- Constant α is to be determined (usually close to 0.99)
- Starting temperature T_0 is to be determined (problem-dependent)

Why is T_0 problem-dependent? How to improve this?

Simulated Annealing: the abstract algorithm

```
1: $T \leftarrow T_0$
2: $x \leftarrow buildSolution()$
3: $x^* \leftarrow x$
4: while stopping criterion not reached do
5:    $x' \leftarrow randomNeighbour(x)$
6:    $x \leftarrow accept(x, x', T)$
7:    $update(T)$
8:    if $z(x) < z(x^*)$ then
9:       $x^* \leftarrow x$
10:   end if
11: end while
```

- Things we do not investigate today
- Things we already specified

Experimental results

Simulated Annealing Vs Nearest neighbour

Instance Size	Construction	SA	Gap	CPU (s)
40	584.69	488.36	19.73%	3.67
50	641.47	566.11	13.31%	4.45
60	762.95	656.91	16.14%	5.30
70	833.43	689.32	20.91%	6.24
80	995.04	802.56	23.98%	7.02
90	959.64	851.84	12.65%	7.90
100	964.81	821.15	17.49%	8.75

After the first Simulated Annealing

Things to know

- Starting with a relatively low temperature can help (!)
- Optimality can be proved under certain conditions
- These conditions cannot be met in practice
  $\rightarrow$ Infinite computation time...
- Generally, “pure” Simulated Annealing algorithms performs rather poorly,
- But they can be improved in various ways.
- Tuning T_0 and the cooling schedule is of crucial importance.

Simulated Annealing: Discussion

Qualitative analysis check list

- Clarity, Modularity, Simplicity
- Intensification
- Diversification

Simulated Annealing: Discussion

Qualitative analysis check list

- Clarity, Modularity, Simplicity
- Intensification
- Diversification

Your opinion

- Questions
- Suggestions

Outline

Before we start

In this section, we will use $P||C_{max}$ examples.

Quick reminder

- m parallel machines (numbered 1 to m)
- n jobs (numbered 0 to $n-1$), with different durations
- Objective: minimise total completion time (a.k.a. makespan)

Before we start

In this section, we will use $P||C_{max}$ examples.

Quick reminder

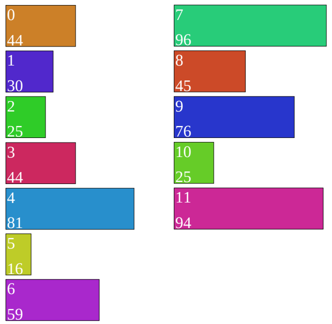
- m parallel machines (numbered 1 to m)
- n jobs (numbered 0 to $n-1$), with different durations
- Objective: minimise total completion time (a.k.a. makespan)

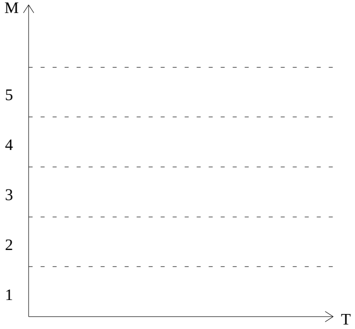
With these examples, we will:

- Construct (not always good) solutions
- Design an appropriate neighbourhood for improvement heuristics
- Emphasise problems related to local optima
- Address them with Tabu Search
- Verify that Tabu Search is useful and efficient

Paralle Machines, revisited

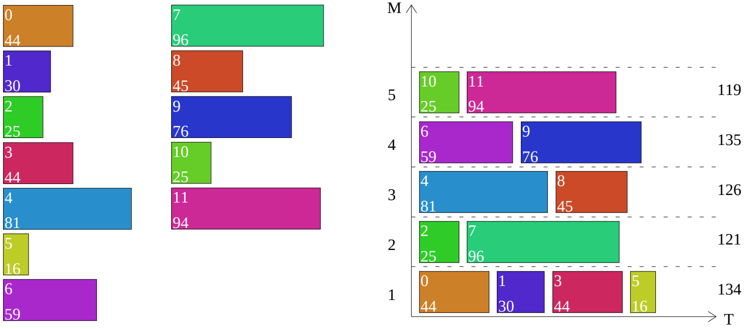
A relatively small example (5 machines, 12 jobs):





Paralle Machines, revisited

A relatively small example (5 machines, 12 jobs):



Enumeration is still possible! It took 19.5 minutes on my laptop.

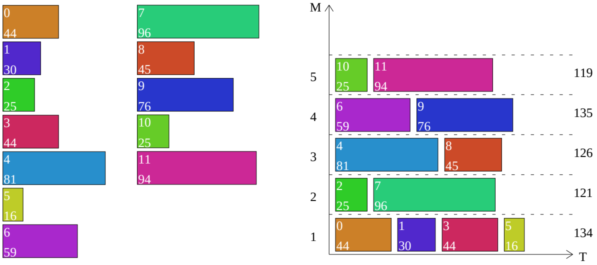
But we would like to use heuristics. . .

Solutions representation

- Suppose that *machineForJob* is an array representing a solution
- *machineForJob*[*i*] contains the index of the machine to which job *i* is assigned in this solution
- Please note that unlike lists, arrays allow constant-time access, i.e. accessing any element has complexity $O(1)$

Solutions representation

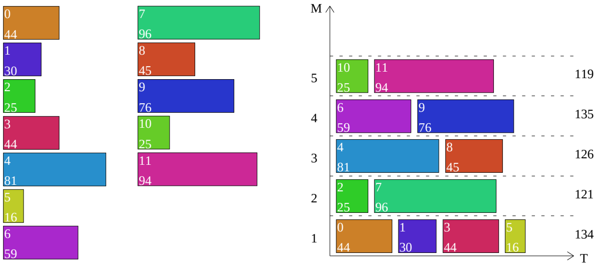
- Suppose that *machineForJob* is an array representing a solution
- *machineForJob*$[i]$ contains the index of the machine to which job i is assigned in this solution
- Please note that unlike lists, arrays allow constant-time access, i.e. accessing any element has complexity $O(1)$



Exercise: write the content of *machineForJob* for this solution

Solutions representation

- Suppose that *machineForJob* is an array representing a solution
- *machineForJob*$[i]$ contains the index of the machine to which job i is assigned in this solution
- Please note that unlike lists, arrays allow constant-time access, i.e. accessing any element has complexity $O(1)$



Exercise: write the content of *machineForJob* for this solution

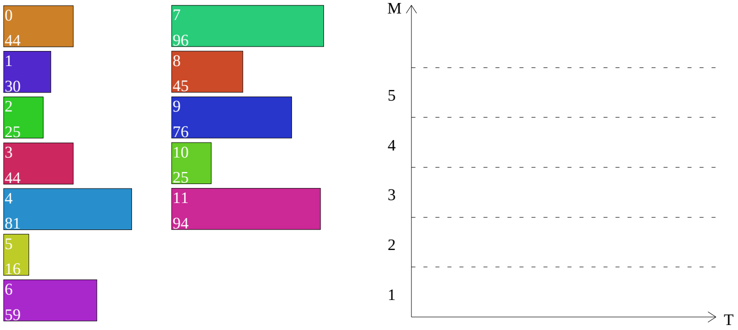
Job	0	1	2	3	4	5	6	7	8	9	10	11
Machine												

Construction heuristics for the $P||C_{max}$

Exercise: design a construction heuristic for the $P||C_{max}$
(it doesn't have to be good)

Construction heuristics for the $P||C_{max}$

Exercise: apply this construction heuristic to this $P||C_{max}$ instance



Job	0	1	2	3	4	5	6	7	8	9	10	11
Machine												

Total duration =

Step 2: Choose a neighbourhood

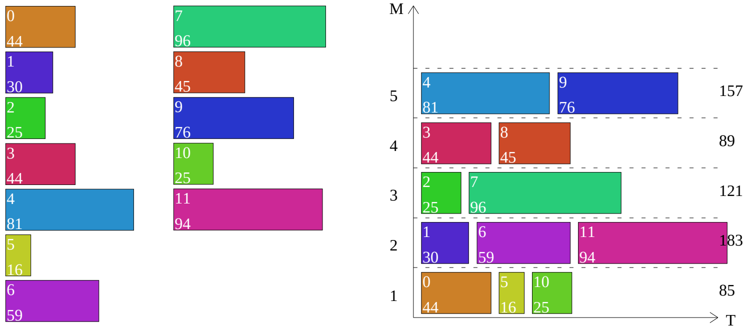
Now we want to improve this solution
$\rightarrow$ we need a good neighbourhood

<u>Exercise: design a neighbourhood for the $P||C_{max}$</u>

Neighbourhood for the $P||C_{max}$

For all the Tabu Search section, we use the neighbourhood consisting in moving a job from one machine to another.

Exercise: perform a steepest descent on our bad starting solution



Job	0	1	2	3	4	5	6	7	8	9	10	11
Machine	1	2	3	4	5	1	2	3	4	5	1	2

Duration = 183

Calculating the objective value of a solution

Assume the duration of job i is given by $jobDuration[i]$...

Exercise: write the abstract algorithm for function $getDuration(x)$

Introducing Tabu Search

Tabu Search aims, among other, at escaping local optima!

Introducing Tabu Search

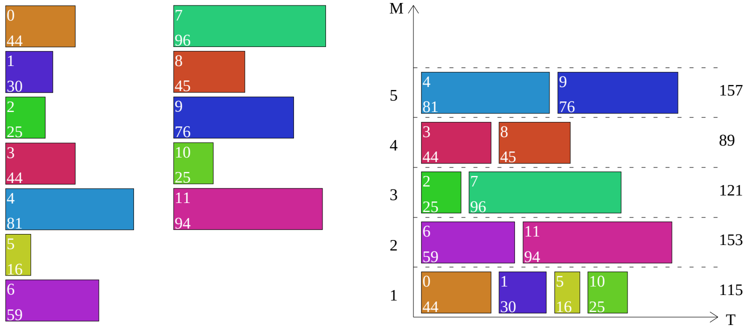
Tabu Search aims, among other, at escaping local optima!

Tabu Search idea #1: Move to the best neighbour

In Tabu Search, the next move should always be to the best neighbour.
This includes cases in which the best neighbour is less good than the incumbent.

In practice

Exercise: apply this first idea to the previous local optimum



Job	0	1	2	3	4	5	6	7	8	9	10	11
Machine	1	1	3	4	5	1	2	3	4	5	1	2

Duration = 157

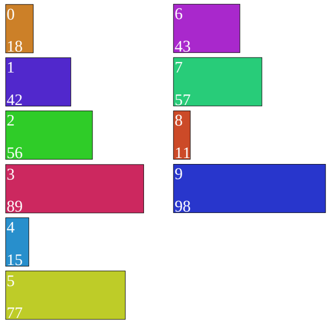
The idea is nice but...

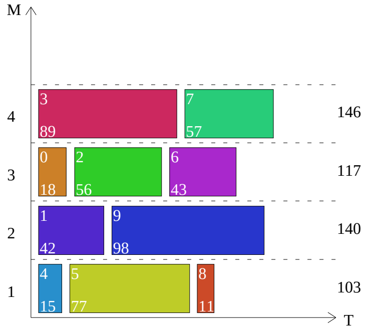
Exercise: this idea #1 can lead to a problem; what is it?

Cycles: an example

Looking for the best neighbour

Exercise: find the best neighbour a few times





Avoiding cycles

Tabu Search is called Tabu for a reason!

Tabu Search idea #2: Short-term memory

- Something needs to be done in order to avoid cycling
- Tabu Search remembers parts of the recent search history, in a Tabu List (TL)
- These parts are called Tabus, and are disregarded when exploring the neighbourhood
- This way cycling moves can be detected and avoided beforehand

Avoiding cycles

Tabu Search is called Tabu for a reason!

Tabu Search idea #2: Short-term memory

- Something needs to be done in order to avoid cycling
- Tabu Search remembers parts of the recent search history, in a Tabu List (TL)
- These parts are called Tabus, and are disregarded when exploring the neighbourhood
- This way cycling moves can be detected and avoided beforehand

Two crucial questions:

1. What is a tabu? (i.e. what kind of information do we store)
2. How long should a tabu remain active?

Defining what is a tabu

(At least) two design possibilities:

1. Tabus are whole solutions
2. Tabus are moves

Defining what is a tabu

(At least) two design possibilities:

1. Tabus are whole solutions
2. Tabus are moves

Then given a move we want to perform, we should

- If tabus are moves: check if this move is in the tabu list
- If tabus are solutions: check if the neighbour that this move will produce is in the tabu list

Lookup is made through successive comparisons with elements of the list.

Defining what is a tabu

Exercise: one of these two possibilities is a bad idea; which one, and why?

Tabu moves: possibilities

The principle

- Suppose we just moved job i from machine p to q, and we want to avoid cycling
- Then we must forbid the reversal, i.e. moving i from q to p

Tabu moves: possibilities

The principle

- Suppose we just moved job i from machine p to q, and we want to avoid cycling
- Then we must forbid the reversal, i.e. moving i from q to p

Again, there are several ways to design what is a tabu:

- A single move: (i, q, p) meaning "move job i from machine q to machine p"
- A set of moves: (i, p) meaning "move job i to machine p", regardless of the current machine for job i
- A larger set of moves: (i) meaning "move job i"

In the following, we use the latter (quite restrictive)

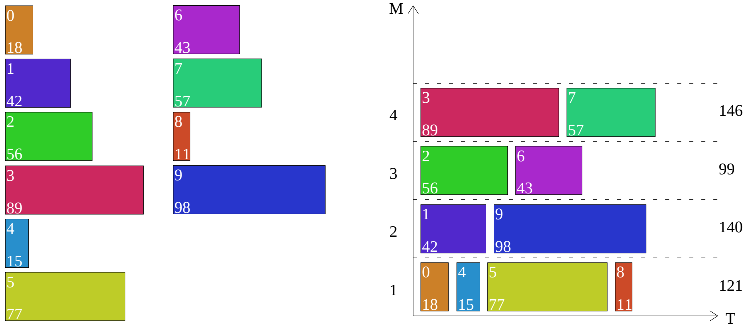
Tabu tenure

Definition

Tenure: the number of iterations during which a given tabu will remain in the list.

Exercise: if the tenure is p, what is the size of the tabu list?

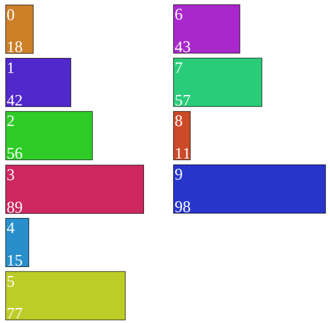
Full Tabu Search: an example (TLS = 2)

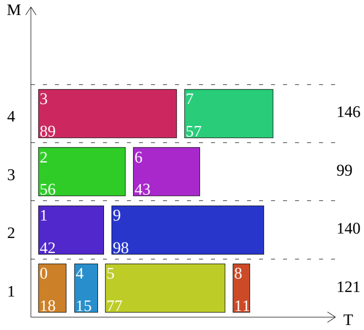


Tabu List =()

Job	0	1	2	3	4	5	6	7	8	9
Machine	1	2	3	4	1	1	3	4	1	2

Full Tabu Search: an example (TLS = 2)

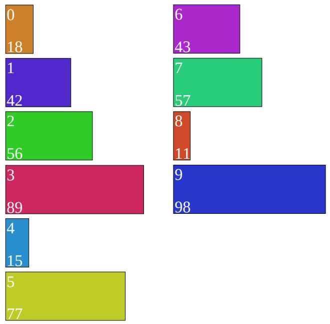


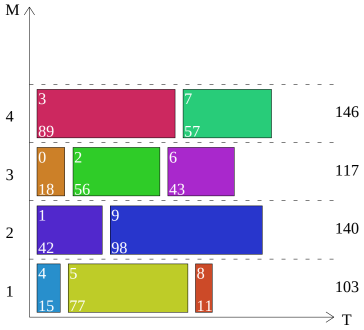


Tabu List =()

Job	0	1	2	3	4	5	6	7	8	9
Machine	1	2	3	4	1	1	3	4	1	2

Full Tabu Search: an example (TLS = 2)

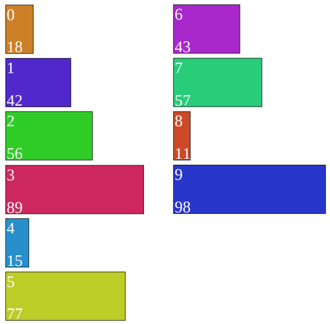


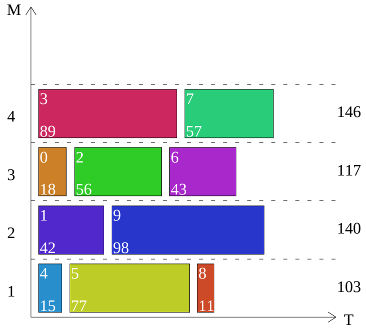


Tabu List =()

Job	0	1	2	3	4	5	6	7	8	9
Machine	3	2	3	4	1	1	3	4	1	2

Full Tabu Search: an example (TLS = 2)

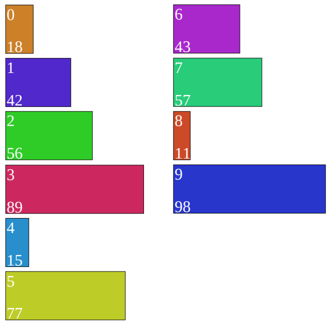


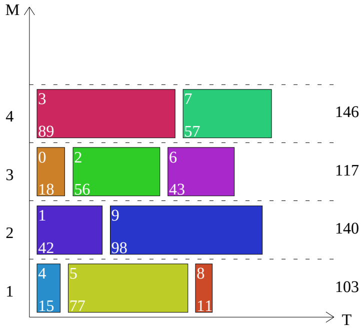


Tabu List =(0)

Job	0	1	2	3	4	5	6	7	8	9
Machine	3	2	3	4	1	1	3	4	1	2

Full Tabu Search: an example (TLS = 2)

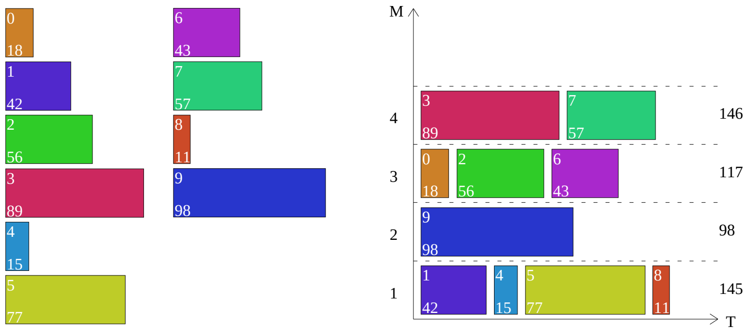




Tabu List =(0)

Job	0	1	2	3	4	5	6	7	8	9
Machine	3	2	3	4	1	1	3	4	1	2

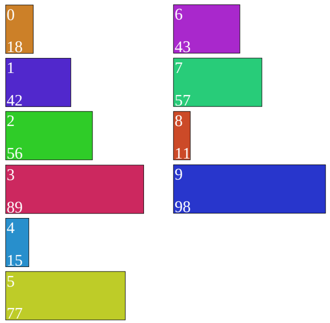
Full Tabu Search: an example (TLS = 2)

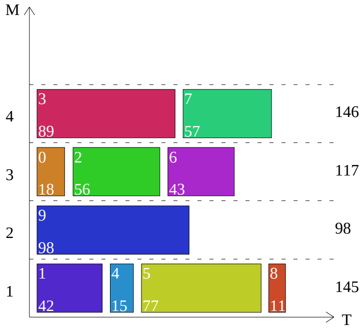


Tabu List =(0)

Job	0	1	2	3	4	5	6	7	8	9
Machine	3	1	3	4	1	1	3	4	1	2

Full Tabu Search: an example (TLS = 2)

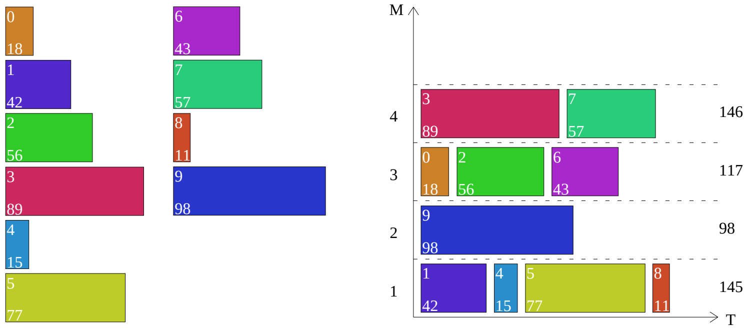




Tabu List $=(0, 1)$

Job	0	1	2	3	4	5	6	7	8	9
Machine	3	1	3	4	1	1	3	4	1	2

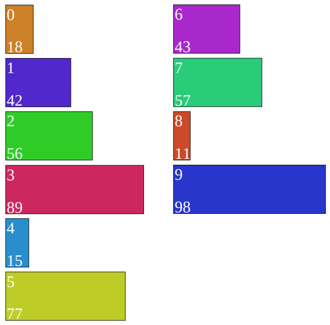
Full Tabu Search: an example (TLS = 2)

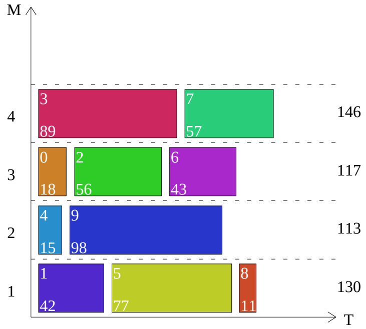


Tabu List =(0, 1)

Job	0	1	2	3	4	5	6	7	8	9
Machine	3	1	3	4	1	1	3	4	1	2

Full Tabu Search: an example (TLS = 2)

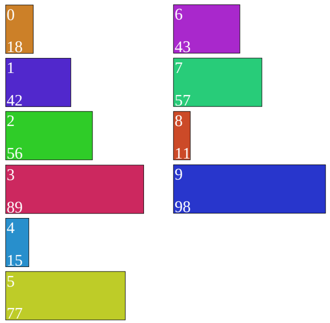


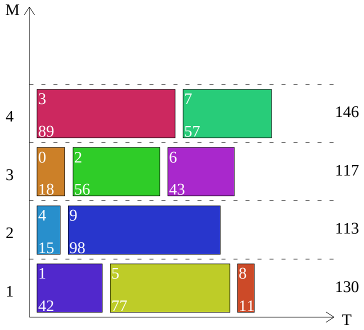


Tabu List $=(0,1)$

Job	0	1	2	3	4	5	6	7	8	9
Machine	3	1	3	4	2	1	3	4	1	2

Full Tabu Search: an example (TLS = 2)

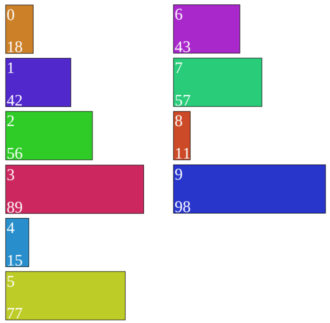


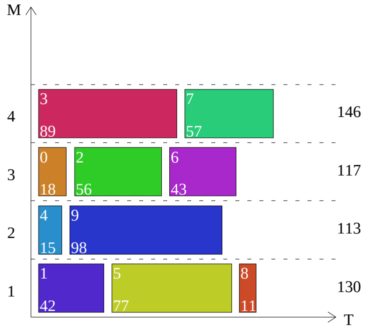


Tabu List $=(1, 4)$

Job	0	1	2	3	4	5	6	7	8	9
Machine	3	1	3	4	2	1	3	4	1	2

Full Tabu Search: an example (TLS = 2)

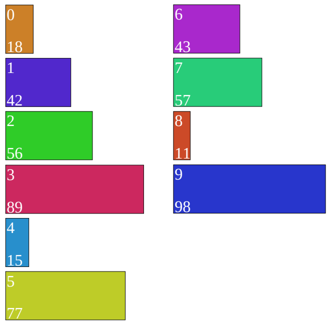


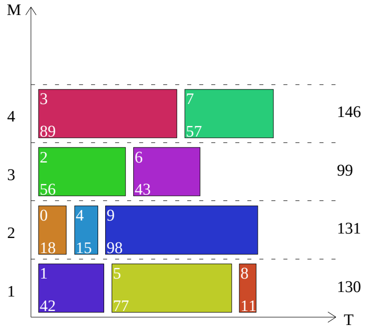


Tabu List =(1, 4)

Job	0	1	2	3	4	5	6	7	8	9
Machine	3	1	3	4	2	1	3	4	1	2

Full Tabu Search: an example (TLS = 2)

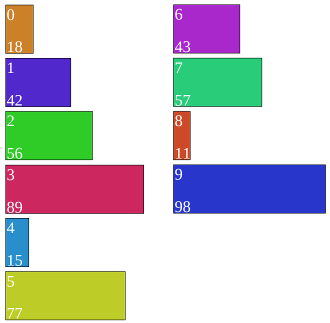


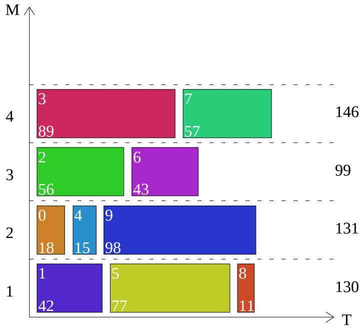


Tabu List =$(1, 4)$

Job	0	1	2	3	4	5	6	7	8	9
Machine	2	1	3	4	2	1	3	4	1	2

Full Tabu Search: an example (TLS $= 2$)

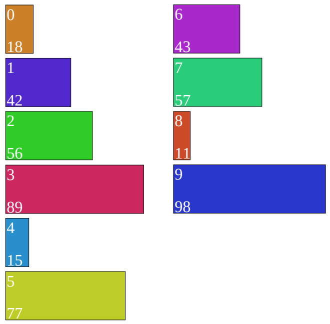


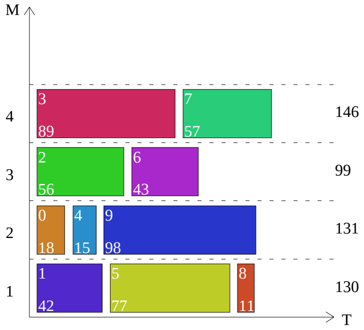


Tabu List $=(4,0)$

Job	0	1	2	3	4	5	6	7	8	9
Machine	2	1	3	4	2	1	3	4	1	2

Full Tabu Search: an example (TLS = 2)

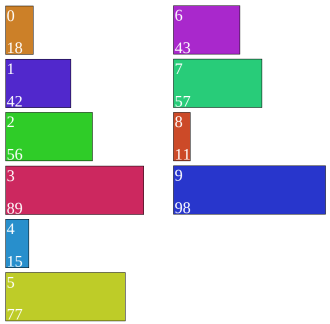


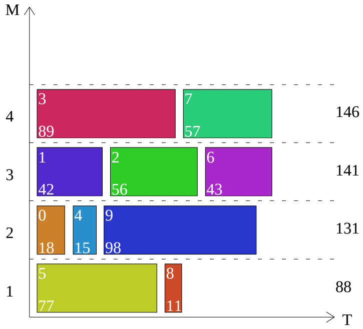


Tabu List =$(4, 0)$

Job	0	1	2	3	4	5	6	7	8	9
Machine	2	1	3	4	2	1	3	4	1	2

Full Tabu Search: an example (TLS = 2)

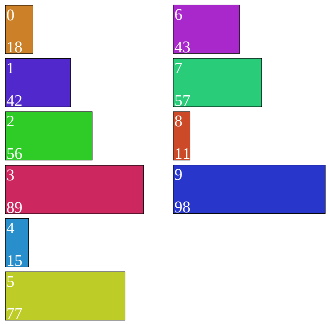


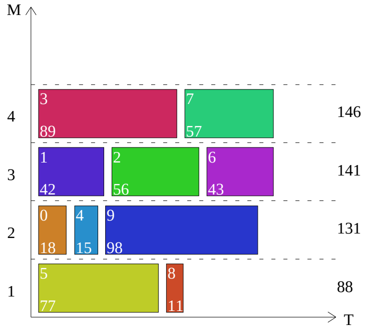


Tabu List =(4, 0)

Job	0	1	2	3	4	5	6	7	8	9
Machine	2	3	3	4	2	1	3	4	1	2

Full Tabu Search: an example (TLS = 2)

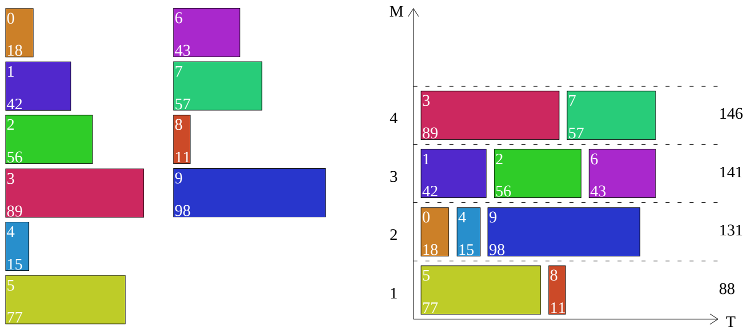




Tabu List $=(0,1)$

Job	0	1	2	3	4	5	6	7	8	9
Machine	2	3	3	4	2	1	3	4	1	2

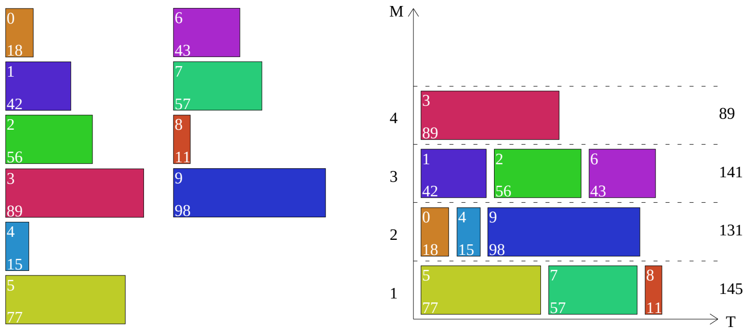
Full Tabu Search: an example (TLS = 2)



Tabu List =$(0, 1)$

Job	0	1	2	3	4	5	6	7	8	9
Machine	2	3	3	4	2	1	3	4	1	2

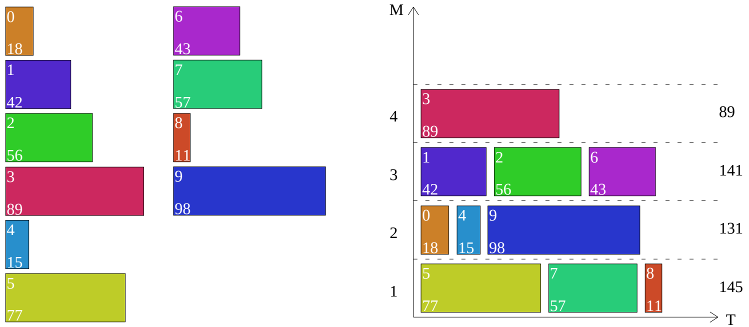
Full Tabu Search: an example (TLS = 2)



Tabu List =(0, 1)

Job	0	1	2	3	4	5	6	7	8	9
Machine	2	3	3	4	2	1	3	1	1	2

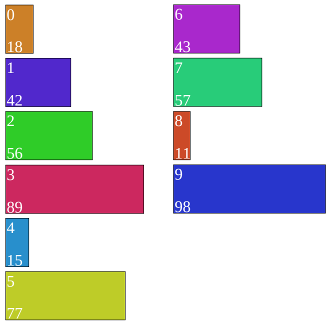
Full Tabu Search: an example (TLS = 2)

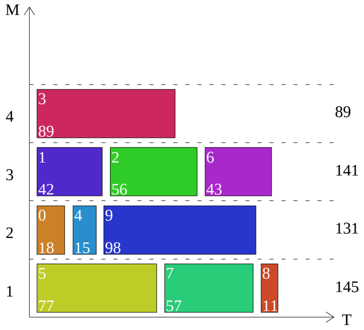


Tabu List $=(1, 7)$

Job	0	1	2	3	4	5	6	7	8	9
Machine	2	3	3	4	2	1	3	1	1	2

Full Tabu Search: an example (TLS = 2)

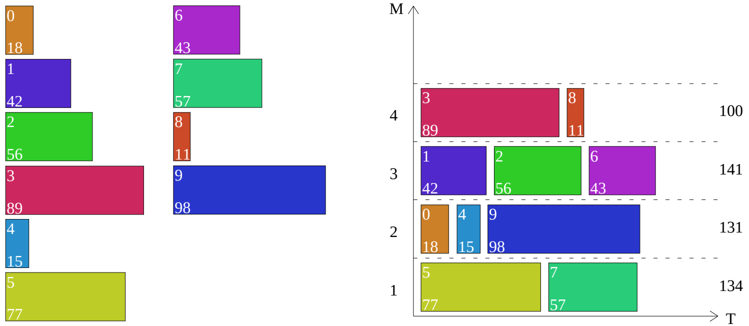




Tabu List $=(1,7)$

Job	0	1	2	3	4	5	6	7	8	9
Machine	2	3	3	4	2	1	3	1	1	2

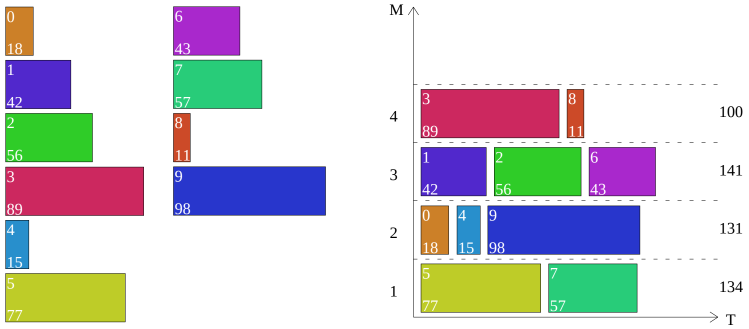
Full Tabu Search: an example (TLS = 2)



Tabu List =(1, 7)

Job	0	1	2	3	4	5	6	7	8	9
Machine	2	3	3	4	2	1	3	1	4	2

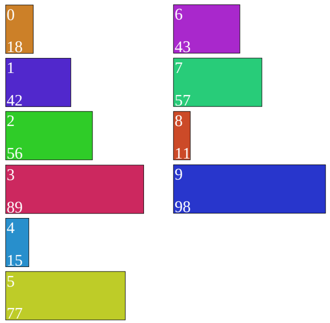
Full Tabu Search: an example (TLS = 2)

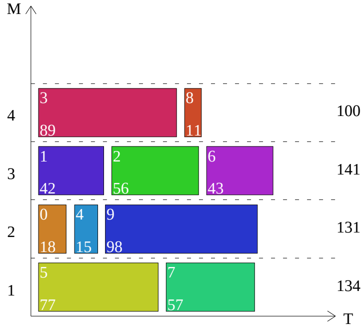


Tabu List =(7, 8)

Job	0	1	2	3	4	5	6	7	8	9
Machine	2	3	3	4	2	1	3	1	4	2

Full Tabu Search: an example (TLS = 2)





Tabu List $=(7,8)$

That's already better!

Job	0	1	2	3	4	5	6	7	8	9
Machine	2	3	3	4	2	1	3	1	4	2

Tabu List handling

For each job, we need to store:

- If it is tabu
- If so, how long it should remain tabu

Exercise: suggest a data structure to store the Tabu List

Tabu List Handling

When a move has been performed, we want to add a new tabu.
Suppose the tenure is τ, then we can write the abstract algorithm:

addTabu(TL, i)

```
1: $TL[i] \leftarrow \tau$
```

Tabu List Handling

When a move has been performed, we want to add a new tabu.
Suppose the tenure is τ, then we can write the abstract algorithm:

addTabu(*TL*, *i*)

```
1: $TL[i] \leftarrow \tau$
```

But tabus should not remain forever!
After each iteration, we need to update the list, to decrease the number of iterations after which the tabu will be outdated.

Tabu List Handling

Exercise: write the abstract algorithm for the TL update

Finding the best non-tabu move: abstract algorithm

bestNonTabuMove(x, TL)

```
1: $bestDuration \leftarrow \infty$
2: for $i \leftarrow 0$ to $n-1$ do
3:     if $TL[i] = 0$ then
4:         for $j \leftarrow 1$ to $m$ do
5:             $x' \leftarrow x$
6:             $x'[i] \leftarrow j$
7:             if $getDuration(x') < bestDuration$ then
8:                 $bestDuration \leftarrow getDuration(x')$
9:                 $bestJob \leftarrow i$
10:                $bestMachine \leftarrow j$
11:            end if
12:        end for
13:    end if
14: end for
15: Return $bestJob, bestMachine$
```

Applying a move to a solution: abstract algorithm

This algorithm returns the neighbour of x obtained by moving job i to machine p:

moveJob(x, i, p)

```
1: $x' \leftarrow x$
2: $x'[i] \leftarrow p$
3: Return $x'$
```

Tabu Search: abstract algorithm

TabuSearch()

```
1: $x \leftarrow buildSolution()$
2: $TL \leftarrow \emptyset$
3: $x^* \leftarrow x$
4: while stopping criterion not reached do
5:     $i, p \leftarrow bestNonTabuMove(x, TL)$
6:     $update(TL)$
7:     $addTabu(TL, i)$
8:     $x \leftarrow moveJob(x, i, p)$
9:     if $z(x) < z(x^*)$ then
10:        $x^* \leftarrow x$
11:    end if
12: end while
13: Return $x^*$
```

Tabu Search: abstract algorithm

TabuSearch()

```
1: $x \leftarrow buildSolution()$
2: $TL \leftarrow \emptyset$
3: $x^* \leftarrow x$
4: while stopping criterion not reached do
5:     $i, p \leftarrow bestNonTabuMove(x, TL)$
6:     $update(TL)$
7:     $addTabu(TL, i)$
8:     $x \leftarrow moveJob(x, i, p)$
9:     if $z(x) < z(x^*)$ then
10:        $x^* \leftarrow x$
11:    end if
12: end while
13: Return $x^*$
```

- Things we already defined

Tabu Search: abstract algorithm

TabuSearch()

```
1: $x \leftarrow buildSolution()$
2: $TL \leftarrow \emptyset$
3: $x^* \leftarrow x$
4: while stopping criterion not reached do
5:     $i, p \leftarrow bestNonTabuMove(x, TL)$
6:     $update(TL)$
7:     $addTabu(TL, i)$
8:     $x \leftarrow moveJob(x, i, p)$
9:     if $z(x) < z(x^*)$ then
10:        $x^* \leftarrow x$
11:    end if
12: end while
13: Return $x^*$
```

- Things we already defined
- Things we still need to specify

Tabu Search stopping criteria

Once again, several possibilities:

- Number of iterations
- Number of iterations without improvement
- Anything you can imagine

Tabu Search stopping criteria

Once again, several possibilities:

- Number of iterations
- Number of iterations without improvement
- Anything you can imagine

In the following, we use the total number of iterations.

- I implemented these abstract algorithms in a very perfectible way
- Computation never took more than a few seconds

Quantitative analysis and parameter tuning

10000 iterations

On small instances:

- We can compare our results with the optimum

Instance (m, n)	Opt	TLS $= 3$	TLS $= 6$	TLS $= 9$
4, 8	109	110	109	111
4, 10	131	146	131	139
4, 12	217	221	219	217
5, 8	74	74	74	74
5, 10	117	117	117	117
5, 12	135	135	135	135

Quantitative analysis and parameter tuning

10000 iterations

On small instances:

- We can compare our results with the optimum

Instance (m, n)	Opt	TLS $= 3$	TLS $= 6$	TLS $= 9$
4, 8	109	110	109	111
4, 10	131	146	131	139
4, 12	217	221	219	217
5, 8	74	74	74	74
5, 10	117	117	117	117
5, 12	135	135	135	135

- Our Tabu Search can find the optimum on all instances
- But not always with the same parameter setting

Quantitative analysis and parameter tuning

10000 iterations

On bigger instances:

- We can only compare our results between themselves
- Some parameter settings are better than others

Instance (m, n)	TLS $= 3$	TLS $= 6$	TLS $= 9$	TLS $= n/3$
10, 20	133	125	125	125
10, 30	193	188	185	187
10, 40	217	212	212	214
10, 50	295	282	282	283
10, 60	366	361	362	362

Quantitative analysis and parameter tuning

10000 iterations

On bigger instances:

- We can only compare our results between themselves
- Some parameter settings are better than others

Instance (m, n)	TLS $= 3$	TLS $= 6$	TLS $= 9$	TLS $= n/3$
10, 20	133	125	125	125
10, 30	193	188	185	187
10, 40	217	212	212	214
10, 50	295	282	282	283
10, 60	366	361	362	362

- Higher problem instances seem to require a higher TLS
- But too high values don't give better results
- Our algorithm looks relatively stable despite variations of the TLS

Quantitative analysis: comparison with steepest descent

10000 iterations

Instance (m, n)	TLS $= 6$	Steepest descent	Gap
4, 8	109	114	4.6%
4, 10	131	146	11.5%
4, 12	219	253	15.5%
5, 8	74	74	0%
5, 10	117	122	4.3%
5, 12	135	157	16.3%
10, 20	125	135	8.0%
10, 30	188	203	7.98%
10, 40	212	268	26.4%
10, 50	282	295	4.6%
10, 60	361	393	8.9%

- Except for one instance, important improvements

Quantitative analysis: preliminary conclusions

Our experiments are too few to conclude anything serious, see them as a stub.

Quantitative analysis: preliminary conclusions

Our experiments are too few to conclude anything serious, see them as a stub.

However, we showed that on our very restricted set of instances:

- Our TS finds near-optimal solutions
- Parameter setting has an impact, and we found a tenure setting working better than the others
- However, variations are not dramatic as long as the tenure stays within a certain range
- Our TS works much better than a steepest descent

Tabu Search: Discussion

Qualitative analysis check list

- Clarity, Modularity, Simplicity
- Intensification
- Diversification

Your opinion

- Questions
- Suggestions

Things to know about Tabu Search

It is still not perfect!

Problem # 1

It can happen that a move is tabu when it would in fact lead to a useful solution (maybe better than the best found so far).

Exercise: propose solutions to address this problem

Things to know about Tabu Search

Problem # 2

The TL is a short-term memory, and might provide insufficient diversification.

Exercise: propose solutions to address this problem

Resources

- Fred Glover & Manuel Laguna, "Tabu Search", Kluwer Academic Publishers, 1997.
- Click here for a good tutorial by Michel Gendreau, made for students.