合肥学院
HEFEI UNIVERSITY

Metaheuristic Optimization

8. Tabu Search

Thomas Weise · 汤卫思

tweise@hfuu.edu.cn · http://iao.hfuu.edu.cn

Hefei University, South Campus 2	合肥学院 南艳湖校区/南2区
Faculty of Computer Science and Technology	计算机科学与技术系
Institute of Applied Optimization	应用优化研究所
230601 Shushan District, Hefei, Anhui, China	中国 安徽省 合肥市 蜀山区 230601
Econ. & Tech. Devel. Zone, Jinxiu Dadao 99	经济技术开发区 锦绣大道99号

Outline

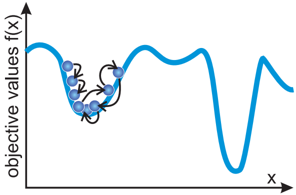
1. Introduction
2. Tabu Search
3. Example 1: MAX-SAT
4. Example 2: Traveling Salesman Problem
5. Iterated Local Search
6. Summary

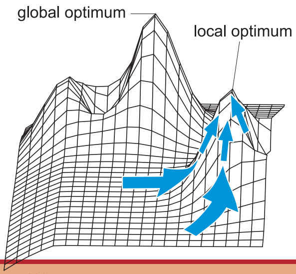
website

Section Outline

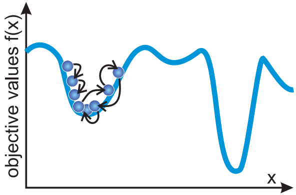
1. Introduction
2. Tabu Search
3. Example 1: MAX-SAT
4. Example 2: Traveling Salesman Problem
5. Iterated Local Search
6. Summary

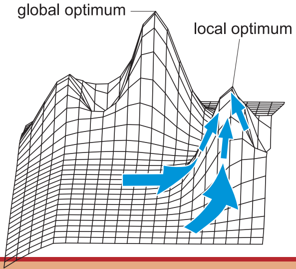
- An optimum is a solution which is better than all of its neighboring solutions.



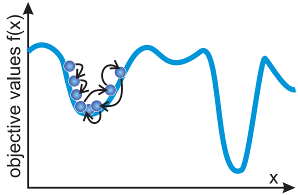


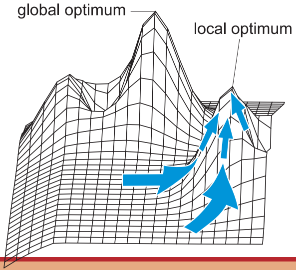
- An optimum is a solution which is better than all of its neighboring solutions.
- A “neighboring solution” is a solution which can be reached with a single application of a unary search operation.



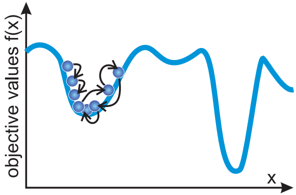


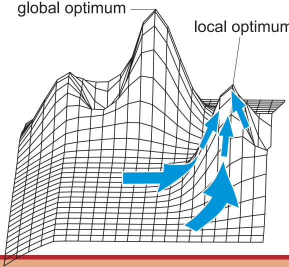
- An optimum is a solution which is better than all of its neighboring solutions.
- A “neighboring solution” is a solution which can be reached with a single application of a unary search operation.
- A *local* optimum $x^{\star}$ is an optimum which is worse than the global optimum $\overset{\star}{x}^{\star}$, i.e., $f(x^{\star}) > f(\overset{\star}{x}^{\star})$ on minimimization problems.





- An optimum is a solution which is better than all of its neighboring solutions.
- A "neighboring solution" is a solution which can be reached with a single application of a unary search operation.
- A *local* optimum $x^{\star}$ is an optimum which is worse than the global optimum $x^{\star\!\!\star}$, i.e., $f(x^{\star}) > f(x^{\star\!\!\star})$ on minimimization problems.
- Hill climbers will get stuck at *any* optimum, because they will only move from one solution to a better solution.





- An optimum is a solution which is better than all of its neighboring solutions.
- A "neighboring solution" is a solution which can be reached with a single application of a unary search operation.
- A *local* optimum $x^{\star}$ is an optimum which is worse than the global optimum $x^{\star\star}$, i.e., $f(x^{\star}) > f(x^{\star\star})$ on minimimization problems.
- Hill climbers will get stuck at *any* optimum, because they will only move from one solution to a better solution.
- They are likely to converge to local optimum, i.e., may not give us the globally optimal solution.

- An optimum is a solution which is better than all of its neighboring solutions.
- A “neighboring solution” is a solution which can be reached with a single application of a unary search operation.
- A *local* optimum $x^{\star}$ is an optimum which is worse than the global optimum $x^{\star\star}$, i.e., $f(x^{\star}) > f(x^{\star\star})$ on minimimization problems.
- Hill climbers will get stuck at *any* optimum, because they will only move from one solution to a better solution.
- They are likely to converge to local optimum, i.e., may not give us the globally optimal solution.
- Simulated Annealing can avoid this, because it sometimes (proabilistically) also accepts worse candidate solutions.

Introduction

- An optimum is a solution which is better than all of its neighboring solutions.
- A "neighboring solution" is a solution which can be reached with a single application of a unary search operation.
- A *local* optimum $x^{\star}$ is an optimum which is worse than the global optimum $x^{\star\!\!\star}$, i.e., $f(x^{\star}) > f(x^{\star\!\!\star})$ on minimimization problems.
- Hill climbers will get stuck at *any* optimum, because they will only move from one solution to a better solution.
- They are likely to converge to local optimum, i.e., may not give us the globally optimal solution.
- Simulated Annealing can avoid this, because it sometimes (proabilistically) also accepts worse candidate solutions.
- Tabu Search, introduced by Glover, Glover [1, 2], is another local search which introduces another, similar approach.

Section Outline

1. Introduction
2. Tabu Search
3. Example 1: MAX-SAT
4. Example 2: Traveling Salesman Problem
5. Iterated Local Search
6. Summary

Move to Best Solution

- Simulated Annealing, in each step, applies the unary search operation to create a (often randomly modified) copy of the current solution.

Move to Best Solution

- Simulated Annealing, in each step, applies the unary search operation to create a (often randomly modified) copy of the current solution.
- Tabu Search scans the *whole neighborhood* of the current solution and picks the best neighboring solution as next solution.

Move to Best Solution

- Simulated Annealing, in each step, applies the unary search operation to create a (often randomly modified) copy of the current solution.
- Tabu Search scans the *whole neighborhood* of the current solution and picks the best neighboring solution as next solution.
- It will pick this solution even if it is worse than the current solution.

Move to Best Solution

- Simulated Annealing, in each step, applies the unary search operation to create a (often randomly modified) copy of the current solution.
- Tabu Search scans the *whole neighborhood* of the current solution and picks the best neighboring solution as next solution.
- It will pick this solution even if it is worse than the current solution.
- Problem: This can easily lead to cycles (if the current solution is a local optimum, the search will go to a worse solution and then immediately back to the previous one, the local optimum).

Move to Best Solution

- Simulated Annealing, in each step, applies the unary search operation to create a (often randomly modified) copy of the current solution.
- Tabu Search scans the *whole neighborhood* of the current solution and picks the best neighboring solution as next solution.
- It will pick this solution even if it is worse than the current solution.
- Problem: This can easily lead to cycles (if the current solution is a local optimum, the search will go to a worse solution and then immediately back to the previous one, the local optimum).
- Solution: Introduce a *tabu criterion* which forbids certain solutions to be visited, to avoid re-visiting already seen solutions.

Tabu List

- The tabu list stores information about previously visited solutions in order to avoid visiting them again.

Tabu List

- The tabu list stores information about previously visited solutions in order to avoid visiting them again.
- Usually does not store complete solutions, but only features of solutions.

Tabu List

- The tabu list stores information about previously visited solutions in order to avoid visiting them again.
- Usually does not store complete solutions, but only features of solutions.
- These features often depend on the search moves

Tabu List

- The tabu list stores information about previously visited solutions in order to avoid visiting them again.
- Usually does not store complete solutions, but only features of solutions.
- These features often depend on the search moves:
  - If we scan an single-edge-exchange neighborhood of a tour for the Traveling Salesman Problem, we may simply forbid the removed edge from being inserted again.

Tabu List

- The tabu list stores information about previously visited solutions in order to avoid visiting them again.
- Usually does not store complete solutions, but only features of solutions.
- These features often depend on the search moves:
  - If we scan an single-edge-exchange neighborhood of a tour for the Traveling Salesman Problem, we may simply forbid the removed edge from being inserted again.
  - If we scan a single-bit-flip neighborhood in a MAX-SAT problem, we simply may forbit the same variable from being flipped again.

Tabu List

- The tabu list stores information about previously visited solutions in order to avoid visiting them again.
- Usually does not store complete solutions, but only features of solutions.
- These features often depend on the search moves:
  - If we scan an single-edge-exchange neighborhood of a tour for the Traveling Salesman Problem, we may simply forbid the removed edge from being inserted again.
  - If we scan a single-bit-flip neighborhood in a MAX-SAT problem, we simply may forbit the same variable from being flipped again.
  - More generally: If we reach a new solution p_{new} via search move $move$, we may either forbit the inverse move $\overline{move}$ or any move touching the same decision variables.

Tabu List

- The tabu list stores information about previously visited solutions in order to avoid visiting them again.
- Usually does not store complete solutions, but only features of solutions.
- These features often depend on the search moves:
  - If we scan an single-edge-exchange neighborhood of a tour for the Traveling Salesman Problem, we may simply forbid the removed edge from being inserted again.
  - If we scan a single-bit-flip neighborhood in a MAX-SAT problem, we simply may forbit the same variable from being flipped again.
  - More generally: If we reach a new solution p_{new} via search move $move$, we may either forbit the inverse move $\overline{move}$ or any move touching the same decision variables.
- Store features of tt most recently visited solutions tt is called *tabu tenure* or *tabu list length*).

Tabu List

- The tabu list stores information about previously visited solutions in order to avoid visiting them again.
- Usually does not store complete solutions, but only features of solutions.
- These features often depend on the search moves:
  - If we scan an single-edge-exchange neighborhood of a tour for the Traveling Salesman Problem, we may simply forbid the removed edge from being inserted again.
  - If we scan a single-bit-flip neighborhood in a MAX-SAT problem, we simply may forbit the same variable from being flipped again.
  - More generally: If we reach a new solution p_{new} via search move $move$, we may either forbit the inverse move $\overline{move}$ or any move touching the same decision variables.
- Store features of tt most recently visited solutions tt is called *tabu tenure* or *tabu list length*).
- Solutions with features from the tabu list are forbidden.

Tabu List

- The tabu list stores information about previously visited solutions in order to avoid visiting them again.
- Usually does not store complete solutions, but only features of solutions.
- These features often depend on the search moves:
  - If we scan an single-edge-exchange neighborhood of a tour for the Traveling Salesman Problem, we may simply forbid the removed edge from being inserted again.
  - If we scan a single-bit-flip neighborhood in a MAX-SAT problem, we simply may forbit the same variable from being flipped again.
  - More generally: If we reach a new solution p_{new} via search move $move$, we may either forbit the inverse move $\overline{move}$ or any move touching the same decision variables.
- Store features of tt most recently visited solutions tt is called *tabu tenure* or *tabu list length*).
- Solutions with features from the tabu list are forbidden.
- Choice of tt has big influence on performance.

Aspiration Criterion

- Tabu criterion may also prevent previously unseen solutions from being explored.

Aspiration Criterion

- Tabu criterion may also prevent previously unseen solutions from being explored.
- Some of these might be better than the best solution we have found so far

- Tabu criterion may also prevent previously unseen solutions from being explored.
- Some of these might be better than the best solution we have found so far, i.e., very interesting regardless whether they are tabu or not…
- *Aspiration criteria*: criteria that override the tabu criterion and allow the search to move to a solution even if it is tabu.

Putting it Together

$p_{best} \longleftarrow \text{tabuSearch}(f, tt)$

Input: f: the objective function subject to minization
Input: [implicit] shouldTerminate: the termination criterion
Data: p_{new}: the new solution to be tested
Data: p_{cur}: the current solution
Data: $move$: the $move$ reaching p_{test}
Data: $move_b$: the $move$ reaching p_{new}
Output: p_{best}: the best individual ever discovered

```
begin
    p_best.x ⟵ create initial solution
    p_best.y ⟵ f(p_best.x)
    p_cur.y ⟵ p_best
    tabu ⟵ empty list
    while ¬(shouldTerminate ∨ (p_cur ≠ ∅)) do
        p_new ⟵ ∅
        foreach p_test ∈ neighborhood of p_cur do
            p_test.y ⟵ f(p_test.x)
            if
              ((move ∉ tabu) ∧ ((p_new = ∅) ∨ (p_test.y < p_new.y)))∨
              (p_test.y ≤ p_best.y)
            then
                p_new ⟵ p_test
                move_b ⟵ move
        p_cur ⟵ p_new
        if (p_cur ≠ ∅) then
            if p_cur.y ≤ p_best.y then p_best ⟵ p_cur

            append ‾move_b‾ to tabu
            if length of tabu ≥ tt then remove oldest element from tabu
    return p_best
```

- We assume a simple Tabu Search where

Putting it Together

$p_{best} \longleftarrow \text{tabuSearch}(f, tt)$

Input: f: the objective function subject to minization
Input: [implicit] shouldTerminate: the termination criterion
Data: p_{new}: the new solution to be tested
Data: p_{cur}: the current solution
Data: $move$: the $move$ reaching p_{test}
Data: $move_b$: the $move$ reaching p_{new}
Output: p_{best}: the best individual ever discovered

```
begin
    p_best.x ⟵ create initial solution
    p_best.y ⟵ f(p_best.x)
    p_cur.y ⟵ p_best
    tabu ⟵ empty list
    while ¬(shouldTerminate ∨ (p_cur ≠ ∅)) do
        p_new ⟵ ∅
        foreach p_test ∈ neighborhood of p_cur do
            p_test.y ⟵ f(p_test.x)
            if
             ((move ∉ tabu) ∧ ((p_new = ∅) ∨ (p_test.y < p_new.y)))∨
             (p_test.y ≤ p_best.y)
            then
                p_new ⟵ p_test
                move_b ⟵ move
        p_cur ⟵ p_new
        if (p_cur ≠ ∅) then
            if p_cur.y ≤ p_best.y then p_best ⟵ p_cur

            append ‾move_b‾ to tabu
            if length of tabu ≥ tt then remove oldest element from tabu
    return p_best
```

- We assume a simple Tabu Search where
  - search- and solution space are the same ($\mathbb{G} = \mathbb{X}$)

Putting it Together

$p_{best} \longleftarrow \mathrm{tabuSearch}(f, tt)$

Input: f: the objective function subject to minization
Input: [implicit] $\mathrm{shouldTerminate}$: the termination criterion
Data: p_{new}: the new solution to be tested
Data: p_{cur}: the current solution
Data: $move$: the $move$ reaching p_{test}
Data: $move_b$: the $move$ reaching p_{new}
Output: p_{best}: the best individual ever discovered

begin
- $p_{best}.x \longleftarrow$ create initial solution
- $p_{best}.y \longleftarrow f(p_{best}.x)$
- $p_{cur}.y \longleftarrow p_{best}$
- $tabu \longleftarrow$ empty list
- **while** $\neg(shouldTerminate \vee (p_{cur} \neq \emptyset))$ **do**
  - $p_{new} \longleftarrow \emptyset$
  - **foreach** $p_{test} \in$ neighborhood of p_{cur} **do**
    - $p_{test}.y \longleftarrow f(p_{test}.x)$
    - **if** $((move \notin tabu) \wedge ((p_{new} = \emptyset) \vee (p_{test}.y < p_{new}.y))) \vee (p_{test}.y \leq p_{best}.y)$ **then**
      - $p_{new} \longleftarrow p_{test}$
      - $move_b \longleftarrow move$
  - $p_{cur} \longleftarrow p_{new}$
  - **if** $(p_{cur} \neq \emptyset)$ **then**
    - **if** $p_{cur}.y \leq p_{best}.y$ **then** $p_{best} \longleftarrow p_{cur}$
    - append $\overline{move_b}$ to $tabu$
    - **if** length of $tabu \geq tt$ **then** remove oldest element from $tabu$
- **return** p_{best}

- We assume a simple Tabu Search where
  - search- and solution space are the same ($\mathbb{G} = \mathbb{X}$) and
  - where the tabu criterion is the applied search move

Putting it Together

$p_{best} \longleftarrow \text{tabuSearch}(f, tt)$

Input: f: the objective function subject to minization
Input: [implicit] shouldTerminate: the termination criterion
Data: p_{new}: the new solution to be tested
Data: p_{cur}: the current solution
Data: $move$: the $move$ reaching p_{test}
Data: $move_b$: the $move$ reaching p_{new}
Output: p_{best}: the best individual ever discovered

```
begin
    p_best.x ⟵ create initial solution
    p_best.y ⟵ f(p_best.x)
    p_cur.y ⟵ p_best
    tabu ⟵ empty list
    while ¬(shouldTerminate ∨ (p_cur ≠ ∅)) do
        p_new ⟵ ∅
        foreach p_test ∈ neighborhood of p_cur do
            p_test.y ⟵ f(p_test.x)
            if
              ((move ∉ tabu) ∧ ((p_new = ∅) ∨ (p_test.y < p_new.y)))∨
              (p_test.y ≤ p_best.y)
              then
                p_new ⟵ p_test
                move_b ⟵ move
        p_cur ⟵ p_new
        if (p_cur ≠ ∅) then
            if p_cur.y ≤ p_best.y then p_best ⟵ p_cur

            append ‾move_b‾ to tabu
            if length of tabu ≥ tt then remove oldest element from tabu
    return p_best
```

- We assume a simple Tabu Search where
  - search- and solution space are the same ($\mathbb{G} = \mathbb{X}$) and
  - where the tabu criterion is the applied search move and
  - where the aspiration criterion is that any solution better than best currently known solution p_{best} will always be accepted

Putting it Together

$p_{best} \longleftarrow \text{tabuSearch}(f, tt)$

Input: f: the objective function subject to minization
Input: [implicit] shouldTerminate: the termination criterion
Data: p_{new}: the new solution to be tested
Data: p_{cur}: the current solution
Data: $move$: the $move$ reaching p_{test}
Data: $move_b$: the $move$ reaching p_{new}
Output: p_{best}: the best individual ever discovered

```
begin
    p_best.x ⟵ create initial solution
    p_best.y ⟵ f(p_best.x)
    p_cur.y ⟵ p_best
    tabu ⟵ empty list
    while ¬(shouldTerminate ∨ (p_cur ≠ ∅)) do
        p_new ⟵ ∅
        foreach p_test ∈ neighborhood of p_cur do
            p_test.y ⟵ f(p_test.x)
            if
              ((move ∉ tabu) ∧ ((p_new = ∅) ∨ (p_test.y < p_new.y)))∨
              (p_test.y ≤ p_best.y)
             then
                p_new ⟵ p_test
                move_b ⟵ move
        p_cur ⟵ p_new
        if (p_cur ≠ ∅) then
            if p_cur.y ≤ p_best.y then p_best ⟵ p_cur

            append ‾move_b‾ to tabu
            if length of tabu ≥ tt then remove oldest element from tabu
    return p_best
```

- We start by creating the starting point of our search (here directly in form of candidate solution $p_{cur}.x$).

Putting it Together

$p_{best} \longleftarrow \text{tabuSearch}(f, tt)$

Input: f: the objective function subject to minization
Input: [implicit] shouldTerminate: the termination criterion
Data: p_{new}: the new solution to be tested
Data: p_{cur}: the current solution
Data: $move$: the $move$ reaching p_{test}
Data: $move_b$: the $move$ reaching p_{new}
Output: p_{best}: the best individual ever discovered

```
begin
    p_best.x ⟵ create initial solution
    p_best.y ⟵ f(p_best.x)
    p_cur.y ⟵ p_best
    tabu ⟵ empty list
    while ¬(shouldTerminate ∨ (p_cur ≠ ∅)) do
        p_new ⟵ ∅
        foreach p_test ∈ neighborhood of p_cur do
            p_test.y ⟵ f(p_test.x)
            if
              ((move ∉ tabu) ∧ ((p_new = ∅) ∨ (p_test.y < p_new.y)))∨
              (p_test.y ≤ p_best.y)
             then
                p_new ⟵ p_test
                move_b ⟵ move
        p_cur ⟵ p_new
        if (p_cur ≠ ∅) then
            if p_cur.y ≤ p_best.y then p_best ⟵ p_cur

            append overline(move_b) to tabu
            if length of tabu ≥ tt then remove oldest element from tabu
    return p_best
```

- We start by creating the starting point of our search (here directly in form of candidate solution $p_{cur}.x$).
- This could happen randomly or via a simple logic (constructive heuristic)

Putting it Together

$p_{best} \longleftarrow \text{tabuSearch}(f, tt)$

Input: f: the objective function subject to minization
Input: [implicit] shouldTerminate: the termination criterion
Data: p_{new}: the new solution to be tested
Data: p_{cur}: the current solution
Data: $move$: the $move$ reaching p_{test}
Data: $move_b$: the $move$ reaching p_{new}
Output: p_{best}: the best individual ever discovered

```
begin
    p_best.x ⟵ create initial solution
    p_best.y ⟵ f(p_best.x)
    p_cur.y ⟵ p_best
    tabu ⟵ empty list
    while ¬(shouldTerminate ∨ (p_cur ≠ ∅)) do
        p_new ⟵ ∅
        foreach p_test ∈ neighborhood of p_cur do
            p_test.y ⟵ f(p_test.x)
            if
              ((move ∉ tabu) ∧ ((p_new = ∅) ∨ (p_test.y < p_new.y)))∨
              (p_test.y ≤ p_best.y)
             then
                p_new ⟵ p_test
                move_b ⟵ move
        p_cur ⟵ p_new
        if (p_cur ≠ ∅) then
            if p_cur.y ≤ p_best.y then p_best ⟵ p_cur

            append ‾move_b‾ to tabu
            if length of tabu ≥ tt then remove oldest element from tabu
    return p_best
```

- We compute the objective value $f(p_{cur}.x)$ of the initial solution and remember it in variable $p_{cur}.y$.

$p_{best} \longleftarrow \text{tabuSearch}(f, tt)$

Input: f: the objective function subject to minization
Input: [implicit] shouldTerminate: the termination criterion
Data: p_{new}: the new solution to be tested
Data: p_{cur}: the current solution
Data: $move$: the $move$ reaching p_{test}
Data: $move_b$: the $move$ reaching p_{new}
Output: p_{best}: the best individual ever discovered

```
begin
    p_best.x ⟵ create initial solution
    p_best.y ⟵ f(p_best.x)
    p_cur.y ⟵ p_best
    tabu ⟵ empty list
    while ¬(shouldTerminate ∨ (p_cur ≠ ∅)) do
        p_new ⟵ ∅
        foreach p_test ∈ neighborhood of p_cur do
            p_test.y ⟵ f(p_test.x)
            if
              ((move ∉ tabu) ∧ ((p_new = ∅) ∨ (p_test.y < p_new.y)))∨
              (p_test.y ≤ p_best.y)
             then
                p_new ⟵ p_test
                move_b ⟵ move
        p_cur ⟵ p_new
        if (p_cur ≠ ∅) then
            if p_cur.y ≤ p_best.y then p_best ⟵ p_cur

            append ‾move_b‾ to tabu
            if length of tabu ≥ tt then remove oldest element from tabu
    return p_best
```

- The initial solution p_{cur} is also the best solution p_{best} we know so far.

Putting it Together

$p_{best} \longleftarrow \text{tabuSearch}(f, tt)$

Input: f: the objective function subject to minization
Input: [implicit] shouldTerminate: the termination criterion
Data: p_{new}: the new solution to be tested
Data: p_{cur}: the current solution
Data: $move$: the $move$ reaching p_{test}
Data: $move_b$: the $move$ reaching p_{new}
Output: p_{best}: the best individual ever discovered

```
begin
    p_best.x ⟵ create initial solution
    p_best.y ⟵ f(p_best.x)
    p_cur.y ⟵ p_best
    tabu ⟵ empty list
    while ¬(shouldTerminate ∨ (p_cur ≠ ∅)) do
        p_new ⟵ ∅
        foreach p_test ∈ neighborhood of p_cur do
            p_test.y ⟵ f(p_test.x)
            if
              ((move ∉ tabu) ∧ ((p_new = ∅) ∨ (p_test.y < p_new.y)))∨
              (p_test.y ≤ p_best.y)
            then
                p_new ⟵ p_test
                move_b ⟵ move
        p_cur ⟵ p_new
        if (p_cur ≠ ∅) then
            if p_cur.y ≤ p_best.y then p_best ⟵ p_cur

            append ‾move_b‾ to tabu
            if length of tabu ≥ tt then remove oldest element from tabu
    return p_best
```

- Initially, the tabu list $tabu$ is empty, everything is allowed.

Putting it Together

$p_{best} \longleftarrow \text{tabuSearch}(f, tt)$

Input: f: the objective function subject to minization
Input: [implicit] shouldTerminate: the termination criterion
Data: p_{new}: the new solution to be tested
Data: p_{cur}: the current solution
Data: $move$: the $move$ reaching p_{test}
Data: $move_b$: the $move$ reaching p_{new}
Output: p_{best}: the best individual ever discovered

```
begin
    p_best.x ⟵ create initial solution
    p_best.y ⟵ f(p_best.x)
    p_cur.y ⟵ p_best
    tabu ⟵ empty list
    while ¬(shouldTerminate ∨ (p_cur ≠ ∅)) do
        p_new ⟵ ∅
        foreach p_test ∈ neighborhood of p_cur do
            p_test.y ⟵ f(p_test.x)
            if
              ((move ∉ tabu) ∧ ((p_new = ∅) ∨ (p_test.y < p_new.y)))∨
              (p_test.y ≤ p_best.y)
            then
                p_new ⟵ p_test
                move_b ⟵ move
        p_cur ⟵ p_new
        if (p_cur ≠ ∅) then
            if p_cur.y ≤ p_best.y then p_best ⟵ p_cur

            append ‾move_b‾ to tabu
            if length of tabu ≥ tt then remove oldest element from tabu
    return p_best
```

- In every iteration, we first check the termination criterion whether we should quit.

Putting it Together

$p_{best} \longleftarrow \text{tabuSearch}(f, tt)$

Input: f: the objective function subject to minization
Input: [implicit] shouldTerminate: the termination criterion
Data: p_{new}: the new solution to be tested
Data: p_{cur}: the current solution
Data: $move$: the $move$ reaching p_{test}
Data: $move_b$: the $move$ reaching p_{new}
Output: p_{best}: the best individual ever discovered

```
begin
  p_best.x ⟵ create initial solution
  p_best.y ⟵ f(p_best.x)
  p_cur.y ⟵ p_best
  tabu ⟵ empty list
  while ¬(shouldTerminate ∨ (p_cur ≠ ∅)) do
    p_new ⟵ ∅
    foreach p_test ∈ neighborhood of p_cur do
      p_test.y ⟵ f(p_test.x)
      if
        ((move ∉ tabu) ∧ ((p_new = ∅) ∨ (p_test.y < p_new.y)))∨
        (p_test.y ≤ p_best.y)
      then
        p_new ⟵ p_test
        move_b ⟵ move
    p_cur ⟵ p_new
    if (p_cur ≠ ∅) then
      if p_cur.y ≤ p_best.y then p_best ⟵ p_cur

      append ‾move_b‾ to tabu
      if length of tabu ≥ tt then remove oldest element from tabu
  return p_best
```

- We should also stop if all solutions surrounding our current solution are tabu and the aspiration criterion does not hold for any, i.e., if there is no next solution to move to.

Putting it Together

$p_{best} \longleftarrow \text{tabuSearch}(f, tt)$

Input: f: the objective function subject to minization
Input: [implicit] shouldTerminate: the termination criterion
Data: p_{new}: the new solution to be tested
Data: p_{cur}: the current solution
Data: $move$: the $move$ reaching p_{test}
Data: $move_b$: the $move$ reaching p_{new}
Output: p_{best}: the best individual ever discovered

begin
- $p_{best}.x \longleftarrow$ create initial solution
- $p_{best}.y \longleftarrow f(p_{best}.x)$
- $p_{cur}.y \longleftarrow p_{best}$
- $tabu \longleftarrow$ empty list
- **while** $\neg\,(shouldTerminate \lor (p_{cur} \neq \emptyset))$ **do**
  - $p_{new} \longleftarrow \emptyset$
  - **foreach** $p_{test} \in$ neighborhood of p_{cur} **do**
    - $p_{test}.y \longleftarrow f(p_{test}.x)$
    - **if** $((move \notin tabu) \land ((p_{new} = \emptyset) \lor (p_{test}.y < p_{new}.y)))\lor$ $(p_{test}.y \leq p_{best}.y)$ **then**
      - $p_{new} \longleftarrow p_{test}$
      - $move_b \longleftarrow move$
  - $p_{cur} \longleftarrow p_{new}$
  - **if** $(p_{cur} \neq \emptyset)$ **then**
    - **if** $p_{cur}.y \leq p_{best}.y$ **then** $p_{best} \longleftarrow p_{cur}$
    - append $\overline{move_b}$ to $tabu$
    - **if** length of $tabu \geq tt$ **then** remove oldest element from $tabu$
- **return** p_{best}

- In each step, we first assume that there is no solution p_{new} we can move to from p_{cur}.

Putting it Together

$p_{best} \longleftarrow \mathrm{tabuSearch}(f, tt)$

Input: f: the objective function subject to minization
Input: [implicit] shouldTerminate: the termination criterion
Data: p_{new}: the new solution to be tested
Data: p_{cur}: the current solution
Data: $move$: the $move$ reaching p_{test}
Data: $move_b$: the $move$ reaching p_{new}
Output: p_{best}: the best individual ever discovered

```
begin
    p_best.x ⟵ create initial solution
    p_best.y ⟵ f(p_best.x)
    p_cur.y ⟵ p_best
    tabu ⟵ empty list
    while ¬(shouldTerminate ∨ (p_cur ≠ ∅)) do
        p_new ⟵ ∅
        foreach p_test ∈ neighborhood of p_cur do
            p_test.y ⟵ f(p_test.x)
            if
              ((move ∉ tabu) ∧ ((p_new = ∅) ∨ (p_test.y < p_new.y)))∨
              (p_test.y ≤ p_best.y)
             then
                p_new ⟵ p_test
                move_b ⟵ move
        p_cur ⟵ p_new
        if (p_cur ≠ ∅) then
            if p_cur.y ≤ p_best.y then p_best ⟵ p_cur

            append ‾move_b‾ to tabu
            if length of tabu ≥ tt then remove oldest element from tabu
    return p_best
```

- We then scan the complete neighborhood of p_{cur}.

Putting it Together

$p_{best} \longleftarrow \mathrm{tabuSearch}(f, tt)$

Input: f: the objective function subject to minization
Input: [implicit] $\mathrm{shouldTerminate}$: the termination criterion
Data: p_{new}: the new solution to be tested
Data: p_{cur}: the current solution
Data: $move$: the $move$ reaching p_{test}
Data: $move_b$: the $move$ reaching p_{new}
Output: p_{best}: the best individual ever discovered

```
begin
  p_best.x ⟵ create initial solution
  p_best.y ⟵ f(p_best.x)
  p_cur.y ⟵ p_best
  tabu ⟵ empty list
  while ¬(shouldTerminate ∨ (p_cur ≠ ∅)) do
    p_new ⟵ ∅
    foreach p_test ∈ neighborhood of p_cur do
      p_test.y ⟵ f(p_test.x)
      if
        ((move ∉ tabu) ∧ ((p_new = ∅) ∨ (p_test.y < p_new.y)))∨
        (p_test.y ≤ p_best.y)
      then
        p_new ⟵ p_test
        move_b ⟵ move
    p_cur ⟵ p_new
    if (p_cur ≠ ∅) then
      if p_cur.y ≤ p_best.y then p_best ⟵ p_cur

      append ‾move_b‾ to tabu
      if length of tabu ≥ tt then remove oldest element from tabu
  return p_best
```

- We then scan the complete neighborhood of p_{cur}.
- This neighborhood is defined by possible search moves $move$ that can be applied to the current candidate solution $p_{cur}.x$ (again, here we assume that $\mathbb{G} = \mathbb{X}$).

Putting it Together

$p_{best} \longleftarrow \mathrm{tabuSearch}(f, tt)$

Input: f: the objective function subject to minization
Input: [implicit] $\mathrm{shouldTerminate}$: the termination criterion
Data: p_{new}: the new solution to be tested
Data: p_{cur}: the current solution
Data: $move$: the $move$ reaching p_{test}
Data: $move_b$: the $move$ reaching p_{new}
Output: p_{best}: the best individual ever discovered

```
begin
    p_best.x ⟵ create initial solution
    p_best.y ⟵ f(p_best.x)
    p_cur.y ⟵ p_best
    tabu ⟵ empty list
    while ¬(shouldTerminate ∨ (p_cur ≠ ∅)) do
        p_new ⟵ ∅
        foreach p_test ∈ neighborhood of p_cur do
            p_test.y ⟵ f(p_test.x)
            if
              ((move ∉ tabu) ∧ ((p_new = ∅) ∨ (p_test.y < p_new.y)))∨
              (p_test.y ≤ p_best.y)
            then
                p_new ⟵ p_test
                move_b ⟵ move
        p_cur ⟵ p_new
        if (p_cur ≠ ∅) then
            if p_cur.y ≤ p_best.y then p_best ⟵ p_cur

            append ‾move_b‾ to tabu
            if length of tabu ≥ tt then remove oldest element from tabu
    return p_best
```

- We then scan the complete neighborhood of p_{cur}.
- This neighborhood is defined by possible search moves $move$ that can be applied to the current candidate solution $p_{cur}.x$ (again, here we assume that $\mathbb{G} = \mathbb{X}$).
- For example, if our candidate solutions are strings of n bits, a neighborhood could be any string that can be reached by flipping a single bit in $p_{cur}.x$ (and this neighborhood would contain n other solutions $p_{test}.x$).

$p_{best} \longleftarrow \text{tabuSearch}(f, tt)$

Input: f: the objective function subject to minization
Input: [implicit] shouldTerminate: the termination criterion
Data: p_{new}: the new solution to be tested
Data: p_{cur}: the current solution
Data: $move$: the $move$ reaching p_{test}
Data: $move_b$: the $move$ reaching p_{new}
Output: p_{best}: the best individual ever discovered

```
begin
    p_best.x ⟵ create initial solution
    p_best.y ⟵ f(p_best.x)
    p_cur.y ⟵ p_best
    tabu ⟵ empty list
    while ¬(shouldTerminate ∨ (p_cur ≠ ∅)) do
        p_new ⟵ ∅
        foreach p_test ∈ neighborhood of p_cur do
            p_test.y ⟵ f(p_test.x)
            if
              ((move ∉ tabu) ∧ ((p_new = ∅) ∨ (p_test.y < p_new.y)))∨
              (p_test.y ≤ p_best.y)
            then
                p_new ⟵ p_test
                move_b ⟵ move
        p_cur ⟵ p_new
        if (p_cur ≠ ∅) then
            if p_cur.y ≤ p_best.y then p_best ⟵ p_cur

            append ‾move_b‾ to tabu
            if length of tabu ≥ tt then remove oldest element from tabu
    return p_best
```

- We compute the objective value $f(p_{test}.x)$ of the initial solution and remember it in variable $p_{test}.y$.

Putting it Together

$p_{best} \longleftarrow \text{tabuSearch}(f, tt)$

Input: f: the objective function subject to minization
Input: [implicit] shouldTerminate: the termination criterion
Data: p_{new}: the new solution to be tested
Data: p_{cur}: the current solution
Data: $move$: the $move$ reaching p_{test}
Data: $move_b$: the $move$ reaching p_{new}
Output: p_{best}: the best individual ever discovered

begin
- $p_{best}.x \longleftarrow$ create initial solution
- $p_{best}.y \longleftarrow f(p_{best}.x)$
- $p_{cur}.y \longleftarrow p_{best}$
- $tabu \longleftarrow$ empty list
- **while** $\neg(shouldTerminate \vee (p_{cur} \neq \emptyset))$ **do**
  - $p_{new} \longleftarrow \emptyset$
  - **foreach** $p_{test} \in$ neighborhood of p_{cur} **do**
    - $p_{test}.y \longleftarrow f(p_{test}.x)$
    - **if** $((move \notin tabu) \wedge ((p_{new} = \emptyset) \vee (p_{test}.y < p_{new}.y))) \vee$ $(p_{test}.y \leq p_{best}.y)$ **then**
      - $p_{new} \longleftarrow p_{test}$
      - $move_b \longleftarrow move$
  - $p_{cur} \longleftarrow p_{new}$
  - **if** $(p_{cur} \neq \emptyset)$ **then**
    - **if** $p_{cur}.y \leq p_{best}.y$ **then** $p_{best} \longleftarrow p_{cur}$
    - append $\overline{move_b}$ to $tabu$
    - **if** length of $tabu \geq tt$ **then** remove oldest element from $tabu$
- **return** p_{best}

- p_{test} would be a candidate for the next step of our search

Putting it Together

$p_{best} \longleftarrow \text{tabuSearch}(f, tt)$

Input: f: the objective function subject to minization
Input: [implicit] shouldTerminate: the termination criterion
Data: p_{new}: the new solution to be tested
Data: p_{cur}: the current solution
Data: $move$: the $move$ reaching p_{test}
Data: $move_b$: the $move$ reaching p_{new}
Output: p_{best}: the best individual ever discovered

```
begin
    p_best.x ⟵ create initial solution
    p_best.y ⟵ f(p_best.x)
    p_cur.y ⟵ p_best
    tabu ⟵ empty list
    while ¬(shouldTerminate ∨ (p_cur ≠ ∅)) do
        p_new ⟵ ∅
        foreach p_test ∈ neighborhood of p_cur do
            p_test.y ⟵ f(p_test.x)
            if
              ((move ∉ tabu) ∧ ((p_new = ∅) ∨ (p_test.y < p_new.y)))∨
              (p_test.y ≤ p_best.y)
             then
                p_new ⟵ p_test
                move_b ⟵ move
        p_cur ⟵ p_new
        if (p_cur ≠ ∅) then
            if p_cur.y ≤ p_best.y then p_best ⟵ p_cur

            append ‾move_b‾ to tabu
            if length of tabu ≥ tt then remove oldest element from tabu
    return p_best
```

- p_{test} would be a candidate for the next step of our search if and only if
    1. the move $move$ leading to it from p_{cur} is not tabu

Putting it Together

$p_{best} \longleftarrow \mathrm{tabuSearch}(f, tt)$

Input: f: the objective function subject to minization
Input: [implicit] $\mathrm{shouldTerminate}$: the termination criterion
Data: p_{new}: the new solution to be tested
Data: p_{cur}: the current solution
Data: $move$: the $move$ reaching p_{test}
Data: $move_b$: the $move$ reaching p_{new}
Output: p_{best}: the best individual ever discovered

```
begin
    p_best.x ⟵ create initial solution
    p_best.y ⟵ f(p_best.x)
    p_cur.y ⟵ p_best
    tabu ⟵ empty list
    while ¬(shouldTerminate ∨ (p_cur ≠ ∅)) do
        p_new ⟵ ∅
        foreach p_test ∈ neighborhood of p_cur do
            p_test.y ⟵ f(p_test.x)
            if
             ((move ∉ tabu) ∧ ((p_new = ∅) ∨ (p_test.y < p_new.y)))∨
             (p_test.y ≤ p_best.y)
            then
                p_new ⟵ p_test
                move_b ⟵ move
        p_cur ⟵ p_new
        if (p_cur ≠ ∅) then
            if p_cur.y ≤ p_best.y then p_best ⟵ p_cur

            append ‾move_b‾ to tabu
            if length of tabu ≥ tt then remove oldest element from tabu
    return p_best
```

- p_{test} would be a candidate for the next step of our search if and only if
  1. the move $move$ leading to it from p_{cur} is not tabu and
     1. it is better than the currently best acceptable neighbor p_{new}

Putting it Together

$p_{best} \longleftarrow \text{tabuSearch}(f, tt)$

Input: f: the objective function subject to minization
Input: [implicit] shouldTerminate: the termination criterion
Data: p_{new}: the new solution to be tested
Data: p_{cur}: the current solution
Data: $move$: the $move$ reaching p_{test}
Data: $move_b$: the $move$ reaching p_{new}
Output: p_{best}: the best individual ever discovered

```
begin
  p_best.x ⟵ create initial solution
  p_best.y ⟵ f(p_best.x)
  p_cur.y ⟵ p_best
  tabu ⟵ empty list
  while ¬(shouldTerminate ∨ (p_cur ≠ ∅)) do
    p_new ⟵ ∅
    foreach p_test ∈ neighborhood of p_cur do
      p_test.y ⟵ f(p_test.x)
      if
       ((move ∉ tabu) ∧ ((p_new = ∅) ∨ (p_test.y < p_new.y)))∨
       (p_test.y ≤ p_best.y)
      then
        p_new ⟵ p_test
        move_b ⟵ move
    p_cur ⟵ p_new
    if (p_cur ≠ ∅) then
      if p_cur.y ≤ p_best.y then p_best ⟵ p_cur

      append ‾move_b‾ to tabu
      if length of tabu ≥ tt then remove oldest element from tabu
  return p_best
```

- p_{test} would be a candidate for the next step of our search if and only if
  1. the move $move$ leading to it from p_{cur} is not tabu and
     1. it is better than the currently best acceptable neighbor p_{new} or
     2. it is the first acceptable neighbor.

Putting it Together

$p_{best} \longleftarrow \mathrm{tabuSearch}(f, tt)$

Input: f: the objective function subject to minization
Input: [implicit] $\mathrm{shouldTerminate}$: the termination criterion
Data: p_{new}: the new solution to be tested
Data: p_{cur}: the current solution
Data: $move$: the $move$ reaching p_{test}
Data: $move_b$: the $move$ reaching p_{new}
Output: p_{best}: the best individual ever discovered

```
begin
    p_best.x ⟵ create initial solution
    p_best.y ⟵ f(p_best.x)
    p_cur.y ⟵ p_best
    tabu ⟵ empty list
    while ¬(shouldTerminate ∨ (p_cur ≠ ∅)) do
        p_new ⟵ ∅
        foreach p_test ∈ neighborhood of p_cur do
            p_test.y ⟵ f(p_test.x)
            if
              ((move ∉ tabu) ∧ ((p_new = ∅) ∨ (p_test.y < p_new.y)))∨
              (p_test.y ≤ p_best.y)
              then
                p_new ⟵ p_test
                move_b ⟵ move
        p_cur ⟵ p_new
        if (p_cur ≠ ∅) then
            if p_cur.y ≤ p_best.y then p_best ⟵ p_cur

            append ‾move_b‾ to tabu
            if length of tabu ≥ tt then remove oldest element from tabu
    return p_best
```

- p_{test} would be a candidate for the next step of our search if and only if
  1. the move $move$ leading to it from p_{cur} is not tabu and
     1. it is better than the currently best acceptable neighbor p_{new} or
     2. it is the first acceptable neighbor.
  2. or the aspiration criterion kicks in, which here means that it is better than the best solution p_{best} we have ever seen.

Putting it Together

$p_{best} \longleftarrow \mathrm{tabuSearch}(f, tt)$

Input: f: the objective function subject to minization
Input: [implicit] $\mathrm{shouldTerminate}$: the termination criterion
Data: p_{new}: the new solution to be tested
Data: p_{cur}: the current solution
Data: $move$: the $move$ reaching p_{test}
Data: $move_b$: the $move$ reaching p_{new}
Output: p_{best}: the best individual ever discovered

```
begin
    p_best.x ⟵ create initial solution
    p_best.y ⟵ f(p_best.x)
    p_cur.y ⟵ p_best
    tabu ⟵ empty list
    while ¬(shouldTerminate ∨ (p_cur ≠ ∅)) do
        p_new ⟵ ∅
        foreach p_test ∈ neighborhood of p_cur do
            p_test.y ⟵ f(p_test.x)
            if
              ((move ∉ tabu) ∧ ((p_new = ∅) ∨ (p_test.y < p_new.y)))∨
              (p_test.y ≤ p_best.y)
             then
                p_new ⟵ p_test
                move_b ⟵ move
        p_cur ⟵ p_new
        if (p_cur ≠ ∅) then
            if p_cur.y ≤ p_best.y then p_best ⟵ p_cur

            append ‾move_b‾ to tabu
            if length of tabu ≥ tt then remove oldest element from tabu
    return p_best
```

- In this case, we
  - remember it in variable p_{new}

Putting it Together

$p_{best} \longleftarrow \text{tabuSearch}(f, tt)$

Input: f: the objective function subject to minization
Input: [implicit] shouldTerminate: the termination criterion
Data: p_{new}: the new solution to be tested
Data: p_{cur}: the current solution
Data: $move$: the $move$ reaching p_{test}
Data: $move_b$: the $move$ reaching p_{new}
Output: p_{best}: the best individual ever discovered

begin
- $p_{best}.x \longleftarrow$ create initial solution
- $p_{best}.y \longleftarrow f(p_{best}.x)$
- $p_{cur}.y \longleftarrow p_{best}$
- $tabu \longleftarrow$ empty list
- **while** $\neg\,(shouldTerminate \vee (p_{cur} \neq \emptyset))$ **do**
  - $p_{new} \longleftarrow \emptyset$
  - **foreach** $p_{test} \in$ neighborhood of p_{cur} **do**
    - $p_{test}.y \longleftarrow f(p_{test}.x)$
    - **if** $((move \notin tabu) \wedge ((p_{new} = \emptyset) \vee (p_{test}.y < p_{new}.y))) \vee (p_{test}.y \leq p_{best}.y)$ **then**
      - $p_{new} \longleftarrow p_{test}$
      - $move_b \longleftarrow move$
  - $p_{cur} \longleftarrow p_{new}$
  - **if** $(p_{cur} \neq \emptyset)$ **then**
    - **if** $p_{cur}.y \leq p_{best}.y$ **then** $p_{best} \longleftarrow p_{cur}$
    - append $\overline{move_b}$ to $tabu$
    - **if** length of $tabu \geq tt$ **then** remove oldest element from $tabu$
- **return** p_{best}

- In this case, we
  - remember it in variable p_{new} and
  - store the $move$ leading to it (coming from p_{cur}) in variable $move_b$.

Putting it Together

$p_{best} \longleftarrow \text{tabuSearch}(f, tt)$

Input: f: the objective function subject to minization
Input: [implicit] shouldTerminate: the termination criterion
Data: p_{new}: the new solution to be tested
Data: p_{cur}: the current solution
Data: $move$: the $move$ reaching p_{test}
Data: $move_b$: the $move$ reaching p_{new}
Output: p_{best}: the best individual ever discovered

```
begin
    p_best.x ⟵ create initial solution
    p_best.y ⟵ f(p_best.x)
    p_cur.y ⟵ p_best
    tabu ⟵ empty list
    while ¬(shouldTerminate ∨ (p_cur ≠ ∅)) do
        p_new ⟵ ∅
        foreach p_test ∈ neighborhood of p_cur do
            p_test.y ⟵ f(p_test.x)
            if
              ((move ∉ tabu) ∧ ((p_new = ∅) ∨ (p_test.y < p_new.y)))∨
              (p_test.y ≤ p_best.y)
             then
                p_new ⟵ p_test
                move_b ⟵ move
        p_cur ⟵ p_new
        if (p_cur ≠ ∅) then
            if p_cur.y ≤ p_best.y then p_best ⟵ p_cur

            append ‾move_b‾ to tabu
            if length of tabu ≥ tt then remove oldest element from tabu
    return p_best
```

- After we have scanned the whole neighborhood of p_{cur}, we store the best discovered acceptable solution p_{new} in p_{cur}. (This could also be nothing $\emptyset$...)

Putting it Together

$p_{best} \longleftarrow \text{tabuSearch}(f, tt)$

Input: f: the objective function subject to minization
Input: [implicit] shouldTerminate: the termination criterion
Data: p_{new}: the new solution to be tested
Data: p_{cur}: the current solution
Data: $move$: the $move$ reaching p_{test}
Data: $move_b$: the $move$ reaching p_{new}
Output: p_{best}: the best individual ever discovered

```
begin
    p_best.x ⟵ create initial solution
    p_best.y ⟵ f(p_best.x)
    p_cur.y ⟵ p_best
    tabu ⟵ empty list
    while ¬(shouldTerminate ∨ (p_cur ≠ ∅)) do
        p_new ⟵ ∅
        foreach p_test ∈ neighborhood of p_cur do
            p_test.y ⟵ f(p_test.x)
            if
              ((move ∉ tabu) ∧ ((p_new = ∅) ∨ (p_test.y < p_new.y)))∨
              (p_test.y ≤ p_best.y)
             then
                p_new ⟵ p_test
                move_b ⟵ move
        p_cur ⟵ p_new
        if (p_cur ≠ ∅) then
            if p_cur.y ≤ p_best.y then p_best ⟵ p_cur

            append ‾move_b‾ to tabu
            if length of tabu ≥ tt then remove oldest element from tabu
    return p_best
```

- If we actually found new acceptable point p_{cur}

Putting it Together

$p_{best} \longleftarrow \text{tabuSearch}(f, tt)$

Input: f: the objective function subject to minization
Input: [implicit] shouldTerminate: the termination criterion
Data: p_{new}: the new solution to be tested
Data: p_{cur}: the current solution
Data: $move$: the $move$ reaching p_{test}
Data: $move_b$: the $move$ reaching p_{new}
Output: p_{best}: the best individual ever discovered

```
begin
    p_best.x ⟵ create initial solution
    p_best.y ⟵ f(p_best.x)
    p_cur.y ⟵ p_best
    tabu ⟵ empty list
    while ¬(shouldTerminate ∨ (p_cur ≠ ∅)) do
        p_new ⟵ ∅
        foreach p_test ∈ neighborhood of p_cur do
            p_test.y ⟵ f(p_test.x)
            if
              ((move ∉ tabu) ∧ ((p_new = ∅) ∨ (p_test.y < p_new.y)))∨
              (p_test.y ≤ p_best.y)
            then
                p_new ⟵ p_test
                move_b ⟵ move
        p_cur ⟵ p_new
        if (p_cur ≠ ∅) then
            if p_cur.y ≤ p_best.y then p_best ⟵ p_cur

            append ¯move_b to tabu
            if length of tabu ≥ tt then remove oldest element from tabu
    return p_best
```

- If we actually found new acceptable point p_{cur}
  - We check if it is better than the best solution p_{best} we have ever found and, if so, store it in p_{best}.

Putting it Together

$p_{best} \longleftarrow \text{tabuSearch}(f, tt)$

Input: f: the objective function subject to minization
Input: [implicit] shouldTerminate: the termination criterion
Data: p_{new}: the new solution to be tested
Data: p_{cur}: the current solution
Data: $move$: the $move$ reaching p_{test}
Data: $move_b$: the $move$ reaching p_{new}
Output: p_{best}: the best individual ever discovered

```
begin
    p_best.x ⟵ create initial solution
    p_best.y ⟵ f(p_best.x)
    p_cur.y ⟵ p_best
    tabu ⟵ empty list
    while ¬(shouldTerminate ∨ (p_cur ≠ ∅)) do
        p_new ⟵ ∅
        foreach p_test ∈ neighborhood of p_cur do
            p_test.y ⟵ f(p_test.x)
            if
             ((move ∉ tabu) ∧ ((p_new = ∅) ∨ (p_test.y < p_new.y)))∨
              (p_test.y ≤ p_best.y)
             then
                p_new ⟵ p_test
                move_b ⟵ move
        p_cur ⟵ p_new
        if (p_cur ≠ ∅) then
            if p_cur.y ≤ p_best.y then p_best ⟵ p_cur

            append overline(move_b) to tabu
            if length of tabu ≥ tt then remove oldest element from tabu
    return p_best
```

- If we actually found new acceptable point p_{cur}
  - We store the inverse $\overline{move_b}$ of the move $move_b$ leading from the “old” p_{cur} to the “new” p_{cur} in the tabu list $tabu$ to prevent us from going back to the “old” p_{cur} in the next tt iterations.

Putting it Together

$p_{best} \longleftarrow \text{tabuSearch}(f, tt)$

Input: f: the objective function subject to minization
Input: [implicit] shouldTerminate: the termination criterion
Data: p_{new}: the new solution to be tested
Data: p_{cur}: the current solution
Data: $move$: the $move$ reaching p_{test}
Data: $move_b$: the $move$ reaching p_{new}
Output: p_{best}: the best individual ever discovered

```
begin
    p_best.x ⟵ create initial solution
    p_best.y ⟵ f(p_best.x)
    p_cur.y ⟵ p_best
    tabu ⟵ empty list
    while ¬(shouldTerminate ∨ (p_cur ≠ ∅)) do
        p_new ⟵ ∅
        foreach p_test ∈ neighborhood of p_cur do
            p_test.y ⟵ f(p_test.x)
            if
              ((move ∉ tabu) ∧ ((p_new = ∅) ∨ (p_test.y < p_new.y)))∨
              (p_test.y ≤ p_best.y)
            then
                p_new ⟵ p_test
                move_b ⟵ move
        p_cur ⟵ p_new
        if (p_cur ≠ ∅) then
            if p_cur.y ≤ p_best.y then p_best ⟵ p_cur

            append ‾move_b‾ to tabu
            if length of tabu ≥ tt then remove oldest element from tabu
    return p_best
```

- If we actually found new acceptable point p_{cur}
  - We store the inverse $\overline{move}_b$ of the move $move_b$ leading from the "old" p_{cur} to the "new" p_{cur} in the tabu list $tabu$ to prevent us from going back to the "old" p_{cur} in the next tt iterations.
  - If the tabu list $tabu$ is now longer than the tabu tenure tt, we delete the oldest element from it.

Putting it Together

$p_{best} \longleftarrow \text{tabuSearch}(f, tt)$

Input: f: the objective function subject to minization
Input: [implicit] shouldTerminate: the termination criterion
Data: p_{new}: the new solution to be tested
Data: p_{cur}: the current solution
Data: $move$: the $move$ reaching p_{test}
Data: $move_b$: the $move$ reaching p_{new}
Output: p_{best}: the best individual ever discovered

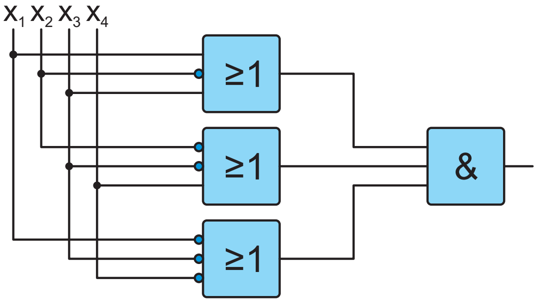
```
begin
    p_best.x ⟵ create initial solution
    p_best.y ⟵ f(p_best.x)
    p_cur.y ⟵ p_best
    tabu ⟵ empty list
    while ¬(shouldTerminate ∨ (p_cur ≠ ∅)) do
        p_new ⟵ ∅
        foreach p_test ∈ neighborhood of p_cur do
            p_test.y ⟵ f(p_test.x)
            if
              ((move ∉ tabu) ∧ ((p_new = ∅) ∨ (p_test.y < p_new.y)))∨
              (p_test.y ≤ p_best.y)
             then
                p_new ⟵ p_test
                move_b ⟵ move
        p_cur ⟵ p_new
        if (p_cur ≠ ∅) then
            if p_cur.y ≤ p_best.y then p_best ⟵ p_cur

            append ‾move_b‾ to tabu
            if length of tabu ≥ tt then remove oldest element from tabu
    return p_best
```

- Finally, if we have met the termination criterion shouldTerminate or there simply is no acceptable solution to go to anymore, we return the best solution p_{best} we found so far.

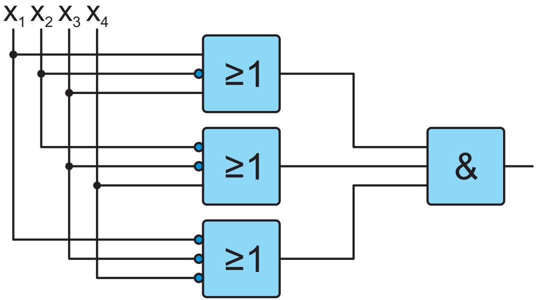
Section Outline

1. Introduction
2. Tabu Search
3. Example 1: MAX-SAT
4. Example 2: Traveling Salesman Problem
5. Iterated Local Search
6. Summary

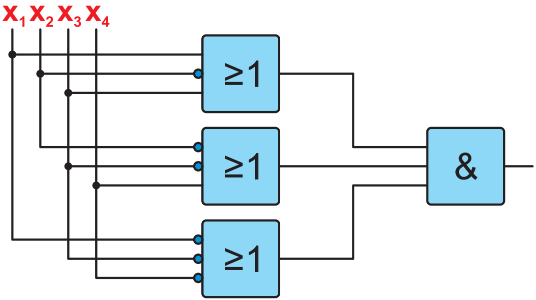
- Satisfiability Problems (SAT) [3]



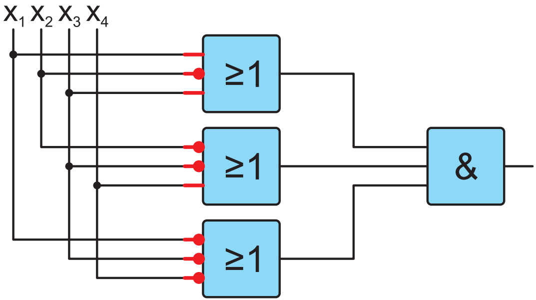
- Satisfiability Problems (SAT) [3]:
  - Given: Formula B in Boolean logic



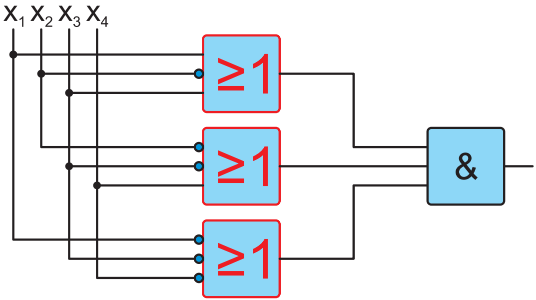
- Satisfiability Problems (SAT) [3]:
  - Given: Formula B in Boolean logic with of n Boolean variables $\vec{x} = (x_1, x_2, \dots, x_n)$



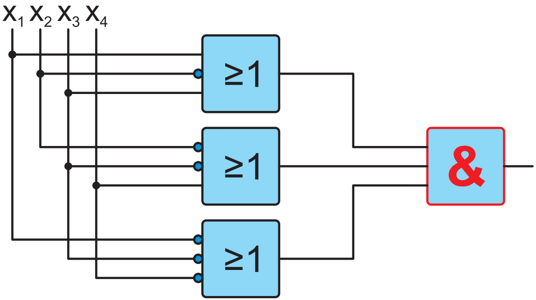
- Satisfiability Problems (SAT) [3]:
  - Given: Formula B in Boolean logic with of n Boolean variables $\vec{x} = (x_1, x_2, \dots, x_n)$, which appear either directly or negated



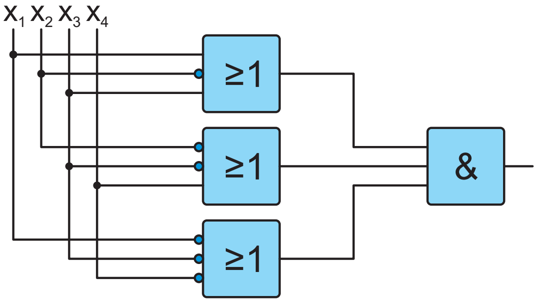
- Satisfiability Problems (SAT) [3]:
  - Given: Formula B in Boolean logic with of n Boolean variables $\vec{x} = (x_1, x_2, \dots, x_n)$, which appear either directly or negated in k “or” clauses



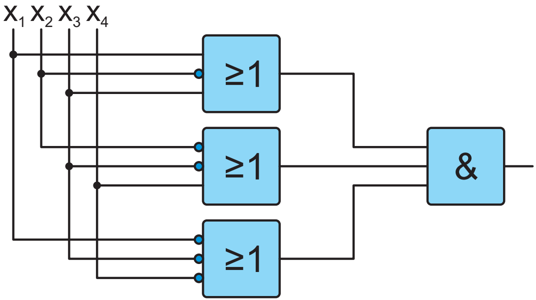
- Satisfiability Problems (SAT) [3]:
  - Given: Formula B in Boolean logic with of n Boolean variables $\vec{x} = (x_1, x_2, \dots, x_n)$, which appear either directly or negated in k "or" clauses, which are all combined with into one "and"



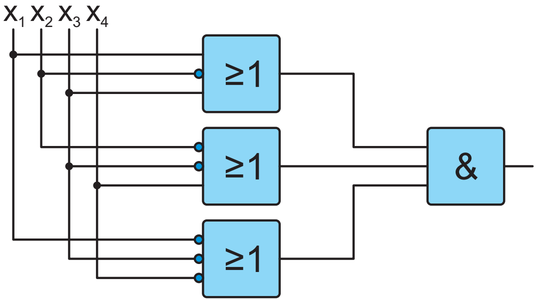
- Satisfiability Problems (SAT) [3]:
  - Given: Formula B in Boolean logic with of n Boolean variables $\vec{x} = (x_1, x_2, \dots, x_n)$, which appear either directly or negated in k "or" clauses, which are all combined with into one "and"
  - SAT Goal: find a setting for these variables so that B becomes true



- Maximum Satisfiability Problems (SAT) [4]:
  - Given: Formula B in Boolean logic with of n Boolean variables $\vec{x} = (x_1, x_2, \dots, x_n)$, which appear either directly or negated in k "or" clauses, which are all combined with into one "and"
  - MAX-SAT Goal [4]: minimize objective function $f(\vec{x})$ = number of clauses which are false.



- Maximum Satisfiability Problems (SAT) [4]:
  - Given: Formula B in Boolean logic with of n Boolean variables $\vec{x} = (x_1, x_2, \dots, x_n)$, which appear either directly or negated in k "or" clauses, which are all combined with into one "and"
  - MAX-SAT Goal [4]: minimize objective function $f(\vec{x})$ = number of clauses which are `false`.
  - $f(\vec{x}) = 0 \Longrightarrow$ all clauses are `true`, SAT problem solved



Example: MAX-SAT

- Maximum Satisfiability Problems (SAT) [4]:
  - Given: Formula B in Boolean logic with of n Boolean variables $\vec{x} = (x_1, x_2, \dots, x_n)$, which appear either directly or negated in k "or" clauses, which are all combined with into one "and"
  - MAX-SAT Goal [4]: minimize objective function $f(\vec{x}) =$ number of clauses which are `false`.
  - $f(\vec{x}) = 0 \Longrightarrow$ all clauses are `true`, SAT problem solved
- Candidate solution: string of n bits.

Tabu Search for MAX-SAT

- Let us consider a Tabu Search method for the MAX-SAT problem.

- Let us consider a Tabu Search method for the MAX-SAT problem.
- Neighborhood of candidate solution x: other bit strings assignments which differ in exactly one bit

- Let us consider a Tabu Search method for the MAX-SAT problem.
- Neighborhood of candidate solution x: other bit strings assignments which differ in exactly one bit
- Tabu feature: variables

- Let us consider a Tabu Search method for the MAX-SAT problem.
- Neighborhood of candidate solution x: other bit strings assignments which differ in exactly one bit
- Tabu feature: variables
- Tabu criterion: flipping the same variable again is forbidden for tt iterations

Tabu Search for MAX-SAT

- Let us consider a Tabu Search method for the MAX-SAT problem.
- Neighborhood of candidate solution x: other bit strings assignments which differ in exactly one bit
- Tabu feature: variables
- Tabu criterion: flipping the same variable again is forbidden for tt iterations
- Aspiration criterion: if flipping the variable would lead to a new best-so-far solution, we will accept it even if it is tabu

Section Outline

1 Introduction

2 Tabu Search

3 Example 1: MAX-SAT

4 Example 2: Traveling Salesman Problem

5 Iterated Local Search

6 Summary

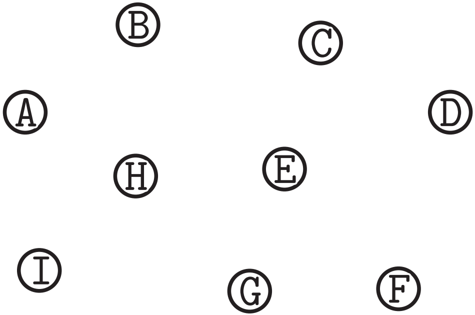
- Example: Traveling Salesman Problem (TSP)

- Example: Traveling Salesman Problem (TSP): Find a cyclic path of minimal costs that visits a set of cities V [5–8]

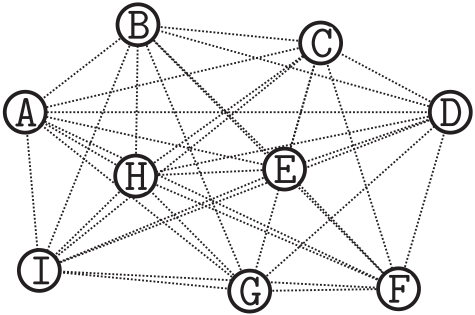
Example: Traveling Salesman Problem

- Example: Traveling Salesman Problem (TSP): Find a cyclic path of minimal costs that visits a set of cities V [5–8]
- Problem instance defined as:
  - set V of n_v nodes $v \in V$

- Example: Traveling Salesman Problem (TSP): Find a cyclic path of minimal costs that visits a set of cities V [5–8]
- Problem instance defined as:
  - set V of n_v nodes $v \in V$, e.g., $V = \{\mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{D}, \mathtt{E}, \mathtt{F}, \mathtt{G}, \mathtt{H}, \mathtt{I}\}$



- Example: Traveling Salesman Problem (TSP): Find a cyclic path of minimal costs that visits a set of cities V [5–8]
- Problem instance defined as:
  - set V of n_v nodes $v \in V$,
  - set $E = V \times V$ of edges $e = \overline{v_i\, v_j}$



Example: Traveling Salesman Problem

- Example: Traveling Salesman Problem (TSP): Find a cyclic path of minimal costs that visits a set of cities V [5–8]
- Problem instance defined as:
  - set V of n_v nodes $v \in V$,
  - set $E = V \times V$ of edges $e = \overline{v_i\, v_j}$, and
  - `cost` function to compute the cost of traveling along an edge $e \in E$

Example: Traveling Salesman Problem

- Example: Traveling Salesman Problem (TSP): Find a cyclic path of minimal costs that visits a set of cities V [5–8]
- Symmetric problem instance defined as:
  - set V of n_v nodes $v \in V$,
  - set $E = V \times V$ of edges $e = \overline{v_i\, v_j}$, and
  - `cost` function to compute the cost of traveling along an edge $e \in E$

Example: Traveling Salesman Problem

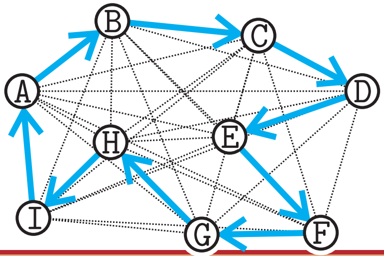
- Example: Traveling Salesman Problem (TSP): Find a cyclic path of minimal costs that visits a set of cities V [5–8]
- Symmetric problem instance defined as:
  - set V of n_v nodes $v \in V$,
  - set $E = V \times V$ of undirected edges $e = \overline{v_i\,v_j}$, and
  - `cost` function to compute the cost of traveling along an edge $e \in E$

Example: Traveling Salesman Problem

- Example: Traveling Salesman Problem (TSP): Find a cyclic path of minimal costs that visits a set of cities V [5–8]
- Symmetric problem instance defined as:
  - set V of n_v nodes $v \in V$,
  - set $E = V \times V$ of undirected edges $e = \overline{v_i\, v_j}$, and
  - `cost` function to compute the cost of traveling along an edge $e \in E$ (with $\texttt{cost}(\overline{\texttt{A}\,\texttt{B}}) = \texttt{cost}(\overline{\texttt{B}\,\texttt{A}})$)

- Example: Traveling Salesman Problem (TSP): Find a cyclic path of minimal costs that visits a set of cities V [5–8]
- Symmetric problem instance defined as:
  - set V of n_v nodes $v \in V$,
  - set $E = V \times V$ of undirected edges $e = \overline{v_i\, v_j}$, and
  - `cost` function to compute the cost of traveling along an edge $e \in E$
- Candidate solutions $\mathbf{x} \in \mathbb{X}$: permutations of the n_v nodes

- Example: Traveling Salesman Problem (TSP): Find a cyclic path of minimal costs that visits a set of cities V [5–8]
- Symmetric problem instance defined as:
  - set V of n_v nodes $v \in V$,
  - set $E = V \times V$ of undirected edges $e = \overline{v_i\, v_j}$, and
  - `cost` function to compute the cost of traveling along an edge $e \in E$
- Candidate solutions $\mathbf{x} \in \mathbb{X}$: permutations of the n_v nodes, e.g., $\mathbf{x} = (\mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{D}, \mathtt{E}, \mathtt{F}, \mathtt{G}, \mathtt{H}, \mathtt{I})$



Example: Traveling Salesman Problem

- Example: Traveling Salesman Problem (TSP): Find a cyclic path of minimal costs that visits a set of cities V [5–8]
- Symmetric problem instance defined as:
  - set V of n_v nodes $v \in V$,
  - set $E = V \times V$ of undirected edges $e = \overline{v_i\, v_j}$, and
  - `cost` function to compute the cost of traveling along an edge $e \in E$
- Candidate solutions $\mathbf{x} \in \mathbb{X}$: permutations of the n_v nodes
- Objective function f is the total tour cost

Example: Traveling Salesman Problem

- Example: Traveling Salesman Problem (TSP): Find a cyclic path of minimal costs that visits a set of cities V [5–8]
- Symmetric problem instance defined as:
  - set V of n_v nodes $v \in V$,
  - set $E = V \times V$ of undirected edges $e = \overline{v_i\, v_j}$, and
  - `cost` function to compute the cost of traveling along an edge $e \in E$
- Candidate solutions $\mathbf{x} \in \mathbb{X}$: permutations of the n_v nodes
- Objective function f is the total tour cost:

$$f(\mathbf{x}) = \sum_{i=1}^{n_v - 1} \texttt{cost}(\overline{\mathbf{x}_i\, \mathbf{x}_{i+1}}) \tag{1}$$

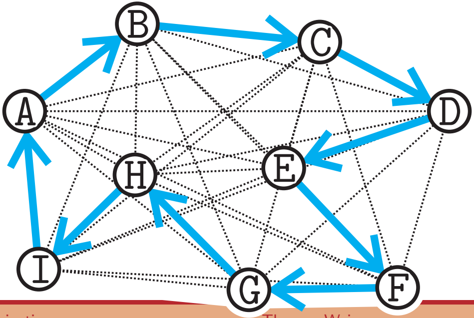
- Example: Traveling Salesman Problem (TSP): Find a cyclic path of minimal costs that visits a set of cities V [5–8]
- Symmetric problem instance defined as:
  - set V of n_v nodes $v \in V$,
  - set $E = V \times V$ of undirected edges $e = \overline{v_i\, v_j}$, and
  - `cost` function to compute the cost of traveling along an edge $e \in E$
- Candidate solutions $\mathbf{x} \in \mathbb{X}$: permutations of the n_v nodes
- Objective function f is the total tour cost:

$$f(\mathbf{x}) = \sum_{i=1}^{n_v-1} \mathtt{cost}(\overline{\mathbf{x}_i\, \mathbf{x}_{i+1}}) + \mathtt{cost}(\overline{\mathbf{x}_{n_v}\, \mathbf{x}_1}) \tag{1}$$

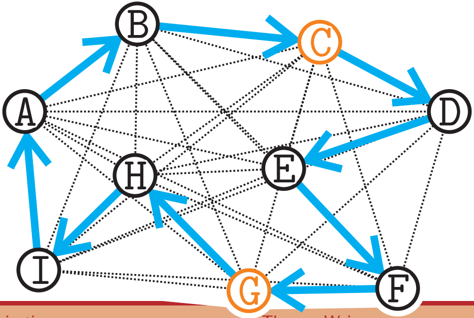
Neighborhood 1: Swap Operator

- $\texttt{swap}(\mathbf{x}, i, j)$: swap the element at index i in permutation $\mathbf{x}$ with element at index j [9–14]

- $\texttt{swap}(\mathbf{x}, i, j)$: swap the element at index i in permutation $\mathbf{x}$ with element at index j [9–14]
- $\mathbf{x} = (\texttt{A},\texttt{B},\texttt{C},\texttt{D},\texttt{E},\texttt{F},\texttt{G},\texttt{H},\texttt{I})$

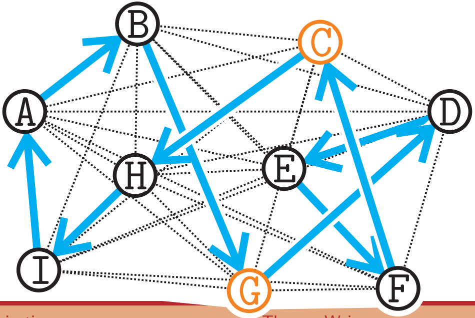


- swap($\mathbf{x}, i, j$): swap the element at index i in permutation $\mathbf{x}$ with element at index j [9–14]
- $\mathbf{x} = ($A,B,C,D,E,F,G,H,I$)$
- swap($\mathbf{x}, 3, 7$)

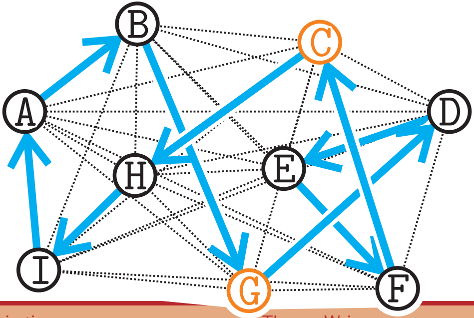


Neighborhood 1: Swap Operator

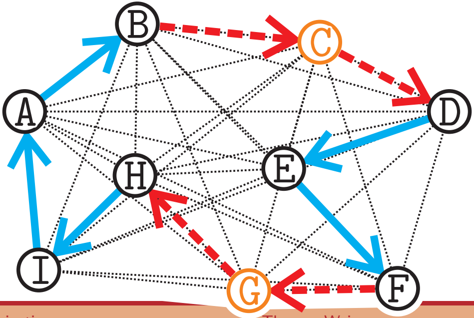
- swap($\mathbf{x}, i, j$): swap the element at index i in permutation $\mathbf{x}$ with element at index j [9–14]
- $\mathbf{x} = (\mathtt{A,B,C,D,E,F,G,H,I})$
- swap($\mathbf{x}, 3, 7$) $= (\mathtt{A,B,\underline{G},D,E,F,\underline{C},H,I})$



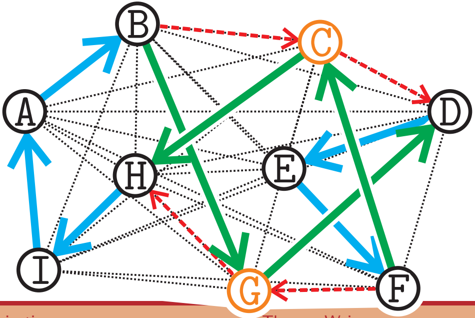
- swap$(\mathbf{x}, i, j)$: swap the element at index i in permutation $\mathbf{x}$ with element at index j [9–14]
- $\mathbf{x} = (\mathtt{A},\mathtt{B},\mathtt{C},\mathtt{D},\mathtt{E},\mathtt{F},\mathtt{G},\mathtt{H},\mathtt{I})$
- swap$(\mathbf{x}, 3, 7) = (\mathtt{A},\mathtt{B},\underline{\mathtt{G}},\mathtt{D},\mathtt{E},\mathtt{F},\underline{\mathtt{C}},\mathtt{H},\mathtt{I})$
- Possible 4-opt move



- $\mathtt{swap}(\mathbf{x}, i, j)$: swap the element at index i in permutation $\mathbf{x}$ with element at index j [9–14]
- $\mathbf{x} = (\mathtt{A},\mathtt{B},\mathtt{C},\mathtt{D},\mathtt{E},\mathtt{F},\mathtt{G},\mathtt{H},\mathtt{I})$
- $\mathtt{swap}(\mathbf{x}, 3, 7) = (\mathtt{A},\mathtt{B},\underline{\mathtt{G}},\mathtt{D},\mathtt{E},\mathtt{F},\underline{\mathtt{C}},\mathtt{H},\mathtt{I})$
- Possible 4-opt move: delete four edges

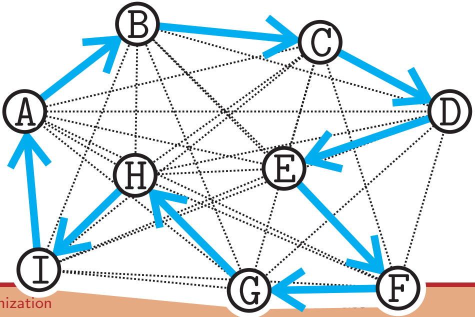


- $\mathtt{swap}(\mathbf{x}, i, j)$: swap the element at index i in permutation $\mathbf{x}$ with element at index j [9–14]
- $\mathbf{x} = (\mathtt{A,B,C,D,E,F,G,H,I})$
- $\mathtt{swap}(\mathbf{x}, 3, 7) = (\mathtt{A,B,\underline{G},D,E,F,\underline{C},H,I})$
- Possible 4-opt move: delete four edges and add four edges

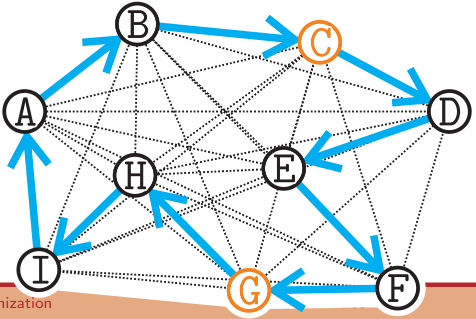


- `reverse`$(\mathbf{x}, i, j)$: reverse the subsequence between indexes i and j in permutation $\mathbf{x}$ [9, 15–19]

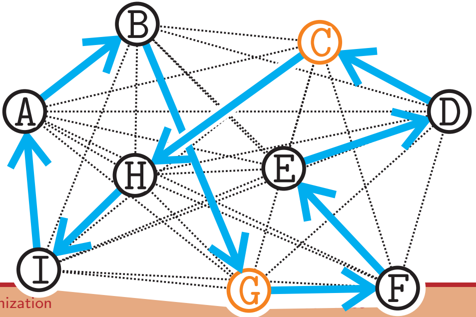
- $\texttt{reverse}(\mathbf{x}, i, j)$: reverse the subsequence between indexes i and j in permutation $\mathbf{x}$ [9, 15–19]
- $\mathbf{x} = (\texttt{A,B,C,D,E,F,G,H,I})$



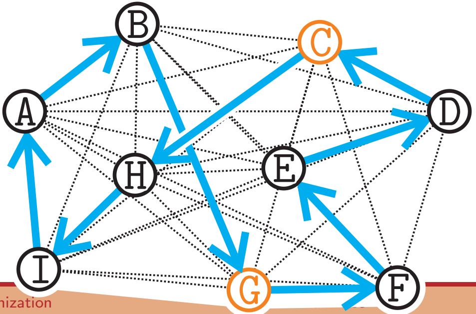
- $\texttt{reverse}(\mathbf{x}, i, j)$: reverse the subsequence between indexes i and j in permutation $\mathbf{x}$ [9, 15–19]
- $\mathbf{x} = (\texttt{A},\texttt{B},\texttt{C},\texttt{D},\texttt{E},\texttt{F},\texttt{G},\texttt{H},\texttt{I})$
- $\texttt{reverse}_1(\mathbf{x}, 3, 7)$



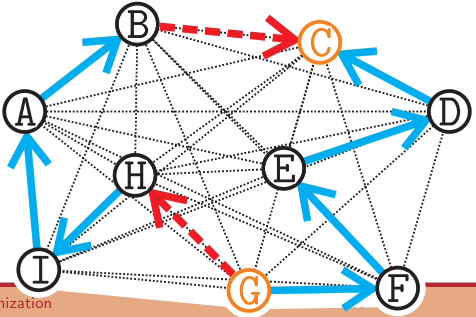
- `reverse`$(\mathbf{x}, i, j)$: reverse the subsequence between indexes i and j in permutation $\mathbf{x}$ [9, 15–19]
- $\mathbf{x} = (\texttt{A,B,C,D,E,F,G,H,I})$
- $\texttt{reverse}_1(\mathbf{x}, 3, 7) = (\texttt{A,B,}\underline{\texttt{G,F,E,D,C}}\texttt{,H,I})$



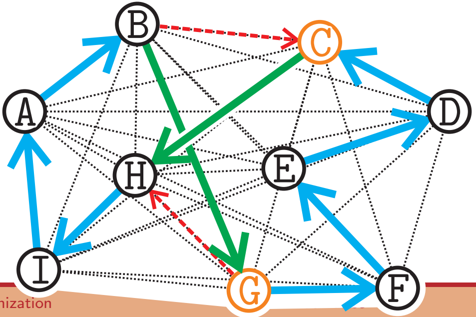
- $\texttt{reverse}(\mathbf{x}, i, j)$: reverse the subsequence between indexes i and j in permutation $\mathbf{x}$ [9, 15–19]
- $\mathbf{x} = (\texttt{A,B,C,D,E,F,G,H,I})$
- $\texttt{reverse}_1(\mathbf{x}, 3, 7) = (\texttt{A,B,}\underline{\texttt{G,F,E,D,C}}\texttt{,H,I})$
- Possible 2-opt move [19–21]



- $\texttt{reverse}(\mathbf{x}, i, j)$: reverse the subsequence between indexes i and j in permutation $\mathbf{x}$ [9, 15–19]
- $\mathbf{x} = (\texttt{A,B,C,D,E,F,G,H,I})$
- $\texttt{reverse}_1(\mathbf{x}, 3, 7) = (\texttt{A,B,}\underline{\texttt{G,F,E,D,C}}\texttt{,H,I})$
- Possible 2-opt move [19–21]: delete two edges



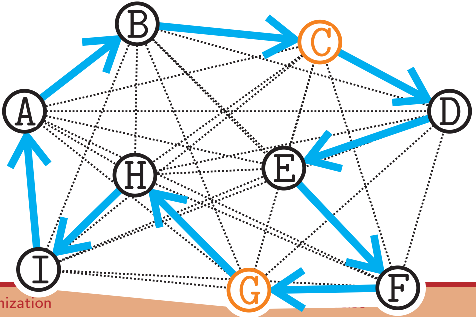
- $\texttt{reverse}(\mathbf{x}, i, j)$: reverse the subsequence between indexes i and j in permutation $\mathbf{x}$ [9, 15–19]
- $\mathbf{x} = (\texttt{A,B,C,D,E,F,G,H,I})$
- $\texttt{reverse}_1(\mathbf{x}, 3, 7) = (\texttt{A,B,}\underline{\texttt{G,F,E,D,C}}\texttt{,H,I})$
- Possible 2-opt move [19–21]: delete two edges and add two edges



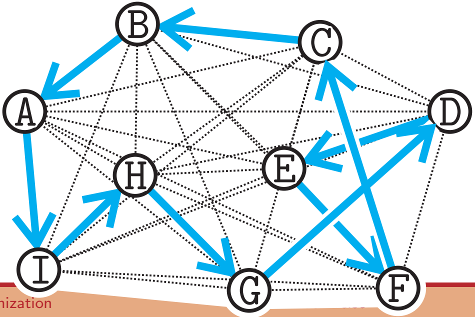
Neighborhood 2: Reverse Operator

- $\texttt{reverse}(\mathbf{x}, i, j)$: Two ways to reverse the subsequence between indexes i and j in permutation $\mathbf{x}$ [9, 15–19]
-

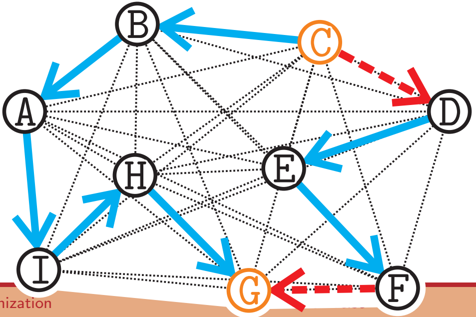
- $\texttt{reverse}(\mathbf{x}, i, j)$: Two ways to reverse the subsequence between indexes i and j in permutation $\mathbf{x}$ [9, 15–19]
- $\mathbf{x} = (\texttt{A,B,C,D,E,F,G,H,I})$
- $\texttt{reverse}_2(\mathbf{x}, 3, 7)$



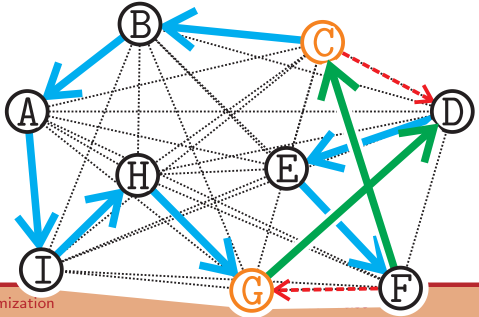
- reverse($\mathbf{x}, i, j$): Two ways to reverse the subsequence between indexes i and j in permutation $\mathbf{x}$ [9, 15–19]
- $\mathbf{x} = (\texttt{A},\texttt{B},\texttt{C},\texttt{D},\texttt{E},\texttt{F},\texttt{G},\texttt{H},\texttt{I})$
- $\texttt{reverse}_2(\mathbf{x}, 3, 7) = (\underline{\texttt{I},\texttt{H},\texttt{G}},\texttt{D},\texttt{E},\texttt{F},\underline{\texttt{C},\texttt{B},\texttt{A}})$



- $\texttt{reverse}(\mathbf{x}, i, j)$: Two ways to reverse the subsequence between indexes i and j in permutation $\mathbf{x}$ [9, 15–19]
- $\mathbf{x} = (\texttt{A,B,C,D,E,F,G,H,I})$
- $\texttt{reverse}_2(\mathbf{x}, 3, 7) = (\underline{\texttt{I,H,G}},\texttt{D,E,F},\underline{\texttt{C,B,A}})$
- Possible 2-opt move [19–21]: delete two edges and add two edges

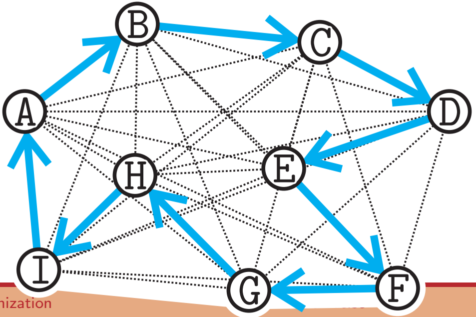


- `reverse`$(\mathbf{x}, i, j)$: Two ways to reverse the subsequence between indexes i and j in permutation $\mathbf{x}$ [9, 15–19]
- $\mathbf{x} = ($A,B,C,D,E,F,G,H,I$)$
- $\texttt{reverse}_2(\mathbf{x}, 3, 7) = (\underline{\texttt{I,H,G}},\texttt{D,E,F},\underline{\texttt{C,B,A}})$
- Possible 2-opt move [19–21]: delete two edges and add two edges

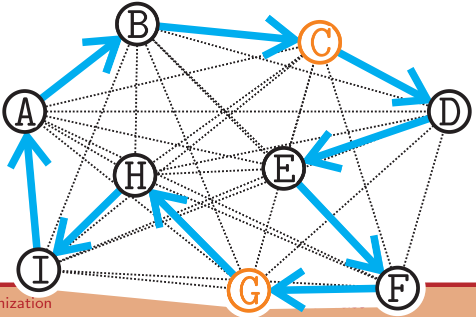


- `rotateLeft`$(\mathbf{x}, i, j)$: rotate the subsequence between indexes i and j in permutation $\mathbf{x}$ one step to the *left* [9, 11, 14, 22]

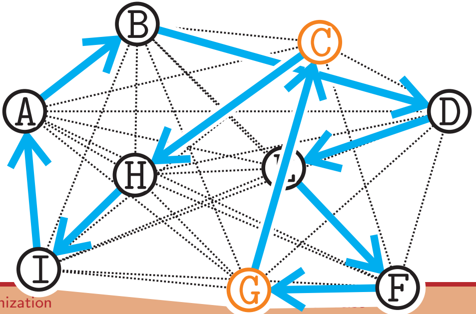
- `rotateLeft`$(\mathbf{x}, i, j)$: rotate the subsequence between indexes i and j in permutation $\mathbf{x}$ one step to the *left* [9, 11, 14, 22]
- `(A,B,C,D,E,F,G,H,I)`



- `rotateLeft`$(\mathbf{x}, i, j)$: rotate the subsequence between indexes i and j in permutation $\mathbf{x}$ one step to the *left* [9, 11, 14, 22]
- `(A,B,C,D,E,F,G,H,I)`
- `rotateLeft`$_1(\mathbf{x}, 3, 7)$

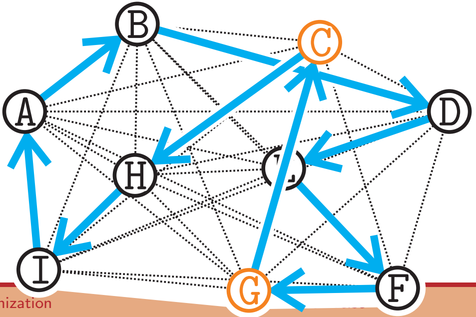


- `rotateLeft`$(\mathbf{x}, i, j)$: rotate the subsequence between indexes i and j in permutation $\mathbf{x}$ one step to the *left* [9, 11, 14, 22]
- `(A,B,C,D,E,F,G,H,I)`
- $\texttt{rotateLeft}_1(\mathbf{x}, 3, 7) = (\texttt{A,B,}\underline{\texttt{D,E,F,G,C}}\texttt{,H,I})$

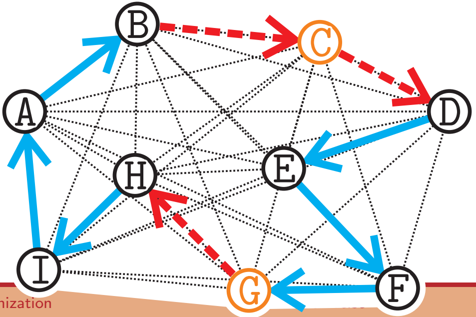


Neighborhood 3: Rotate-Left Operator

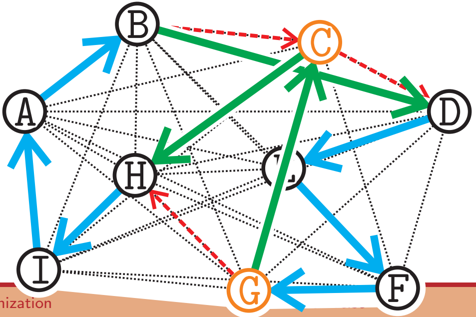
- $\texttt{rotateLeft}(\mathbf{x}, i, j)$: rotate the subsequence between indexes i and j in permutation $\mathbf{x}$ one step to the *left* [9, 11, 14, 22]
- $(\texttt{A,B,C,D,E,F,G,H,I})$
- $\texttt{rotateLeft}_1(\mathbf{x}, 3, 7) = (\texttt{A,B,}\underline{\texttt{D,E,F,G,C}}\texttt{,H,I})$
- Possible 3-opt move



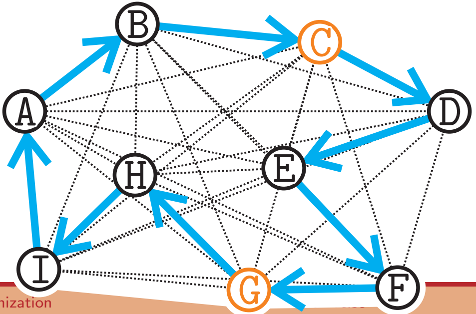
- $\texttt{rotateLeft}(\mathbf{x}, i, j)$: rotate the subsequence between indexes i and j in permutation $\mathbf{x}$ one step to the *left* [9, 11, 14, 22]
- $(\texttt{A,B,C,D,E,F,G,H,I})$
- $\texttt{rotateLeft}_1(\mathbf{x}, 3, 7) = (\texttt{A,B,}\underline{\texttt{D,E,F,G,C}}\texttt{,H,I})$
- Possible 3-opt move: delete three edges



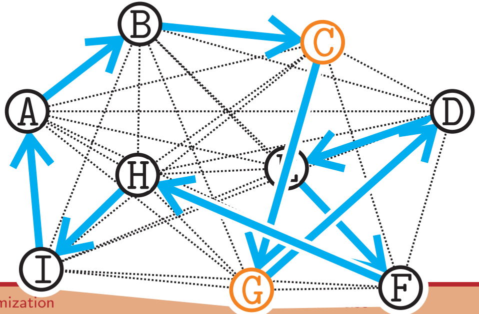
- $\texttt{rotateLeft}(\mathbf{x}, i, j)$: rotate the subsequence between indexes i and j in permutation $\mathbf{x}$ one step to the *left* [9, 11, 14, 22]
- (A,B,C,D,E,F,G,H,I)
- $\texttt{rotateLeft}_1(\mathbf{x}, 3, 7) =$ (A,B,D,E,F,G,C,H,I)
- Possible 3-opt move: delete three edges and add three edges



- `rotateLeft`$(\mathbf{x}, i, j)$: Two ways to rotate the subsequence between indexes i and j in permutation $\mathbf{x}$ one step to the *left* [9, 11, 14, 22]
-



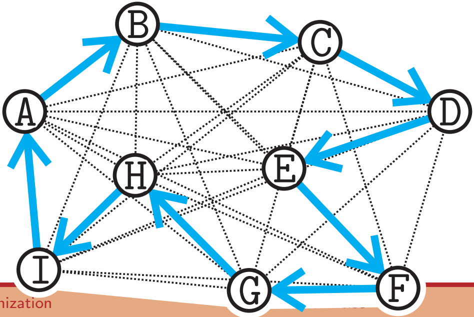
- `rotateLeft`($\mathbf{x}, i, j$): Two ways to rotate the subsequence between indexes i and j in permutation $\mathbf{x}$ one step to the *left* [9, 11, 14, 22]
- $\mathbf{x} = (\mathtt{A,B,C,D,E,F,G,H,I})$
- $\mathtt{rotateLeft}_2(\mathbf{x}, 3, 7) = (\underline{\mathtt{B,C,G}},\mathtt{D,E,F},\underline{\mathtt{H,I,A}})$



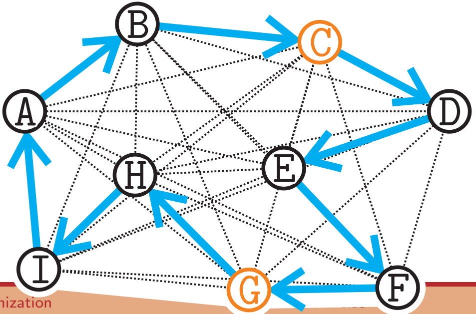
Neighborhood 4: Rotate-Right Operator

- $\texttt{rotateRight}(\mathbf{x}, i, j)$: rotate the subsequence between indexes i and j in permutation $\mathbf{x}$ one step to the *right* [9, 11, 14, 22]

- `rotateRight`$(\mathbf{x}, i, j)$: rotate the subsequence between indexes i and j in permutation $\mathbf{x}$ one step to the *right* [9, 11, 14, 22]
- (A,B,C,D,E,F,G,H,I)

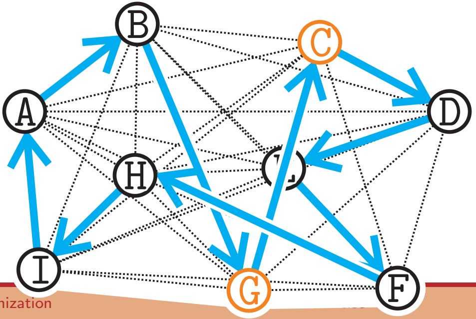


- `rotateRight`$(\mathbf{x}, i, j)$: rotate the subsequence between indexes i and j in permutation $\mathbf{x}$ one step to the *right* [9, 11, 14, 22]
- (A,B,C,D,E,F,G,H,I)
- $\texttt{rotateRight}_1(\mathbf{x}, 3, 7)$



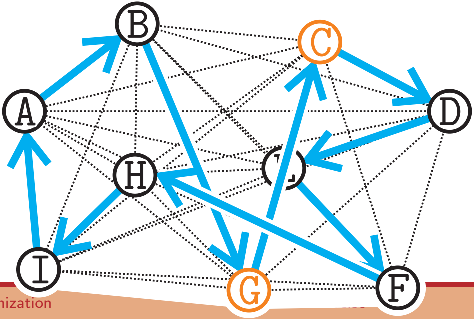
Neighborhood 4: Rotate-Right Operator

- `rotateRight`$(\mathbf{x}, i, j)$: rotate the subsequence between indexes i and j in permutation $\mathbf{x}$ one step to the *right* [9, 11, 14, 22]
- $(\texttt{A,B,C,D,E,F,G,H,I})$
- $\texttt{rotateRight}_1(\mathbf{x}, 3, 7) = (\texttt{A,B,}\underline{\texttt{G,C,D,E,F}}\texttt{,H,I})$

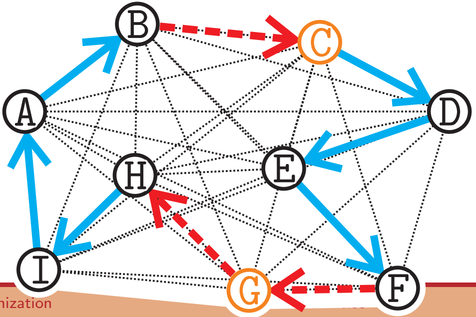


Neighborhood 4: Rotate-Right Operator

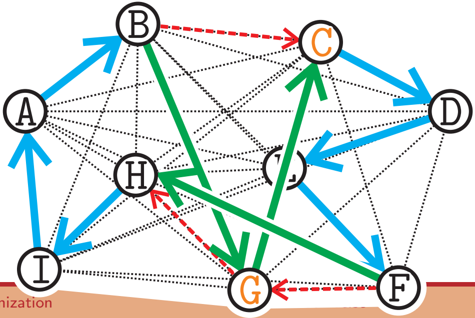
- `rotateRight`$(\mathbf{x}, i, j)$: rotate the subsequence between indexes i and j in permutation $\mathbf{x}$ one step to the *right* [9, 11, 14, 22]
- $(\mathtt{A,B,C,D,E,F,G,H,I})$
- $\mathtt{rotateRight}_1(\mathbf{x}, 3, 7) = (\mathtt{A,B,}\underline{\mathtt{G,C,D,E,F}}\mathtt{,H,I})$
- Possible 3-opt move



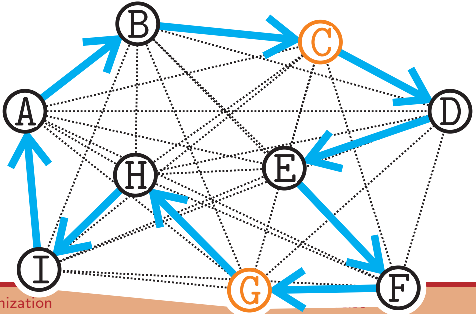
- `rotateRight`$(\mathbf{x}, i, j)$: rotate the subsequence between indexes i and j in permutation $\mathbf{x}$ one step to the *right* [9, 11, 14, 22]
- `(A,B,C,D,E,F,G,H,I)`
- `rotateRight`$_1(\mathbf{x}, 3, 7) =$ `(A,B,`$\underline{\texttt{G,C,D,E,F}}$`,H,I)`
- Possible 3-opt move: delete three edges



- `rotateRight`$(\mathbf{x}, i, j)$: rotate the subsequence between indexes i and j in permutation $\mathbf{x}$ one step to the *right* [9, 11, 14, 22]
- (A,B,C,D,E,F,G,H,I)
- `rotateRight`$_1(\mathbf{x}, 3, 7) =$ (A,B,$\underline{\texttt{G,C,D,E,F}}$,H,I)
- Possible 3-opt move: delete three edges and add three edges

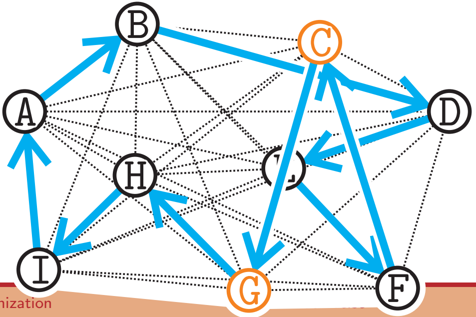


- `rotateRight`$(\mathbf{x}, i, j)$: Two ways to rotate the subsequence between indexes i and j in permutation $\mathbf{x}$ one step to the *right* [9, 11, 14, 22]
-

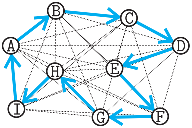


Neighborhood 4: Rotate-Right Operator

- `rotateRight`$(\mathbf{x}, i, j)$: Two ways to rotate the subsequence between indexes i and j in permutation $\mathbf{x}$ one step to the *right* [9, 11, 14, 22]
- $\mathbf{x} = (\texttt{A,B,C,D,E,F,G,H,I})$
- $\texttt{rotateRight}_2(\mathbf{x}, 3, 7) = (\underline{\texttt{I,A,B}},\texttt{D,E,F},\underline{\texttt{C,G,H}})$

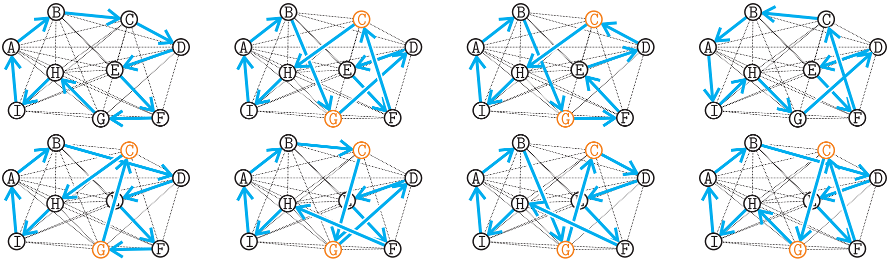


- For a candidate solution $\mathbf{x}$

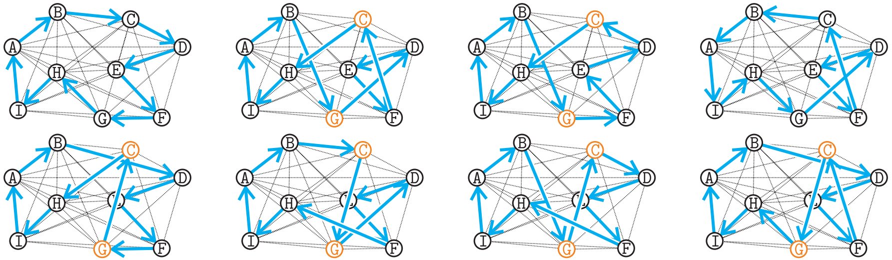


Neighborhoods Summary

- For a candidate solution $\mathbf{x}$ and an index tuple (i, j), we have learned that there are seven modification operations

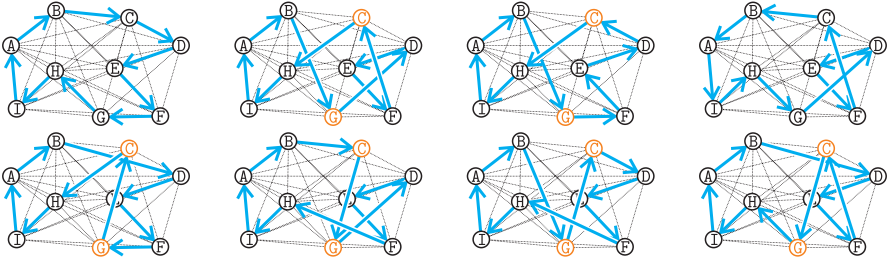


- For a candidate solution $\mathbf{x}$ and an index tuple (i, j), we have learned that there are seven modification operations
- We always can compute $f(\mathbf{x}')$ in $\mathcal{O}(1)$



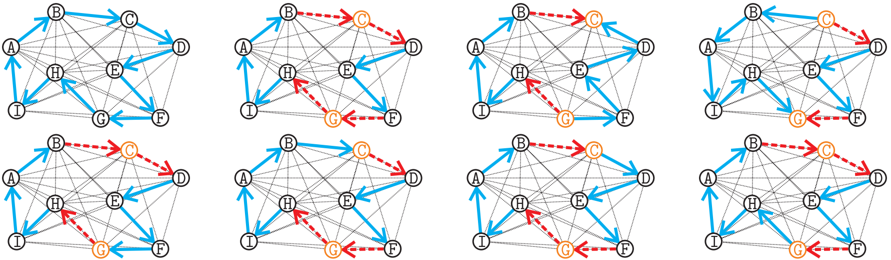
- For a candidate solution $\mathbf{x}$ and an index tuple (i, j), we have learned that there are seven modification operations
- We always can compute $f(\mathbf{x}')$ in $\mathcal{O}(1)$:

$$\Delta f = f(\mathbf{x}') - f(\mathbf{x})$$



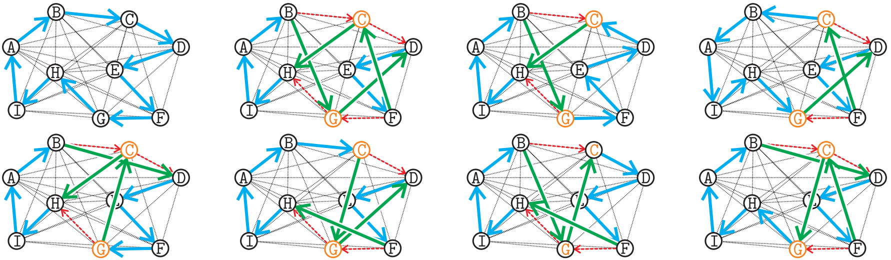
- For a candidate solution $\mathbf{x}$ and an index tuple (i, j), we have learned that there are seven modification operations
- We always can compute $f(\mathbf{x}')$ in $\mathcal{O}(1)$:

$$\Delta f = f(\mathbf{x}') - f(\mathbf{x}) = -\texttt{cost}(\text{deleted edges})$$



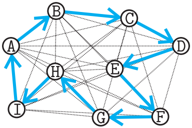
- For a candidate solution $\mathbf{x}$ and an index tuple (i, j), we have learned that there are seven modification operations
- We always can compute $f(\mathbf{x}')$ in $\mathcal{O}(1)$:

$$\Delta f = f(\mathbf{x}') - f(\mathbf{x}) = -\texttt{cost}(\text{deleted edges}) + \texttt{cost}(\text{added edges})$$



Neighborhoods Summary

- For a candidate solution $\mathbf{x}$ and an index tuple (i, j), we have learned that there are seven modification operations
- We always can compute $f(\mathbf{x}')$ in $\mathcal{O}(1)$:
- So if we choose one of these neighborhoods for our Tabu Search, we can scan the neighborhood of a solution by testing all indices i, j and for each neighbor (which is in $\mathcal{O}\left(n_v^2\right)$), we get the corresponding tour length/objective value basically for free...



- Let us consider a Tabu Search method for the Traveling Salesman Problem.

Tabu Search for the TSP

- Let us consider a Tabu Search method for the Traveling Salesman Problem.
- We can choose one of the four discussed neighborhoods, or even multiple neighborhoods.

- Let us consider a Tabu Search method for the Traveling Salesman Problem.
- We can choose one of the four discussed neighborhoods, or even multiple neighborhoods.
- Idea 1

- Let us consider a Tabu Search method for the Traveling Salesman Problem.
- We can choose one of the four discussed neighborhoods, or even multiple neighborhoods.
- Idea 1:
  - Tabu feature: edges

- Let us consider a Tabu Search method for the Traveling Salesman Problem.
- We can choose one of the four discussed neighborhoods, or even multiple neighborhoods.
- Idea 1:
    - Tabu feature: edges
    - Tabu criterion: adding a removed edge is not allowed.

- Let us consider a Tabu Search method for the Traveling Salesman Problem.
- We can choose one of the four discussed neighborhoods, or even multiple neighborhoods.
- Idea 1:
    - Tabu feature: edges
    - Tabu criterion: adding a removed edge is not allowed.
- Idea 2

- Let us consider a Tabu Search method for the Traveling Salesman Problem.
- We can choose one of the four discussed neighborhoods, or even multiple neighborhoods.
- Idea 1:
  - Tabu feature: edges
  - Tabu criterion: adding a removed edge is not allowed.
- Idea 2:
  - Tabu feature: indexes i, j

Tabu Search for the TSP

- Let us consider a Tabu Search method for the Traveling Salesman Problem.
- We can choose one of the four discussed neighborhoods, or even multiple neighborhoods.
- Idea 1:
  - Tabu feature: edges
  - Tabu criterion: adding a removed edge is not allowed.
- Idea 2:
  - Tabu feature: indexes i, j
  - Tabu criterion: using the same indexes i and j again is not allowed

- Let us consider a Tabu Search method for the Traveling Salesman Problem.
- We can choose one of the four discussed neighborhoods, or even multiple neighborhoods.
- Idea 1:
  - Tabu feature: edges
  - Tabu criterion: adding a removed edge is not allowed.
- Idea 2:
  - Tabu feature: indexes i, j
  - Tabu criterion: using the same indexes i and j again is not allowed
- Idea 3

Tabu Search for the TSP

- Let us consider a Tabu Search method for the Traveling Salesman Problem.
- We can choose one of the four discussed neighborhoods, or even multiple neighborhoods.
- Idea 1:
  - Tabu feature: edges
  - Tabu criterion: adding a removed edge is not allowed.
- Idea 2:
  - Tabu feature: indexes i, j
  - Tabu criterion: using the same indexes i and j again is not allowed
- Idea 3:
  - Tabu feature: objective value

Tabu Search for the TSP

- Let us consider a Tabu Search method for the Traveling Salesman Problem.
- We can choose one of the four discussed neighborhoods, or even multiple neighborhoods.
- Idea 1:
  - Tabu feature: edges
  - Tabu criterion: adding a removed edge is not allowed.
- Idea 2:
  - Tabu feature: indexes i, j
  - Tabu criterion: using the same indexes i and j again is not allowed
- Idea 3:
  - Tabu feature: objective value
  - Tabu criterion: creating a tour with the same length as a previously visited one is not allowed

Tabu Search for the TSP

- Let us consider a Tabu Search method for the Traveling Salesman Problem.
- We can choose one of the four discussed neighborhoods, or even multiple neighborhoods.
- Idea 1:
  - Tabu feature: edges
  - Tabu criterion: adding a removed edge is not allowed.
- Idea 2:
  - Tabu feature: indexes i, j
  - Tabu criterion: using the same indexes i and j again is not allowed
- Idea 3:
  - Tabu feature: objective value
  - Tabu criterion: creating a tour with the same length as a previously visited one is not allowed
- Many ideas are possible...

Tabu Search for the TSP

- Let us consider a Tabu Search method for the Traveling Salesman Problem.
- We can choose one of the four discussed neighborhoods, or even multiple neighborhoods.
- Idea 1:
  - Tabu feature: edges
  - Tabu criterion: adding a removed edge is not allowed.
- Idea 2:
  - Tabu feature: indexes i, j
  - Tabu criterion: using the same indexes i and j again is not allowed
- Idea 3:
  - Tabu feature: objective value
  - Tabu criterion: creating a tour with the same length as a previously visited one is not allowed
- Many ideas are possible. . .
- Aspiration criterion: if the new tour would be a new best-so-far solution, we will accept it even if it is tabu

1 Introduction

2 Tabu Search

3 Example 1: MAX-SAT

4 Example 2: Traveling Salesman Problem

5 Iterated Local Search

6 Summary

Iterated Local Search

- We have seen that it might happen that the Tabu Search finds no acceptable solution to go to anymore.

Iterated Local Search

- We have seen that it might happen that the Tabu Search finds no acceptable solution to go to anymore.
- What can we do in this case?

Iterated Local Search

- We have seen that it might happen that the Tabu Search finds no acceptable solution to go to anymore.
- What can we do in this case?:
  - Restart the algorithm at a new (random?) solution p_{cur}

- We have seen that it might happen that the Tabu Search finds no acceptable solution to go to anymore.
- What can we do in this case?:
    - Restart the algorithm at a new (random?) solution p_{cur} (while remembering p_{best}, oft course)

- We have seen that it might happen that the Tabu Search finds no acceptable solution to go to anymore.
- What can we do in this case?:
  - Restart the algorithm at a new (random?) solution p_{cur} (while remembering p_{best}, oft course) $\Longrightarrow$ This is very harsh, as we throw away a potentially good solution structure.

- We have seen that it might happen that the Tabu Search finds no acceptable solution to go to anymore.
- What can we do in this case?:
  - Restart the algorithm at a new (random?) solution p_{cur} (while remembering p_{best}, oft course) $\Longrightarrow$ This is very harsh, as we throw away a potentially good solution structure.
  - Soft restart: Apply a modification to p_{cur} which the current search moves cannot do, i.e., move outside of the current neighborhood of p_{cur} without throwing it away completely.

Iterated Local Search

- We have seen that it might happen that the Tabu Search finds no acceptable solution to go to anymore.
- What can we do in this case?:
  - Restart the algorithm at a new (random?) solution p_{cur} (while remembering p_{best}, oft course) $\Longrightarrow$ This is very harsh, as we throw away a potentially good solution structure.
  - Soft restart: Apply a modification to p_{cur} which the current search moves cannot do, i.e., move outside of the current neighborhood of p_{cur} without throwing it away completely.
- This is a common method for local search, not just for Tabu Search.

- We have seen that it might happen that the Tabu Search finds no acceptable solution to go to anymore.
- What can we do in this case?:
  - Restart the algorithm at a new (random?) solution p_{cur} (while remembering p_{best}, oft course) $\Longrightarrow$ This is very harsh, as we throw away a potentially good solution structure.
  - Soft restart: Apply a modification to p_{cur} which the current search moves cannot do, i.e., move outside of the current neighborhood of p_{cur} without throwing it away completely.
- This is a common method for local search, not just for Tabu Search.
- It can also be applied to Simulated Annealing, or to Hill Climbers if, e.g., they do not find improvements for several steps.

Iterated Local Search

- We have seen that it might happen that the Tabu Search finds no acceptable solution to go to anymore.
- What can we do in this case?:
    - Restart the algorithm at a new (random?) solution p_{cur} (while remembering p_{best}, oft course) $\Longrightarrow$ This is very harsh, as we throw away a potentially good solution structure.
    - Soft restart: Apply a modification to p_{cur} which the current search moves cannot do, i.e., move outside of the current neighborhood of p_{cur} without throwing it away completely.
- This is a common method for local search, not just for Tabu Search.
- It can also be applied to Simulated Annealing, or to Hill Climbers if, e.g., they do not find improvements for several steps.
- Such searches are called *Iterated Local Search* (ILS) [4]

- We have seen that it might happen that the Tabu Search finds no acceptable solution to go to anymore.
- What can we do in this case?:
  - Restart the algorithm at a new (random?) solution p_{cur} (while remembering p_{best}, oft course) $\Longrightarrow$ This is very harsh, as we throw away a potentially good solution structure.
  - Soft restart: Apply a modification to p_{cur} which the current search moves cannot do, i.e., move outside of the current neighborhood of p_{cur} without throwing it away completely.
- This is a common method for local search, not just for Tabu Search.
- It can also be applied to Simulated Annealing, or to Hill Climbers if, e.g., they do not find improvements for several steps.
- Such searches are called *Iterated Local Search* (ILS) [4]
- They have astonishingly great performance, and several of the best application-specific optimization methods are based on them [4, 23].

Section Outline

1. Introduction
2. Tabu Search
3. Example 1: MAX-SAT
4. Example 2: Traveling Salesman Problem
5. Iterated Local Search
6. Summary

Summary

- Tabu Search is another highly efficient local search.

Summary

- Tabu Search is another highly efficient local search.
- It is based on the concept of scanning complete neighborhoods and avoiding to move in a cycle by declaring certain solution features as "tabu".

Summary

- Tabu Search is another highly efficient local search.
- It is based on the concept of scanning complete neighborhoods and avoiding to move in a cycle by declaring certain solution features as "tabu".
- The concept of search moves here is slightly different from the algorithms we discussed before and will discuss afterwards, it is centered around neighborhoods rather than single modifications.

Summary

- Tabu Search is another highly efficient local search.
- It is based on the concept of scanning complete neighborhoods and avoiding to move in a cycle by declaring certain solution features as "tabu".
- The concept of search moves here is slightly different from the algorithms we discussed before and will discuss afterwards, it is centered around neighborhoods rather than single modifications.
- Tabu Search, Simulated Annealing, and many other local search algorithms can be *iterated* by making stronger search moves or restarting them altogether from time to time.

Summary

- Tabu Search is another highly efficient local search.
- It is based on the concept of scanning complete neighborhoods and avoiding to move in a cycle by declaring certain solution features as “tabu”.
- The concept of search moves here is slightly different from the algorithms we discussed before and will discuss afterwards, it is centered around neighborhoods rather than single modifications.
- Tabu Search, Simulated Annealing, and many other local search algorithms can be *iterated* by making stronger search moves or restarting them altogether from time to time.
- We have also looked into two very well-known, classical problems from operations research again, Maximum Satisfiability and the Traveling Salesman Problem.

谢谢

Thank you

Thomas Weise [汤卫思]
tweise@hfuu.edu.cn
http://iao.hfuu.edu.cn

Hefei University, South Campus 2
Institute of Applied Optimization
Shushan District, Hefei, Anhui,
China

Caspar David Friedrich, “Der Wanderer über dem Nebelmeer”, 1818
http://en.wikipedia.org/wiki/Wanderer_above_the_Sea_of_Fog

Bibliography

Bibliography I

1. Fred W. Glover. Tabu search – part i. *ORSA Journal on Computing*, 1(3):190–206, Summer 1989. doi: 10.1287/ijoc.1.3.190. URL http://leeds-faculty.colorado.edu/glover/TS%20-%20Part%20I-ORSA.pdf.
2. Fred W. Glover. Tabu search – part ii. *ORSA Journal on Computing*, 2(1):190–206, Winter 1990. doi: 10.1287/ijoc.2.1.4. URL http://leeds-faculty.colorado.edu/glover/TS%20-%20Part%20II-ORSA-aw.pdf.
3. Holger H. Hoos and Thomas Stützle. Satlib: An online resource for research on sat. In Ian Gent, Hans van Maaren, and Toby Walsh, editors, *SAT2000 – Highlights of Satisfiability Research in the Year 2000*, volume 63 of *Frontiers in Artificial Intelligence and Applications*, pages 283–292. Amsterdam, The Netherlands: IOS Press, 2000. URL http://www.cs.ubc.ca/~hoos/Publ/sat2000-satlib.pdf.
4. Holger H. Hoos and Thomas Stützle. *Stochastic Local Search: Foundations and Applications*. The Morgan Kaufmann Series in Artificial Intelligence. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005. ISBN 1558608729 and 978-1558608726. URL http://books.google.de/books?id=3HAedXnC49IC.
5. Bernhard Friedrich Voigt. *Der Handlungsreisende – wie er sein soll und was er zu thun hat, um Aufträge zu erhalten und eines glücklichen Erfolgs in seinen Geschäften gewiß zu sein – von einem alten Commis-Voyageur*. Ilmenau, Germany: Voigt, 1832. Excerpt: "... Durch geeignete Auswahl und Planung der Tour kann man oft so viel Zeit sparen, daß wir einige Vorschläge zu machen haben. ... Der wichtigste Aspekt ist, so viele Orte wie möglich zu erreichen, ohne einen Ort zweimal zu besuchen. ...".
6. David Lee Applegate, Robert E. Bixby, Vašek Chvátal, and William John Cook. *The Traveling Salesman Problem: A Computational Study*. Princeton Series in Applied Mathematics. Princeton, NJ, USA: Princeton University Press, February 2007. ISBN 0-691-12993-2 and 978-0-691-12993-8. URL http://books.google.de/books?id=nmF4rVNJMVsC.
7. Federico Greco, editor. *Traveling Salesman Problem*. Vienna, Austria: IN-TECH Education and Publishing, September 2008. ISBN 978-953-7619-10-7. URL http://intechweb.org/downloadfinal.php?is=978-953-7619-10-7&type=B.
8. Eugene Leighton (Gene) Lawler, Jan Karel Lenstra, Alexander Hendrik George Rinnooy Kan, and David B. Shmoys. *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*. Estimation, Simulation, and Control – Wiley-Interscience Series in Discrete Mathematics and Optimization. Chichester, West Sussex, UK: Wiley Interscience, September 1985. ISBN 0-471-90413-9 and 978-0-471-90413-7. URL http://books.google.de/books?id=BXBGAAAAYAAJ.
9. Pedro Larrañaga, Cindy M. H. Kuijpers, Roberto H. Murga, Iñaki Inza, and Sejla Dizdarevic. Genetic algorithms for the travelling salesman problem: A review of representations and operators. *Journal of Artificial Intelligence Research (JAIR)*, 13(2):129–170, April 1999. doi: 10.1023/A:1006529012972. URL http://www.dca.fee.unicamp.br/~gomide/courses/EA072/artigos/Genetic_Algorithm_TSPR_eview_Larranaga_1999.pdf.

Bibliography II

10. I. M. Oliver, D. J. Smith, and John Henry Holland. A study of permutation crossover operators on the traveling salesman problem. In John J. Grefenstette, editor, *Proceedings of the Second International Conference on Genetic Algorithms and their Applications (ICGA'87)*, pages 224–230, Cambridge, MA, USA: Massachusetts Institute of Technology (MIT), July 28–31, 1987. Mahwah, NJ, USA: Lawrence Erlbaum Associates, Inc. (LEA).
11. Zbigniew Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. Berlin, Germany: Springer-Verlag GmbH, 1996. ISBN 3-540-58090-5, 3-540-60676-9, 978-3-540-60676-5, and 978-3-642-08233-7. URL `http://books.google.de/books?id=vlhLAobsK68C`.
12. Wolfgang Banzhaf. The "molecular" traveling salesman. *Biological Cybernetics*, 64(1):7–14, November 1990. doi: 10.1007/BF00203625. URL `https://web.cs.mun.ca/~banzhaf/papers/MolTravelSalesman.pdf`.
13. Balamurali Krishna Ambati, Jayakrishna Ambati, and Mazen Moein Mokhtar. Heuristic combinatorial optimization by simulated darwinian evolution: A polynomial time algorithm for the traveling salesman problem. *Biological Cybernetics*, 65 (1):31–35, May 1991. doi: 10.1007/BF00197287.
14. Gilbert Syswerda. Schedule optimization using genetic algorithms. In Lawrence Davis, editor, *Handbook of Genetic Algorithms*, VNR Computer Library, pages 332–349. Stamford, CT, USA: Thomson Publishing Group, Inc. and New York, NY, USA: Van Nostrand Reinhold Co., January 1991.
15. John Henry Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. Ann Arbor, MI, USA: University of Michigan Press, 1975. ISBN 0-472-08460-7 and 978-0-472-08460-9. URL `http://books.google.de/books?id=JE5RAAAAMAAJ`.
16. John J. Grefenstette. Incorporating problem specific knowledge into genetic algorithms. In Lawrence Davis, editor, *Genetic Algorithms and Simulated Annealing*, Research Notes in Artificial Intelligence, pages 42–60. London, UK: Pitman, 1987.
17. David Stifler Johnson, Cecilia R. Aragon, Lyle A. McGeoch, and Catherine A. Schevon. Optimization by simulated annealing: An experimental evaluation. part i, graph partitioning. *Operations Research (Oper. Res.)*, 37(6), November–December 1989. doi: 10.1287/opre.37.6.865.
18. Scott Kirkpatrick, Charles Daniel Gelatt, Jr., and Mario P. Vecchi. Optimization by simulated annealing. *Science Magazine*, 220(4598):671–680, May 13, 1983. doi: 10.1126/science.220.4598.671. URL `http://fezzik.ucd.ie/msc/cscs/ga/kirkpatrick83optimization.pdf`.
19. David Lee Applegate, Robert E. Bixby, Vašek Chvátal, and William John Cook. Finding tours in the tsp: Finding tours. Sonderforschungsbereich 303: Sonderforschungsbereich Information und die Koordination Wirtschaftlicher Aktivitäten – Report 99885, Bonn, North Rhine-Westphalia, Germany: Rheinische Friedrich-Wilhelms-Universität Bonn, 1999. URL `http://www.tsp.gatech.edu/methods/papers/lk_report.ps`.

Bibliography III

20. G. A. Croes. A method for solving traveling-salesman problems. *Operations Research (Oper. Res.)*, 6(6):791–812, November–December 1958. doi: 10.1287/opre.6.6.791. URL http://www.jstor.org/stable/167074.
21. Merrill M. Flood. The traveling-salesman problem. *Operations Research (Oper. Res.)*, 4(1):61–75, February 1956. doi: 10.1287/opre.4.1.61.
22. David B. Fogel. An evolutionary approach to the traveling salesman problem. *Biological Cybernetics*, 60(2):139–144, December 1988. doi: 10.1007/BF00202901. URL http://users.on.net/~jivlain/papers/4%20Fogel.pdf.
23. David Lee Applegate, William John Cook, and André Rohe. Chained lin-kernighan for large traveling salesman problems. *INFORMS Journal on Computing (JOC)*, 15(1):82–92, Winter 2003. URL http://www2.isye.gatech.edu/~wcook/papers/clk_ijoc.pdf.