# Combinatorial Optimaziation: Branch and Bound Algorithm for the Travelling Salesman Problem

Generated by Doxygen 1.8.13

# Contents

# Chapter 1

# Namespace Index

## 1.1 Namespace List

Here is a list of all documented namespaces with brief descriptions:

# Chapter 2

# Class Index

## 2.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

# Chapter 3

# File Index

## 3.1 File List

Here is a list of all documented files with brief descriptions:

# Chapter 4

# Namespace Documentation

## 4.1 TSP Namespace Reference

### Classes

- class BranchingNode
- class Instance
- class Node

  *A `Node` stores an array of neighbors (via their ids).*
- class OneTree

### Typedefs

- using **size_type** = std::size_t
- using **NodeId** = size_type
- using **TSPlibId** = size_type
- using **EdgeId** = size_type

### Functions

- template<class coord_type , class dist_type >
  void compute_minimal_1_tree (TSP::OneTree &tree, const std::vector< double > &lambda, const TSP::↩
  Instance< coord_type, dist_type > &tsp, const TSP::BranchingNode< coord_type, dist_type > &BNode)
- template<class coord_type , class dist_type >
  dist_type Held_Karp (const TSP::Instance< coord_type, dist_type > &tsp, std::vector< double > &lambda,
  TSP::OneTree &tree, const TSP::BranchingNode< coord_type, dist_type > &bn, bool root=false)

#### 4.1.1 Detailed Description

defines elements, methods and classes within the Travelling Salesman Context

#### 4.1.2 Function Documentation

#### 4.1.2.1 compute_minimal_1_tree()

```
template<class coord_type , class dist_type >
void TSP::compute_minimal_1_tree (
              TSP::OneTree & tree,
              const std::vector< double > & lambda,
              const TSP::Instance< coord_type, dist_type > & tsp,
              const TSP::BranchingNode< coord_type, dist_type > & BNode )
```

computes a minimum-1-tree for a given BranchingNode

**Template Parameters**

| | |
|---|---|
| *coord_type* | |
| *dist_type* | |

**Parameters**

| | |
|---|---|
| *tree* | space to save the optimal tree |
| *lambda* | if we are in the root node, we'll save our holy lambda here, else it's just the root lambda |
| *tsp* | The TSP Instance |
| *BNode* | the correspnding BranchingNode |

#### 4.1.2.2 Held_Karp()

```
template<class coord_type , class dist_type >
dist_type TSP::Held_Karp (
              const TSP::Instance< coord_type, dist_type > & tsp,
              std::vector< double > & lambda,
              TSP::OneTree & tree,
              const TSP::BranchingNode< coord_type, dist_type > & bn,
              bool root = false )
```

computes the Held-Karp lower bound

**Template Parameters**

| | |
|---|---|
| *coord_type* | |
| *dist_type* | |

**Parameters**

| | |
|---|---|
| *tsp* | The TSP Instance |
| *lambda* | lambda set by root BranchingNode (or where to set for not BranchingNode) |
| *tree* | container for tree computation |
| *bn* | current BranchingNode |
| *root* | true, if we are in the root of our B'n'B tree |

**Returns**

# Chapter 5

# Class Documentation

## 5.1 TSP::BranchingNode< coord_type, dist_type > Class Template Reference

### Public Member Functions

- BranchingNode (const Instance< coord_type, dist_type > &tsp)
- BranchingNode (const BranchingNode< coord_type, dist_type > &BNode, const Instance< coord_type, dist_type > &tsp, EdgeId e1)
- BranchingNode (const BranchingNode< coord_type, dist_type > &BNode, const Instance< coord_type, dist_type > &tsp, EdgeId e1, EdgeId e2)
- BranchingNode (const BranchingNode< coord_type, dist_type > &BNode, const Instance< coord_type, dist_type > &tsp, EdgeId e1, EdgeId e2, bool both_req)
- bool operator> (const BranchingNode< coord_type, dist_type > &rhs) const
- EdgeId reverse_edge (EdgeId e, size_type n) const
- bool is_required (EdgeId id) const
- bool is_forbidden (EdgeId id) const
- void forbid (NodeId idx, EdgeId e1, EdgeId e2)
- void admit (NodeId idx)
- bool push_required (EdgeId e)
- void add_required (EdgeId e)
- bool push_forbidden (EdgeId e)
- void add_forbidden (EdgeId e)
- const std::vector< EdgeId > & **get_required** () const
- const std::vector< EdgeId > & **get_forbidden** () const
- const std::vector< dist_type > & **get_lambda** () const
- std::vector< dist_type > & **get_lambda** ()
- const OneTree & **get_tree** () const
- OneTree & **get_tree** ()
- const dist_type **get_HK** () const
- const std::vector< Node > & **get_required_neighbors** () const
- bool tworegular ()

### 5.1.1 Detailed Description

**template<class coord_type, class dist_type>**
**class TSP::BranchingNode< coord_type, dist_type >**

Node in our branch and bound tree. It contains the lower bound for itself and saves temporary information as tree, lambda, required, forbidden. Yeah, for the different branching nodes and their 'to append' edges, we implemented separate constructors. It could be achieved by one, but this was easier, as we are lacking some structures necessary for this. - Alex

**Template Parameters**

| | |
|---|---|
| *coord_type* | Container in which the Coordinates are given. Assumably double |
| *dist_type* | Container in which the distances are given. Assumably double |

### 5.1.2 Constructor & Destructor Documentation

#### 5.1.2.1 BranchingNode() [1/4]

```
template<class coord_type, class dist_type>
TSP::BranchingNode< coord_type, dist_type >::BranchingNode (
            const Instance< coord_type, dist_type > & tsp )  [inline]
```

First Constructor: Constructs a BranchingNode without any forbidden or required edges, i.e. the root of our B'n'B tree.

**Parameters**

| | |
|---|---|
| *tsp* | The TSP Instance |

#### 5.1.2.2 BranchingNode() [2/4]

```
template<class coord_type, class dist_type>
TSP::BranchingNode< coord_type, dist_type >::BranchingNode (
            const BranchingNode< coord_type, dist_type > & BNode,
            const Instance< coord_type, dist_type > & tsp,
            EdgeId e1 )  [inline]
```

Second Constructor: Constructs a BranchingNode with $F := F \cup e_1$

**Parameters**

| | |
|---|---|
| *BNode* | predecessor BranchingNode |
| *tsp* | The TSP Instance |
| *e1* | additional edge for forbidden edges |

#### 5.1.2.3 BranchingNode() [3/4]

```
template<class coord_type, class dist_type>
TSP::BranchingNode< coord_type, dist_type >::BranchingNode (
```

```
            const BranchingNode< coord_type, dist_type > & BNode,
            const Instance< coord_type, dist_type > & tsp,
            EdgeId e1,
            EdgeId e2 )  [inline]
```

Third Constructor: Constructs a BranchingNode with $R := R \cup e_1 F := F \cup e_2$

**Parameters**

| | |
|---|---|
| *BNode* | predecessor BranchingNode |
| *tsp* | The TSP Instance |
| *e1* | additional edge for required edges |
| *e2* | additional edge for forbidden edges |

#### 5.1.2.4 BranchingNode() [4/4]

```
template<class coord_type, class dist_type>
TSP::BranchingNode< coord_type, dist_type >::BranchingNode (
            const BranchingNode< coord_type, dist_type > & BNode,
            const Instance< coord_type, dist_type > & tsp,
            EdgeId e1,
            EdgeId e2,
            bool both_req )  [inline]
```

Fourth Constructor: Constructs a BranchingNode with $R := R \cup e_1 \cup e_2$

**Parameters**

| | |
|---|---|
| *BNode* | predecessor BranchingNode |
| *tsp* | The TSP Instance |
| *e1* | additional edge for required edges |
| *e2* | additional edge for forbidden edges |
| *both_req* | bool to distinguish third from forth constructor |

### 5.1.3 Member Function Documentation

#### 5.1.3.1 add_forbidden()

```
template<class coord_type , class dist_type >
void TSP::BranchingNode< coord_type, dist_type >::add_forbidden (
            EdgeId e )
```

external function to add an edge e to forbidden

**Parameters**

| | |
|---|---|
| *e* | |

#### 5.1.3.2 add_required()

```
template<class coord_type , class dist_type >
void TSP::BranchingNode< coord_type, dist_type >::add_required (
            EdgeId e )
```

external function to add an edge e to required

**Parameters**

| | |
|---|---|
| *e* | |

#### 5.1.3.3 admit()

```
template<class coord_type , class dist_type >
void TSP::BranchingNode< coord_type, dist_type >::admit (
            NodeId idx )
```

adds non-forbidden edges of incident edges of idx to required

**Parameters**

| | |
|---|---|
| *idx* | |

#### 5.1.3.4 forbid()

```
template<class coord_type , class dist_type >
void TSP::BranchingNode< coord_type, dist_type >::forbid (
            NodeId idx,
            EdgeId e1,
            EdgeId e2 )
```

forbids all edges incident to idx except e1,e2

**Parameters**

| | |
|---|---|
| *idx* | |
| *e1* | |
| *e2* | |

#### 5.1.3.5 is_forbidden()

```
template<class coord_type, class dist_type>
bool TSP::BranchingNode< coord_type, dist_type >::is_forbidden (
            EdgeId id ) const  [inline]
```

**Parameters**

| | |
|---|---|
| *id* | EdgeId id |

**Returns**

true, if the undirected edge {i,j} is forbidden, else false

#### 5.1.3.6 is_required()

```
template<class coord_type, class dist_type>
bool TSP::BranchingNode< coord_type, dist_type >::is_required (
            EdgeId id ) const  [inline]
```

**Parameters**

| | |
|---|---|
| *id* | EdgeId id |

**Returns**

true, if the undirected edge {i,j} is required, else false

#### 5.1.3.7 operator>()

```
template<class coord_type , class dist_type >
bool TSP::BranchingNode< coord_type, dist_type >::operator> (
            const BranchingNode< coord_type, dist_type > & rhs ) const
```

Overloading operator > and comparing lowerbounds of two BranchingNodes. Used by priority_queue

**Parameters**

| | |
|---|---|
| *rhs* | |

**Returns**

true if >

#### 5.1.3.8 push_forbidden()

```
template<class coord_type , class dist_type >
bool TSP::BranchingNode< coord_type, dist_type >::push_forbidden (
            EdgeId e )
```

pushes both, e and reverse edge of e to forbidden

**Parameters**

| | |
|---|---|
| *e* | |

**Returns**

false, if it was already forbidden

#### 5.1.3.9 push_required()

```
template<class coord_type , class dist_type >
bool TSP::BranchingNode< coord_type, dist_type >::push_required (
            EdgeId e )
```

pushes both, e and reverse edge of e to required

**Parameters**

| | |
|---|---|
| *e* | |

**Returns**

false, if it was already in required

#### 5.1.3.10 reverse_edge()

```
template<class coord_type, class dist_type>
EdgeId TSP::BranchingNode< coord_type, dist_type >::reverse_edge (
            EdgeId e,
            size_type n ) const  [inline]
```

For an EdgeId e corresponding to an edge (i,j) it returns the edge corresponding to the edge (j,i)

**Parameters**

| | |
|---|---|
| *e* | $= (i, j)$ |
| *n* | size on the tsp instance |

**Returns**

$(j, i) = e'$

#### 5.1.3.11 tworegular()

```
template<class coord_type, class dist_type>
bool TSP::BranchingNode< coord_type, dist_type >::tworegular ( )  [inline]
```

checks if the current tree is 2-regular

**Returns**

true, if T 2-regular

The documentation for this class was generated from the following files:

- header/tsp.hpp
- header/tsp_impl.hpp

## 5.2 Instance Class Reference

### 5.2.1 Detailed Description

Constructor of the filename as an argument. File has to be in TSPLIB format

**Parameters**

| | |
|---|---|
| *filename* | |

The documentation for this class was generated from the following file:

- header/tsp.hpp

## 5.3 TSP::Instance< coord_type, dist_type > Class Template Reference

### Public Member Functions

- **Instance** (const std::string &filename)

- dist_type distance (coord_type x1, coord_type y1, coord_type x2, coord_type y2)
- void compute_optimal_tour ()
- void print_optimal_tour (const std::string &filename)
- size_type **size** () const
- size_type **num_edges** () const
- dist_type **weight** (EdgeId id) const
- const std::vector< dist_type > & **weights** () const
- const dist_type & **length** ()

### 5.3.1 Detailed Description

**template<class coord_type, class dist_type>**
**class TSP::Instance< coord_type, dist_type >**

edge weights and reads the instance from a file in TSPLIB format

**Template Parameters**

| | |
|---|---|
| *coord_type* | Container in which the Coordinates are given. Assumably double |
| *dist_type* | Container in which the distances are given. Assumably double |

### 5.3.2 Member Function Documentation

#### 5.3.2.1 compute_optimal_tour()

```
template<class coord_type , class dist_type >
void Instance::compute_optimal_tour ( )
```

Computes and optimal tour on this Instance and saves it as EdgeIds in _tour

#### 5.3.2.2 distance()

```
template<class coord_type, class dist_type>
dist_type TSP::Instance< coord_type, dist_type >::distance (
            coord_type x1,
            coord_type y1,
            coord_type x2,
            coord_type y2 )  [inline]
```

Distance function in the TSP Instance. Could be also made private, since it's only there at init point.

**Parameters**

| | |
|---|---|
| *x1* | |
| *y1* | |
| *x2* | |
| *y2* | |

**Returns**

rounded $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$

#### 5.3.2.3 print_optimal_tour()

```
template<class coord_type , class dist_type >
void Instance::print_optimal_tour (
            const std::string & filename )
```

Output the optimal tour into a file by TSPLIB rules

**Parameters**

| | |
|---|---|
| *filename* | |

The documentation for this class was generated from the following files:

- header/tsp.hpp
- header/tsp_impl.hpp

## 5.4 TSP::Node Class Reference

A `Node` stores an array of neighbors (via their ids).

`#include <graph.hpp>`

### Public Types

- typedef std::size_t **size_type**

### Public Member Functions

- Node ()

  *Create an isolated node (you can add neighbors later).*
- size_type degree () const
- std::vector< NodeId > const & neighbors () const
- void add_neighbor (NodeId const id)

  *Adds `id` to the list of neighbors of this node.*

### Friends

- class **OneTree**
- class **BranchingTree**

### 5.4.1 Detailed Description

A `Node` stores an array of neighbors (via their ids).

**Note**

The neighbors are not necessarily ordered, so searching for a specific neighbor takes O(degree)-time.

### 5.4.2 Member Function Documentation

#### 5.4.2.1 add_neighbor()

```
void TSP::Node::add_neighbor (
            NodeId const id )
```

Adds `id` to the list of neighbors of this node.

`Node` definitions

**Warning**

Does not check whether `id` is already in the list of neighbors (a repeated neighbor is legal, and models parallel edges).
Does not check whether `id` is the identity of the node itself (which would create a loop!).

#### 5.4.2.2 degree()

```
Node::size_type TSP::Node::degree ( ) const  [inline]
```

**Returns**

The number of neighbors of this node.

#### 5.4.2.3 neighbors()

```
std::vector< NodeId > const & TSP::Node::neighbors ( ) const  [inline]
```

**Returns**

The array of ids of the neighbors of this node.

The documentation for this class was generated from the following files:

- header/graph.hpp
- src/graph.cpp

## 5.5 TSP::OneTree Class Reference

`#include <tree.hpp>`

### Public Member Functions

- OneTree (size_t size)
- void add_edge (NodeId i, NodeId j)
- const size_type & **get_num_edges** () const
- const std::vector< EdgeId > & **get_edges** () const
- const std::vector< Node > & **get_nodes** () const
- const Node & **get_node** (NodeId id) const

### 5.5.1 Detailed Description

1-tree implementation as a container for Nodes and meta information

### 5.5.2 Constructor & Destructor Documentation

#### 5.5.2.1 OneTree()

```
TSP::OneTree::OneTree (
            size_t size )  [inline]
```

Only constructor. Initializes the meta information and edge vector

**Parameters**

| | |
|---|---|
| *size* | size of the tree in terms of Nodes |

### 5.5.3 Member Function Documentation

#### 5.5.3.1 add_edge()

```
void TSP::OneTree::add_edge (
            NodeId i,
            NodeId j )  [inline]
```

Adding an edge by their NodeIds. Requires the to_Edge functionality from

**Parameters**

| | |
|---|---|
| *i* | first node |
| *j* | second node |

The documentation for this class was generated from the following file:

- header/tree.hpp

# Chapter 6

# File Documentation

## 6.1 header/graph.hpp File Reference

This file was used as a graph class header, but someone did not like it and now its only use is to serve his owner with a Node class.

```
#include <cstddef>
#include <iosfwd>
#include <limits>
#include <vector>
```

### Classes

- class TSP::Node

  *A `Node` stores an array of neighbors (via their ids).*

### Namespaces

- TSP

### Typedefs

- using **TSP::size_type** = std::size_t
- using **TSP::NodeId** = size_type
- using **TSP::TSPlibId** = size_type

### 6.1.1 Detailed Description

This file was used as a graph class header, but someone did not like it and now its only use is to serve his owner with a Node class.

## 6.2 header/tree.hpp File Reference

Definition of the OneTree class and general thoughts about a tree structure.

```
#include <cstdlib>
#include <vector>
#include <stdexcept>
#include "graph.hpp"
#include "util.hpp"
```

### Classes

- class TSP::OneTree

### Namespaces

- TSP

### Typedefs

- using **TSP::EdgeId** = size_type

#### 6.2.1 Detailed Description

Definition of the OneTree class and general thoughts about a tree structure.

## 6.3 header/tsp.hpp File Reference

header file for the main TSP framework including the TSP Instance and the BranchingNode templates

```
#include <cstdlib>
#include <fstream>
#include <algorithm>
#include <cmath>
#include <string>
#include <climits>
#include <vector>
#include <sstream>
#include <queue>
#include <numeric>
#include <utility>
#include <cassert>
#include "util.hpp"
#include "tree.hpp"
#include "tsp_impl.hpp"
```

### Classes

- class TSP::BranchingNode< coord_type, dist_type >
- class TSP::Instance< coord_type, dist_type >
- class TSP::Instance< coord_type, dist_type >
- class TSP::BranchingNode< coord_type, dist_type >

### Namespaces

- TSP

### Macros

- #define **EPS** 10e-7

#### 6.3.1 Detailed Description

header file for the main TSP framework including the TSP Instance and the BranchingNode templates

## 6.4 header/tsp_impl.hpp File Reference

Implementation details for the tsp.hpp. One might ask, why we are using a .hpp. Well, templates have to be beforehand which is achieved by putting them into the header. This also holds for their implementation. There is one other way, but this is the way conforming the current ISO.

```
#include <cassert>
#include <fstream>
#include <algorithm>
#include <cmath>
#include <vector>
#include <sstream>
#include <queue>
#include <numeric>
#include "tree.hpp"
```

### Namespaces

- TSP

### Functions

- template<class coord_type , class dist_type >
  void TSP::compute_minimal_1_tree (TSP::OneTree &tree, const std::vector< double > &lambda, const TSP::Instance< coord_type, dist_type > &tsp, const TSP::BranchingNode< coord_type, dist_type > &BNode)
- template<class coord_type , class dist_type >
  dist_type TSP::Held_Karp (const TSP::Instance< coord_type, dist_type > &tsp, std::vector< double > &lambda, TSP::OneTree &tree, const TSP::BranchingNode< coord_type, dist_type > &bn, bool root=false)

### 6.4.1 Detailed Description

Implementation details for the tsp.hpp. One might ask, why we are using a .hpp. Well, templates have to be beforehand which is achieved by putting them into the header. This also holds for their implementation. There is one other way, but this is the way conforming the current ISO.

## 6.5 src/graph.cpp File Reference

one small function was not defined inline

```
#include "../header/graph.hpp"
```

### Namespaces

- TSP

### 6.5.1 Detailed Description

one small function was not defined inline

## 6.6 src/main.cpp File Reference

execution of the programm

```
#include <iostream>
#include <cstring>
#include <ctime>
#include "../header/tsp.hpp"
```

### Functions

- int **main** (int argc, char ∗argv[ ])

### 6.6.1 Detailed Description

execution of the programm

# Index